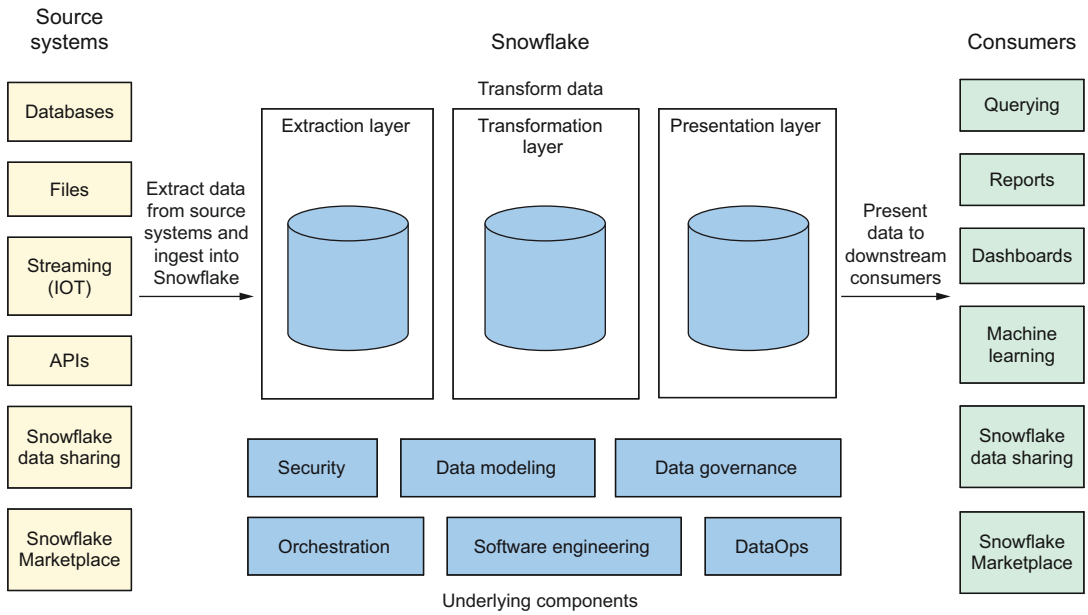
Snowflake DATA ENGINEERING



Maja Ferle
Foreword by Joe Reis

MANNING

Data engineering building blocks with Snowflake



The main building blocks of data engineering pipelines are

- Extracting data from source systems and ingesting it into Snowflake
- Performing relevant data transformations
- Delivering data to downstream consumers for analytics, reporting, data science, or other use cases

In addition to these main building blocks, data pipelines include underlying components such as security, data modeling, data governance, software engineering, orchestration, and DataOps.

Snowflake Data Engineering

Snowflake Data Engineering

MAJA FERLE
FOREWORD BY JOE REIS



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2025 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The authors and publisher have made every effort to ensure that the information in this book was correct at press time. The authors and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.


Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Katie Sposato Johnson
Technical editor:	Daan Bakboord
Review editor:	Radmila Ercegovac
Production editor:	Kathy Rossland
Copy editor:	Kari Lucke
Proofreader:	Melody Dolab
Technical proofreader:	Rohan Pasalkar
Typesetter:	Dennis Dalinnik
Cover designer:	Marija Tudor

ISBN: 9781633436855
Printed in the United States of America

brief contents

PART 1 INTRODUCING DATA ENGINEERING WITH SNOWFLAKE 1

1 ■ Data engineering with Snowflake 3
2 ■ Creating your first data pipeline 17

PART 2 INGESTING, TRANSFORMING, AND STORING DATA 35

3 ■ Best practices for data staging 37
4 ■ Transforming data 58
5 ■ Continuous data ingestion 83
6 ■ Executing code natively with Snowpark 102
7 ■ Augmenting data with outputs from large language models 125
8 ■ Optimizing query performance 144
9 ■ Controlling costs 169
10 ■ Data governance and access control 189

PART 3 BUILDING DATA PIPELINES 205

11 ■ Designing data pipelines 207
12 ■ Ingesting data incrementally 224
13 ■ Orchestrating data pipelines 248
14 ■ Testing for data integrity and completeness 272
15 ■ Data pipeline continuous integration 297

contents

foreword xiv
preface xv
acknowledgments xvii
about this book xix
about the author xxiii
about the cover illustration xxiv

PART 1 INTRODUCING DATA ENGINEERING WITH SNOWFLAKE 1

1 *Data engineering with Snowflake* 3

1.1 Snowflake for data engineering 4

Snowflake architecture 4 ▪ *Snowflake features for data engineering* 5

1.2 Responsibilities of a Snowflake data engineer 6

Extracting data from source systems 8 ▪ *Performing data transformations* 9 ▪ *Presenting data to downstream consumers* 10 ▪ *Applying underlying components* 10

1.3 Building data pipelines 13

1.4 Data engineering with Snowflake applications 14

2 *Creating your first data pipeline 17*

2.1 Setting up your Snowflake account 19

2.2 Staging a CSV file 20

2.3 Loading data from a staged file into a target table 22

Loading data from a staged file into a staging table 24
Merging data from the staging table into the target table 26

2.4 Transforming data with SQL commands 28

2.5 Automating the process with tasks 29

PART 2 INGESTING, TRANSFORMING, AND STORING DATA 35

3 *Best practices for data staging 37*

3.1 Creating external stages 39

Configuring a storage integration 40 ▪ Creating an external stage using a storage integration 43 ▪ Creating an external stage using credentials 44 ▪ Loading data from staged files into a staging table 45 ▪ Avoiding duplication when loading data from staged files 47 ▪ Using a named file format 48

3.2 Viewing stage metadata with directory tables 50

3.3 Preparing data files for efficient ingestion 51

File sizing recommendations 51 ▪ Organizing data by path 52

3.4 Building pipelines with external tables 53

Querying data in external stages with external tables 54
Using materialized views to improve query performance 55

4 *Transforming data 58*

4.1 Ingesting semistructured data from cloud storage 60

Creating a storage integration 61 ▪ Creating an external stage 63 ▪ Examining the JSON structure 63 ▪ Ingesting JSON data into a VARIANT data type 65

4.2 Flattening semistructured data into relational tables 66

4.3 Encapsulating transformations with stored procedures 70

Creating a basic stored procedure 73 ▪ Including a return value in a stored procedure 74 ▪ Implementing exception handling in stored procedures 75

4.4 Adding logging to stored procedures 75
4.5 Building robust data pipelines 78

5 *Continuous data ingestion 83*
5.1 Comparing bulk and continuous data ingestion 85
5.2 Preparing files in cloud storage 85
Creating a storage integration 86 ▪ Creating an external stage 87
5.3 Configuring Snowpipe with cloud messaging 89
Configuring event grid messages for blob storage events 90 Creating a notification integration 92 ▪ Creating a pipe object 93 ▪ Ingesting data continuously 95 ▪ Flattening the JSON structure to relational format 96
5.4 Transforming data with dynamic tables 98

6 *Executing code natively with Snowpark 102*
6.1 Introducing Snowpark 104
6.2 Creating a Snowpark procedure in a worksheet 105
6.3 Using the SQL API from a local development environment 110
Installing and configuring the local development environment 110 Creating a Snowflake session 110 ▪ Providing credentials in a configuration file 111 ▪ Querying data and executing SQL commands 113
6.4 Generating a date dimension in Snowpark Python 113
6.5 Working with data frames 115
6.6 Ingesting data from a CSV file into a Snowflake table 118
6.7 Transforming data with data frames 121

7 *Augmenting data with outputs from large language models 125*
7.1 Configuring external network access 126
7.2 Calling an API endpoint from a Snowpark function 129
Constructing the UDF that retrieves customer reviews 130 Interpreting the results from the UDF 132 ▪ Storing the customer reviews in a table 134
7.3 Deriving customer review sentiments 135

7.4 Interpreting order emails using LLMs to save time 138

Creating a stored procedure that interprets customer emails 138 Constructing the prompt 139 ▪ Saving the CSV result to a table 140 ▪ Evaluating the output 141

8 *Optimizing query performance 144*

8.1 Getting data from the Snowflake Marketplace 145

8.2 Performing analysis of geographical data 148

Snowflake's geography functions 148 ▪ Copying data from the shared database 150 ▪ Viewing query execution parameters using the query profile 152

8.3 Understanding Snowflake micro-partitions 154

A conceptual example of micro-partitions 154 ▪ Micro-partition pruning 156

8.4 Optimizing storage with clustering 157

Viewing clustering information 157 ▪ Adding clustering keys to a table 159 ▪ Monitoring the clustering process 160 ▪ Viewing improved query execution after clustering 161

8.5 Improving query performance with search optimization 162

Adding search optimization to a table 163 ▪ Reviewing query performance after adding search optimization 164

8.6 General tips for improving query performance 165

Writing efficient SQL queries 165 ▪ Identifying queries that are candidates for optimization 166

9 *Controlling costs 169*

9.1 Understanding Snowflake costs 170

Total Snowflake cost 171 ▪ Compute resources cost 172 Virtual warehouse credits 172

9.2 Sizing virtual warehouses 173

Using persisted query results 176 ▪ Comparing query statistics between differently sized warehouses 177 ▪ Optimizing query performance to reduce spilling 179

9.3 Optimizing performance with data caching 181

Illustrating the metadata cache 182 ▪ Utilizing the warehouse cache efficiently 183

9.4 Reducing query queuing 183

Examining queuing 184 ▪ Limiting concurrently running queries 185

9.5 Monitoring compute consumption 186

10 *Data governance and access control 189*

10.1 Role-based access control 190

System-defined roles 191 ▪ Custom roles 191 Designing RBAC 192

10.2 Securing data with row access policies 197

10.3 Protecting sensitive data with masking policies 201

PART 3 BUILDING DATA PIPELINES 205

11 *Designing data pipelines 207*

11.1 Designing data pipelines 208

Extracting data 208 ▪ Comparing data pipeline patterns 209 Choosing data transformation layers 212 ▪ Organizing data warehouse layers 213 ▪ Creating schemas with access control 215

11.2 Building a sample data pipeline 216

Implementing the extraction layer 216 ▪ Implementing the staging layer 218 ▪ Implementing the data warehouse layer 220 Implementing the reporting layer 222

12 *Ingesting data incrementally 224*

12.1 Comparing data ingestion approaches 226

Full ingestion 226 ▪ Incremental ingestion 227

12.2 Preserving history with slowly changing dimensions 228

SCD type 2 228 ▪ Append-only strategy 229 ▪ Designing idempotent data pipelines 230

12.3 Detecting changes with Snowflake streams 230

Ingesting files from cloud storage incrementally 231 ▪ Preserving history when ingesting data incrementally 236

12.4 Maintaining data with dynamic tables 242

Deciding when to use dynamic tables 243 ▪ Querying historical data 244

13 *Orchestrating data pipelines 248*

13.1 Orchestrating with Snowflake tasks 250

Creating a schema to store the orchestration objects 251 Designing the orchestration tasks 252 ▪ Creating tasks with dependencies 253

13.2 Sending email notifications 257

13.3 Orchestrating with task graphs 258

Designing the task graph 259 ▪ Creating the root task 261 Creating the finalizer task 262 ▪ Viewing the task graph 263

13.4 Monitoring data pipeline execution 264

Adding logging functionality to tasks 264 ▪ Summarizing logging information in an email notification 266

13.5 Troubleshooting data pipeline failures 269

14 *Testing for data integrity and completeness 272*

14.1 Data testing methods 273

Performing data testing as steps in the pipeline 273 ▪ Performing data testing independently of the pipeline 275

14.2 Incorporating data testing steps in the pipeline 275

Constructing the partner data quality task 277 ▪ Constructing the product data quality task 280 ▪ Executing the pipeline with the data testing tasks 280

14.3 Applying the Snowflake data metric functions 282

System-defined data metric functions 283 ▪ User-defined data metric functions 284 ▪ Viewing data metric function details 286

14.4 Alerting users when data metrics exceed thresholds 287

14.5 Detecting data volume anomalies 289

Generating random data 290 ▪ Displaying data as a line chart in Snowsight 292 ▪ Working with the anomaly detection model 292

15 *Data pipeline continuous integration 297*

15.1 Separating the data engineering environments 299

15.2 Database change management 300

Comparing the imperative and the declarative approach to DCM 300 ▪ Organizing the code in the repository 302

15.3 Configuring Snowflake to use Git 303

Creating a Git repository stage 304 ▪ Executing commands from a Git repository stage 307

15.4 Using the Snowflake CLI command line interface 308

Installing and configuring Snowflake CLI 308 ▪ Executing scripts with Snowflake CLI 310 ▪ Continuous integration with Snowflake CLI 311

15.5 Connecting to Snowflake securely 313

Configuring key-pair authentication 313

15.6 Applying what we learned in real-world scenarios 314

appendix A Configuring your Snowflake environment 317
appendix B Snowflake objects used in the examples 321

index 333

foreword

Data engineering has witnessed a massive transformation with the advent of cloud data platforms. At the forefront of this transformation (no pun intended) is Snowflake. The ability to seamlessly scale, compute and store, share and collaborate on data, and run AI workloads all in one simple-to-use platform has made Snowflake the platform of choice for data engineers working at businesses seeking to unlock the true potential of their data.

With the explosion of data and AI, data engineers have never been more critical. Yet data engineers must constantly improve their knowledge and skills to stay ahead. This book, *Snowflake Data Engineering*, serves as an indispensable guide for aspiring and seasoned data engineers alike, providing a comprehensive and practical exploration of the art and science of data engineering within the Snowflake ecosystem.

Whether you are new to Snowflake or seeking to expand your data engineering expertise, this book will serve as your trusted companion on your journey to mastering the art of data engineering in the cloud. Maja's done a fantastic job of writing this book, which is the perfect blend of technical explanations and practical examples. By the end of this book, you'll be well-prepared to tackle the challenges of the ever-evolving data landscape and empower your organization to make informed decisions and achieve its strategic objectives.

Enjoy!

—Joe Reis
Author, Data Engineer, "Recovering Data Scientist"

preface

After years of using on-premises data analytics technologies, I was intrigued when Snowflake emerged. The concept of a cloud-provisioned database platform without the hassles of physical installation, sizing, purchasing, and upfront costs was captivating.

My first experience with a Snowflake project was a revelation. The ease with which I could set up the infrastructure without needing heavy initial investment and commitment was a game-changer. The pay-as-you-go model further added to the flexibility and cost-effectiveness of the platform.

Over the years, Snowflake has evolved into a comprehensive platform that can handle a wide range of data-related tasks. It combines storage, elastic compute, built-in AI capabilities, native applications with Python and Streamlit, data sharing, and integration with third-party services and tools for data insights. With all these features hosted on a single platform, you can establish governance to manage security, compliance, privacy, and access to the data and applications.

Writing a detailed book about all the features that Snowflake provides, complete with real-world examples and exercises, would be overwhelming for an author to write and for the reader to comprehend. Therefore, when I discussed the book's content with the publisher, I asked myself: What would be the first step for someone starting to use Snowflake? Since Snowflake stores data at its core, the initial task would be to bring data into the platform to enable other functionalities. This process falls under data engineering, which involves building data pipelines that ingest data from the source, transform it as needed, and deliver it to downstream consumers for analytics.

Snowflake is hosted in the cloud, so the data ingestion process differs from traditional on-premises databases. You can't just install an ODBC driver and select the data as you would in an on-premises environment. Instead, you need to know how to retrieve data from cloud object storage, Snowflake data sharing, APIs, or third-party tools and connectors. I cover that in this book, along with detailed explanations and examples.

acknowledgments

This book would not have been possible without my acquisition editor, Jonathan Gennick. It is my second time working with him. I trust his judgment regarding decisions about what to include and exclude in the book, understanding the target audience, and constructing the table of contents.

My development editor, Katie Sposato Johnson, was there every step of the way, providing meaningful feedback, suggestions, and encouragement. She had a keen eye for pointing out sections where I veered off course during the writing process and steered me back on track. Thank you, Katie, for always being there when I needed your input.

Thanks to my technical editor, Daan Baakboord, who was meticulous in his technical review, checking and rerunning my code, and providing valuable feedback and suggestions for improvement during the writing process. Daan Bakboord is a Snowflake AI Data Cloud Consultant. He is one of the first Snowflake Data Superheroes in the world. Daan, it was wonderful having you on board!

As I started writing this book, I relied on various sources such as books, blogs, training materials, and whitepapers. Some of the sources I used included *Snowflake: The Definitive Guide* by Joyce K. Avila and the "Snowflake for Data Engineering" training course by Tomáš Sobotík. I want to express my gratitude to Joyce and Tomáš for being two of my early reviewers. Their expertise and feedback on the initial chapters of my manuscript were invaluable and provided the encouragement I needed to continue. Additionally, I consulted *Fundamentals of Data Engineering*, which was coauthored by Joe Reis, and I'm thankful to Joe for agreeing to write the foreword.

I am also grateful to my technical proofreader, Rohan Pasalkar, who was very helpful in reviewing the code for the exercises in the book.

I want to thank the entire Manning Publications team for their fantastic work preparing the book. This includes the marketing, graphics, and production teams, some of whom I never met and whose names I don't know. Thanks to all the reviewers who read my manuscript in various stages of completion and provided comments from the point of view of the target reader: Alain Couniot, Albert Nogués, Andriani Stylianou, Ankit Virmani, David Allen Blubaugh, David Krief, Doyle Turner, Emanuele Piccinelli, Eros Pedrini, Gabor Gollnhofer, Hilde Van Gysel, Jesús Antonino Juárez Guerrero, Jonathan Woodard, Krzysztof Kamyczek, Luke Kupka, Madiha Khalid, Nadir Doctor, Oliver Korten, Pavel Filatov, Peter G. Bishop, Rambabu Posa, Richard B. Ward, Sambasiva Andaluri, Satej Sahu, Sean Booker, Simone Sguazza, Shivani Mayekar, Sriram Macharla, Tobias Kaatz, and Ubaldo Pescatore. Their feedback also helped shape the book's contents.

I appreciate my employer, In516ht, for allowing me to work on exciting projects where I could develop my Snowflake data engineering skills.

Finally, I thank Snowflake for providing us with an extraordinary cloud data platform.

about this book

Data engineering is the practice of building solutions that extract data from source systems, transform the data into useful information, and present the harmonized data to users for downstream consumption. Data engineers are responsible for building data pipelines that enable data analysts, data scientists, and other users to access the data they need to do their jobs. Providing high-quality data on time is essential for effective analytics, which is why data engineers play a critical role in the data analytics domain.

This book teaches data engineering skills on the powerful Snowflake platform. It starts by guiding you in building your first simple data pipeline and then expands the pipeline with increasingly complex features, including performance optimization, data governance, security, orchestration, and augmenting your data with generative AI.

After reading this book and completing the included exercises, you will be able to

- Ingest data into Snowflake from cloud object store providers, from the Snowflake Marketplace, or from API calls
- Transform structured and semistructured data within Snowflake using functions, stored procedures, and SQL
- Optimize performance and cost when ingesting data into Snowflake
- Design role-based access control and data governance features to secure your data against unauthorized use
- Orchestrate data pipelines with streams and tasks and monitor their execution
- Use Snowpark for development in programming languages such as Python

- Deploy Snowflake objects and code using continuous integration and continuous deployment principles
- Augment your data with generative AI

Who should read this book

This book is for readers who have some familiarity with Snowflake, such as navigating the Snowsight user interface and using worksheets to execute queries and commands. The readers should have a basic understanding of data warehousing and data ingestion techniques. Previous use of ETL or ELT technologies for data ingestion is beneficial but not required.

Since Snowflake is a relational database, knowledge of SQL querying, including data definition language (DDL) and data manipulation language (DML) operations, is vital. Depending on the reader's preference, if they plan to use Snowpark with Python, they must also know how to write Python code. If they plan to stage data for loading, they must know how to set up a cloud object store bucket/container and upload files with any of the supported providers (AWS S3, Azure blob storage, GCP Google Cloud Storage).

How this book is organized: A road map

This book has 15 chapters organized in three parts.

Part 1 prepares you for your journey into Snowflake data engineering:

- Chapter 1 introduces Snowflake as a modern cloud data platform for various data-related tasks such as data storage, processing, application development, and analytics. Snowflake is a popular choice for solutions encompassing data warehousing, data lakes, data analytics, and data science.
- In chapter 2, you will begin your data engineering journey by creating your first data pipeline in Snowflake. This pipeline involves extracting data from a CSV file into Snowflake, transforming the raw data into the required data model for reporting, and automating the process.

Part 2 expands your initial data pipeline and explores the more advanced aspects of Snowflake data engineering:

- In chapter 3, you'll learn how to ingest data from a cloud storage provider and create external stages in Snowflake. The chapter explains and compares different approaches to ingesting files and offers tips on preparing data files in cloud storage for efficient ingestion.
- In chapter 4, you'll ingest semistructured data in JSON format and flatten it into a relational structure. You will add exception handling and logging to the data pipeline to ensure its resilience against unintended errors.
- In chapter 5, you'll build a new data pipeline that continuously ingests data from files as soon as they appear in the external cloud storage with minimum

delay. The chapter introduces Snowflake features like Snowpipe for continuous data ingestion and dynamic tables for continuous data transformation.

- Chapter 6 covers Snowpark, which consists of libraries and code execution environments that allow Python and other programming languages to run natively in Snowflake.
- In chapter 7, you'll immerse yourself in generative AI and large language models (LLMs). You will learn how to call external API endpoints from Snowflake and use Snowflake's own Cortex LLM functions to enhance your data pipelines.
- In chapter 8, you will utilize Snowflake's query profile tool to understand the mechanics of query execution and identify opportunities for optimizing query performance when dealing with large data volumes.
- Chapter 9 explores what contributes to Snowflake's cost and how to monitor credit consumption. It explains Snowflake virtual warehouses and how they affect query performance.
- In chapter 10, you will learn Snowflake role-based access control and data governance features, such as row access policies and masking policies, that limit data access to authorized users.

Part 3 consolidates all your knowledge gained so far and demonstrates how to build a comprehensive data pipeline that executes on schedule:

- Chapter 11 lays the groundwork for a comprehensive data pipeline by defining the data transformation layers, including extract, staging, data warehouse, and presentation.
- Chapter 12 introduces incremental data ingestion, which is faster than full ingestion, as it involves moving and storing less data, resulting in reduced storage and compute costs.
- Chapter 13 explains data pipeline orchestration as the process that involves scheduling, defining dependencies, error handling, and sending notifications to ensure efficient execution of data pipeline steps.
- In chapter 14, you will learn how to conduct data quality tests that validate data integrity and completeness and take remedial measures when test results don't meet the data quality standards.
- Chapter 15 covers continuous integration, a software development practice in which data engineers frequently merge their code changes into the repository. After the merge, automated scripts execute the code, create database objects, perform integration tests, and carry out other necessary actions.

Each of the chapters explains the topics and emphasizes the learning with code examples that you can execute. The chapters follow a logical sequence and often build on the previous chapters. It is recommended that you read the chapters in sequence.

About the code

Each of the chapters in the book, except chapter 1, include code examples that can be executed in a Snowflake account. Appendix A provides instructions for creating a free trial Snowflake account that you can use to execute the code.

The source code is explained in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (➥). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

You can get executable snippets of code from the liveBook (online) version of this book at https://livebook.manning.com/book/snowflake-data-engineering. The complete code for the examples in the book is available for download from the Manning website at https://www.manning.com/books/snowflake-data-engineering and from GitHub at https://github.com/mferle/snowflake-data-engineering.

liveBook discussion forum

Purchase of *Snowflake Data Engineering* includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to https://livebook.manning.com/book/snowflake-data-engineering/discussion. You can also learn more about Manning's forums and the rules of conduct at https://livebook.manning.com/discussion.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest her interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

Snowflake is constantly evolving, and commands may have changed since this book was written. Each chapter directs the reader to the relevant sections of the official Snowflake documentation, which provides detailed and up-to-date information. The main Snowflake documentation page is available at https://docs.snowflake.com/.

The Snowflake Community at https://community.snowflake.com/ is a hub where you can meet like-minded developers, ask questions, and share information related to your Snowflake journey.

about the author



MAJA FERLE is a seasoned data architect with more than 30 years of experience in data analytics, data warehousing, business intelligence, data engineering, data modeling, and database administration. As a consultant, she has delivered data projects in diverse environments across the globe, always seeking to get her hands on the latest technologies and methodologies. Since embarking on the Snowflake Data Cloud, Maja has served as data architect and data engineer on several successful cloud migration projects. She holds the SnowPro Advanced Data Engineer and the SnowPro Advanced Data Analyst certifications. She is also a Snowflake Subject Matter Expert and a Snowflake Data Superhero.

about the cover illustration

The figure on the cover of *Snowflake Data Engineering* is "Sabioncelline," or "Woman from Pelješac," taken from Balthasar Hacquet's *L'Illyrie et la Dalmatie*, published in 1815.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

Part 1

Introducing data engineering with Snowflake

This part of the book will prepare you for your journey into Snowflake data engineering.

Chapter 1 will cover Snowflake as a modern cloud data platform for various data-related tasks, such as data storage, processing, application development, and analytics. Snowflake is a popular choice for solutions encompassing data warehousing, data lakes, data analytics, and data science.

In chapter 2, you will begin your data engineering journey by creating your first data pipeline in Snowflake. This pipeline will involve extracting data from a CSV file into Snowflake, transforming the raw data into the required data model for reporting, and automating the process.

By the end of this part of the book, you will have a solid understanding of the fundamental Snowflake features needed to extract and transform data from a file into Snowflake. This will serve as a foundation for the next part where you will enhance the pipeline with additional functionality.

1 Data engineering with Snowflake

This chapter covers

- Featuring Snowflake for data engineering
- Examining the responsibilities of a Snowflake data engineer
- Constructing data pipelines with Snowflake
- Data engineering with Snowflake applications

Organizations in just about every industry collect data, often in massive amounts. Data analysts, data scientists, and other business professionals use this data to gain insights that aid decision-making. Raw data is rarely suitable for consumption directly from the source. Consumers require data that is transformed according to business rules, reshaped to fit a particular business need, and optionally enriched with external data.

Data engineering is the practice of building solutions that extract data from source systems, transform the data into useful information, and present the harmonized data to users for downstream consumption. The abbreviations ETL (extract, transform, load) or ELT (extract, load, transform) are also commonly used to denote these solutions.

Data engineers are responsible for building data pipelines that enable data analysts, data scientists, and other users to access the data they need to do their jobs. Providing high-quality data on time is essential for effective analytics, so data engineers play a critical role in the data analytics domain.

1.1 Snowflake for data engineering

Organizations usually extract data from source systems, store it in a data warehouse, and present it to users as dimensional data marts. Additional scenarios include building data lakes, data marts, or domain-oriented data products. In each scenario, data is extracted from source systems, ingested into a target storage platform, and transformed for use by downstream consumers.

Snowflake is a modern cloud data platform for data storage, processing, application development, and analytics. It's a popular choice for solutions encompassing data warehousing, data lakes, data analytics, or data science. The following sections explain the Snowflake capabilities that support data engineering in more detail.

> **TIP** Snowflake is constantly evolving and adding new features. The most up-to-date information is available on Snowflake's website at https://www.snowflake.com/.

1.1.1 Snowflake architecture

Snowflake is a SaaS (software as a service) solution provisioned in the cloud. The currently supported cloud providers are Amazon Web Services, Microsoft Azure, and Google Cloud Platform. When users choose Snowflake as their cloud data platform, they don't have to buy, install, configure, manage, or maintain hardware and software.

In the Snowflake architecture, storage and compute are managed separately. Data is stored once and accessed by multiple workloads, such as data engineering pipelines, query engines, data science solutions, data sharing, or analytics.

Snowflake stores data in cloud storage in a compressed columnar format. It manages the data storage organization, structure, compression, metadata, statistics, and security. The data objects are accessible to users through SQL queries.

Users execute queries using virtual warehouses. Each virtual warehouse consumes its own compute resources without affecting other warehouses. The number of resources for each warehouse is configurable. Snowflake can also scale the resources up or down depending on desired performance and workload.

Snowflake coordinates its different components through the cloud services layer. The services include authentication, access control, infrastructure management, query optimization, metadata management, and security. Figure 1.1 shows the three layers of the Snowflake architecture: database storage, query processing, and cloud services.

Snowflake provides powerful storage features, such as *zero-copy cloning* and *time travel*. Zero-copy cloning makes copies of data without incurring additional storage costs. It allows data engineers to create new environments quickly. For example, they

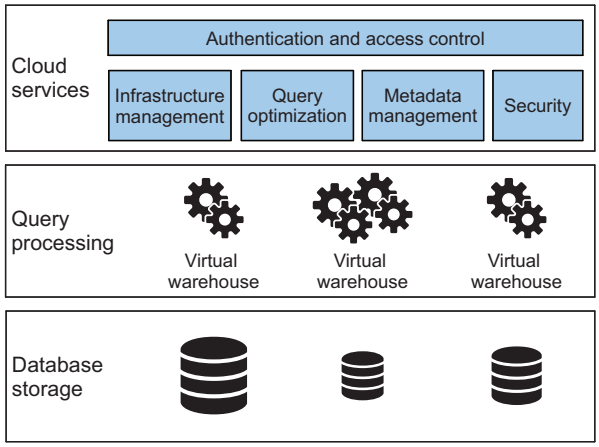


Figure 1.1 The three layers of the Snowflake architecture are database storage, query processing, and cloud services.

can clone a development environment into a test environment or clone a production environment into a development environment for new feature development.

Time travel can protect data against unintentional errors or malicious acts. It allows users to access data at a point in time in the past and is an efficient way to keep data backups within a configurable time.

Snowflake supports replicating databases between Snowflake accounts across regions and cloud platforms. This feature ensures the replicated data is in sync with the originating data. It also brings data closer to consumers in a chosen geographic region to reduce query latency and serves as a backup.

1.1.2 Snowflake features for data engineering

Snowflake supports various features for data ingestion and transformation. These features are described in the following sections.

Ingesting data from cloud storage providers

Users can ingest data from files stored in the supported cloud storage providers: AWS S3 buckets, Azure blob storage containers, and GCP buckets. Files may be in structured or semistructured formats, such as CSV, JSON, XML, Parquet, and others. Snowflake also enables users to work with unstructured data files that contain images, video, or audio. Users can share file access URLs and load the URLs and file metadata into Snowflake tables.

Snowpipe

Snowpipe is a service that supports continuous data ingestion. This service ingests files from cloud storage into Snowflake tables as soon as new files become available in cloud storage. Continuous ingestion is made possible by integrating with the cloud provider notification service, which notifies Snowpipe that new files are available for ingestion.

SNOWFLAKE TASKS

Snowflake supports creating tasks. A task executes SQL statements, stored procedures, or Snowflake scripting blocks according to a recurring schedule. Tasks can depend on other tasks, enabling the implementation of complex data processing pipelines.

SNOWPARK

Snowpark provides API libraries that allow data engineers to build data pipelines in one of the supported programming languages, including Python, Scala, or Java. The code written in these programming languages is executed natively in Snowflake. For example, data engineers can build data pipelines using a programming language, such as Python, and include open source libraries for additional functionality when required.

DATA GOVERNANCE FEATURES

Data stored in Snowflake can contain sensitive information that can only be exposed to authorized users. In multitenant architectures, data from different organizations or business units is stored in a single database. Still, data access is restricted to users from the same organization. To serve these use cases, Snowflake provides data governance features such as masking policies and row-level access policies that prevent unauthorized users from accessing sensitive or confidential data.

SNOWFLAKE MARKETPLACE

The Snowflake Marketplace is a platform where users can discover and access data sets published by other organizations. Users can choose relevant data sets and use them to enrich their data for further analyses. They can also publish their data sets to make them available to others. Data sets can be shared publicly or privately, allowing secure data sharing and collaboration within organizations or externally.

SNOWFLAKE PARTNER CONNECT

With its rich ecosystem of business partners, Snowflake seamlessly integrates with many popular third-party tools and services from many categories, such as business intelligence, data integration, machine learning, data science, security, data governance, and observability. Users employ these tools to ingest data in data engineering use cases, perform data analyses, build reports and dashboards, or work with machine learning and data science algorithms.

With Snowflake, data engineers can choose from numerous features to build data pipelines according to their preferences, architectural direction, and business needs.

1.2 Responsibilities of a Snowflake data engineer

Data engineers are responsible for designing, building, orchestrating, optimizing, monitoring, and maintaining pipelines that extract raw data from source systems and transform it into actionable information for downstream consumption. They ensure that data is readily available, secure, trustworthy, and accessible to consumers based on their requirements.

Professionals with prior experience usually fulfill the data engineer role because it requires a broad range of responsibilities. Many data engineers have a software

engineering or business intelligence analyst background and add more skills as needed. When working in a Snowflake environment, they gain knowledge and experience with Snowflake features.

Knowledge of cloud computing is essential when using Snowflake. For example, data engineers frequently work with cloud object storage, and they understand how to connect to cloud environments securely. Many organizations are shifting toward the cloud and have their infrastructure hosted on cloud providers, meaning that data engineers are becoming comfortable navigating the cloud environment.

Data engineers are familiar with Snowflake storage organization to write efficient SQL queries to retrieve and transform data. They understand how data is stored in Snowflake micro-partitions and how and when to apply clustering, search optimization, query acceleration, or other features to improve query performance.

They also know how to utilize Snowflake virtual warehouses cost-effectively and understand how their solutions affect compute consumption. Virtual warehouses can also help to improve query performance because they maintain a data cache that other queries executed in the same warehouse reuse.

When building solutions in enterprise environments, data engineers apply software engineering principles, such as coding best practices, automated testing, code versioning, continuous integration, and continuous deployment where applicable. If they use Snowpark, they also know a programming language, like Python.

Knowing how to connect to Snowflake securely is crucial to avoid security breaches or unintentionally exposing sensitive data to unauthorized users. Detailed knowledge of the various methods of connecting to Snowflake is essential for all data engineers.

Figure 1.2 illustrates the overall data engineering components for building data engineering pipelines within the Snowflake data cloud platform. Not all components are relevant to every project. Depending on the type of solution that they are building, data engineers only use those components that are relevant and required in the current environment. The main components of data engineering are

- Extracting data from source systems and ingesting it into Snowflake
- Performing data transformations that are relevant to the solution
- Presenting data to downstream consumers for analytics, reporting, data science, or any other use case

In addition to the main components, data engineers also incorporate underlying components when building data engineering pipelines. The underlying components include

- Security
- Data modeling
- Data governance
- Software engineering
- Orchestration
- DataOps

Each of these components is explained in more detail in the following sections (see figure 1.2).

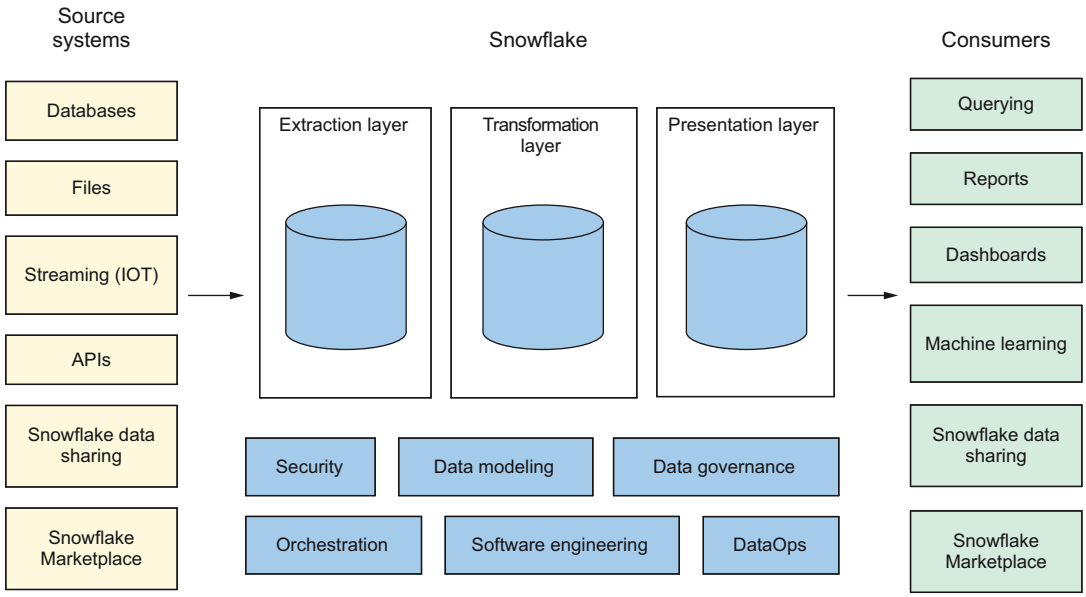


Figure 1.2 Data engineering components with Snowflake include extracting data from source systems, ingesting the data into Snowflake, performing transformations, and presenting the data to downstream consumers, incorporating underlying components and best practices related to security, data modeling, data governance, orchestration, software engineering, and DataOps.

1.2.1 Extracting data from source systems

The first step in the data engineering pipeline is extracting data from source systems. Data engineers gain a basic understanding of the business meaning of the data and how it is used in downstream solutions. Often, more than one source system provides data for the solution, and data engineers build pipelines that seamlessly ingest the data into a single platform with a unified model for the consumers.

Data engineers familiarize themselves with the technology used in the source systems—for example, whether the data sets exist as tables in databases or files in cloud storage, are derived as the output of API calls from applications, originate as streaming data from IOT devices, or anything else. They identify the best way to access data and prepare it for ingesting into Snowflake. In addition to external data sources, data may be available in the form of Snowflake secure data sharing or the Snowflake Marketplace.

INCREMENTAL DATA INGESTION

Data pipelines usually extract data from source systems incrementally by ingesting only new and changed data. Data engineers work together with the owners of the

source systems to determine the best approach to detecting changes. They filter on timestamps or use database change tracking mechanisms or other methods. When extracting data from legacy source systems whose owners are unavailable, the data engineers sometimes figure out the best approach to detecting changes using their judgment and expertise.

When working with systems that don't support easy identification of data changes, data engineers can use Snowflake *streams*. A stream tracks data manipulation operations, such as inserts, updates, or deletes on the underlying data between two points in time.

Using tools vs. custom coding

When starting a new data engineering project, data engineers collaborate with solution architects and other invested parties to decide on the best tools and technologies for data extraction from source systems into Snowflake. These may be open source or commercial tools or custom-written code using open source or proprietary libraries and technologies.

When custom coding, data engineers understand the source data formats and know the commands and options suitable for ingesting the data into Snowflake. They also know the underlying Snowflake commands and options when using third-party data extraction tools. All tools use these commands under the hood, and data engineers should be able to troubleshoot any problems.

1.2.2 Performing data transformations

Data ingested from source systems is transformed from its raw format into useful, actionable information according to user requirements. Data engineers usually receive the transformation rules from data analysts, system architects, or sometimes the users. In smaller data engineering teams, the data engineers may work with business users to define the data transformation rules.

Tools for data transformation

The tools that data engineers use for data transformation vary. Such tools can perform transformations using SQL queries, stored procedures, or functions. Some tools for extracting data from source systems also support data transformations and can be used for this purpose.

Data engineers can also write code in Snowpark, which enables them to write the transformations in one of the supported programming languages, including Python, Java, or Scala. They can also use popular open source libraries for data transformation.

Data validation

In addition to data transformation, data engineers add data validation steps into the data pipelines so that downstream consumers can be confident that their data is reliable and consistent. Data pipelines often include technical data validation rules, such as checking for primary key or foreign key violations, because Snowflake doesn't enforce primary or foreign key constraints in standard tables. The consumers can

define additional data validation rules. Data that doesn't adhere to these rules can be flagged or excluded from downstream analyses.

1.2.3 Presenting data to downstream consumers

Once data has been ingested from the source systems and appropriately transformed, it is presented to the consumers in a shape and form that is appropriate for the tools and solutions used downstream. Data engineers cater to the business needs of the consumers by delivering data according to expected timeliness and performance.

Data engineers build the appropriate user interfaces where needed, such as APIs if applicable, or present the data in the data model suitable for the tool used for analysis. They also provide metadata to allow better data discoverability—for example, by exposing table and column comments in the database and any other relevant metrics, such as the last time the data was refreshed, the number of ingested records, the data quality metrics if used, and more.

Snowflake SQL queries don't require tuning like in many traditional databases, where database administrators spend significant time tuning queries and applying techniques to improve performance. In Snowflake, most queries execute efficiently without additional tuning efforts. However, there are still occasions where Snowflake administrators and data engineers can improve query performance for queries that have high complexity, use substantial data volumes, or consume too many resources. In this case, data engineers can improve performance by implementing Snowflake features such as clustering, search optimization, materialized views, or dynamic tables.

1.2.4 Applying underlying components

While data engineering, on a high level, involves extracting data from source systems, ingesting it into Snowflake, and transforming it for downstream consumption, there are additional components that data engineers apply as needed.

SECURITY

Whether hosted in the cloud or on-premises, IT solutions can be subject to cybersecurity threats and vulnerabilities. Because Snowflake is hosted in the cloud, additional precautions are necessary to prevent unauthorized access. Snowflake administrators usually secure access by various means, such as restricting IP addresses from which users can access Snowflake or requiring strong authentication methods, like multifactor authentication.

While building pipelines, data engineers inevitably connect to Snowflake at some point. Any credentials used for connecting to Snowflake are stored securely in the cloud provider secrets or locally in a secure location that is never exposed to other users or committed to a repository where others could see it. Data engineers must act responsibly to ensure that Snowflake access credentials are not intentionally or accidentally shared with anyone.

Users should have the minimum privileges necessary to complete a task. Even when data engineers are granted the more powerful Snowflake administration roles

to perform specific tasks, they should not use these roles by default. Instead, they should use their designated development roles and only consciously switch to the more powerful administration roles when required to complete a specific administration task.

Data engineers understand the principles of Snowflake role-based access control and use the proper roles when creating objects in Snowflake. While they may not necessarily be responsible for setting up role-based access control, they are aware of how it has been designed so that they can use it correctly.

Data governance

Most organizations have data governance rules that restrict unauthorized users from accessing confidential or sensitive data. Sometimes, especially in highly secure and regulated environments, data engineers are not given access to the actual production data but work with sample data, data where sensitive information has been masked, or subsets of data where sensitive information has been removed.

Data engineers work with data architects and domain owners to understand the data governance requirements. They adhere to these requirements when building data pipelines. For example, they might store sensitive attributes in separate tables with a more restrictive access policy.

Data modeling

Depending on the size of the organization where data engineers work, they may or may not perform data modeling. Large organizations often have dedicated data modeling experts or teams responsible for creating data models. In smaller organizations, data engineers may perform data modeling as part of their job.

Regardless of who creates the data model, a data engineer understands data modeling concepts to correctly load data into the target data models. For example, data models may require that data pipelines generate surrogate keys during data ingestion.

Some examples of the standard data models in data warehousing and analytics solutions include

- Relational data models for enterprise data warehouses
- Ensemble data models such as data vault or anchor modeling that are particularly useful as enterprise data warehouses when ingesting data from many sources
- Dimensional data models for data marts and reporting tools

Some data models allow common and repeatable data loading patterns that automate the data ingestion. Data engineers implement such loading patterns, which helps them build more standardized and efficient data pipelines.

Orchestration

Most data pipelines run on a schedule agreed upon with the owners of the source systems and the downstream consumers. Depending on the users' needs, pipelines can be scheduled in batches or micro-batches, such as daily, hourly, or every few minutes.

When the system contains multiple pipelines, data engineers design dependencies between pipelines and their components. They can use third-party data orchestration

tools, operating system commands, or Snowflake tasks to schedule pipelines in an organized manner.

Snowflake tasks support creating a directed acyclic graph (DAG), which contains a series of tasks with their dependencies. Each DAG originates from a single root task, and the dependent tasks execute in the same direction without loops. Figure 1.3 shows an example of a DAG of Snowflake tasks.

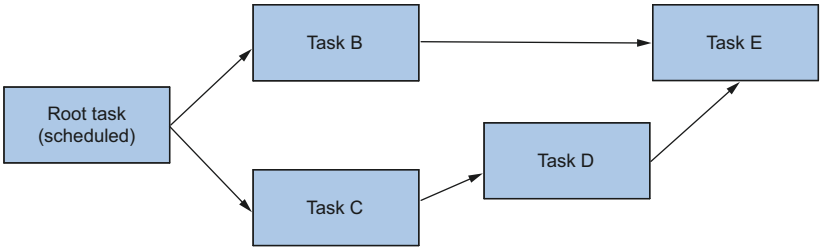


Figure 1.3 A DAG of tasks. The DAG consists of a root task and dependent tasks that start executing when their predecessor completes.

Data engineers monitor scheduled pipelines to ensure they execute without errors and load data as expected. When designing data pipelines, they incorporate logging mechanisms that allow efficient troubleshooting in case of errors. They also design data pipelines that enable restarting from the beginning or starting from the point of failure when execution fails.

SOFTWARE ENGINEERING

Software engineering represents a large chunk of data engineers' work. They build, test, and maintain data engineering pipelines, which are often large systems composed of programming code. Even if most of the code consists of SQL queries instead of traditional programming language coding constructs, software engineering principles like naming conventions, code modularization, code readability, versioning, and system maintainability still apply.

Data engineers also build well-performing, robust, and reliable systems. For example, suppose a data pipeline is accidentally triggered twice. In that case, it shouldn't result in duplicated data, or the pipeline shouldn't execute when the previous instance of the same pipeline is still in progress.

An essential part of data engineering is understanding and utilizing the various environments in the software engineering lifecycle. Organizations typically use a development environment where all the developers work on their code. They then deploy the code to a test environment where it is tested to ensure that the completed pipelines are working as expected. Some large organizations may also have environments for user acceptance testing. Once the code has been tested and approved, it is deployed to the production environment.

DataOps

DataOps is a general term derived from agile software engineering that applies to the data lifecycle, which includes extraction, ingestion, transformation, and presentation. It combines processes and technologies that improve quality and enable continuous development and integration of data engineering pipelines.

Like DevOps in software engineering, DataOps ensures continuous delivery of working software by using tools or technologies that enable team collaboration. For example, data engineering teams use code repositories like Git with CI/CD integration. They can also automate testing of the data pipeline code and data validation. Even in small organizations with only one data engineer, DataOps practices help to organize and structure the work better and ensure consistent and continuous deployment to production.

1.3 Building data pipelines

A *data pipeline* is a set of data-processing components that moves data from the source to the destination. Depending on the requirements, the pipeline may also perform data transformation. Each data pipeline is different. For example, the data engineer ingests files from cloud object storage into Snowflake and presents the data from the files to the downstream consumers for reporting.

Figure 1.4 illustrates this example pipeline. Files from the cloud object storage are made available to Snowflake via an external stage and ingested into a staging table. The pipeline then loads the data into the target table, performs any necessary transformations, and loads the data into the transformed table. Finally, it exposes the data to downstream consumers for further reporting, querying, or analysis.

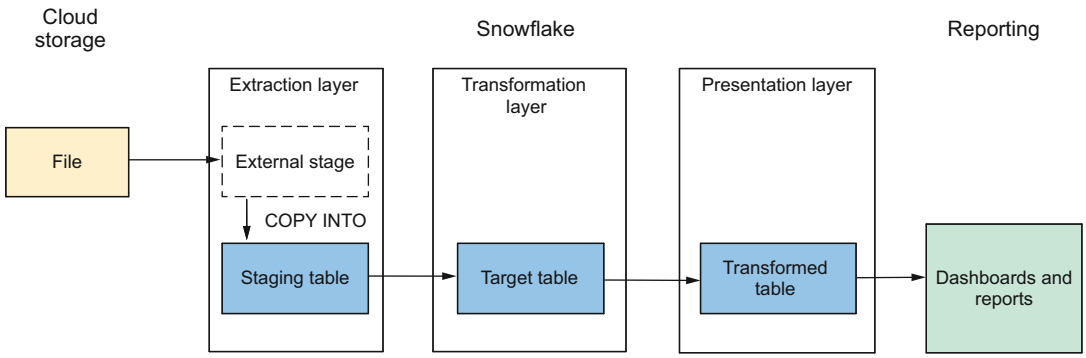


Figure 1.4 Data pipeline that extracts a data file from a cloud storage provider, ingests the data into a Snowflake table, and transforms the data for reporting

The example in figure 1.4 is one of the most common data engineering pipelines in Snowflake. It uses an external stage that points to the cloud object storage location. It then uses the `COPY` command to ingest data from the cloud storage into a Snowflake

staging table. This data is then loaded into the target table by appending it using the `INSERT` command or merging it using the `MERGE` command. The target table is usually designed before the data engineer builds the data pipeline as part of the transformation layer data model.

Any required transformations are performed using SQL statements or stored procedures, and the transformed data is stored in the transformed table. Finally, data is presented in a format that is suitable for reporting. Instead of building data transformation steps in data pipelines, data engineers can create SQL views to transform data because views are easy to maintain and don't require data reloading. However, views may not provide the performance needed by data consumers. In such situations, the data engineer decides on the best approach to improve performance, often by materializing the data.

Many types of data pipelines with various components are used in real-life scenarios. They are described in more detail in subsequent chapters.

1.4 Data engineering with Snowflake applications

The most common data engineering with Snowflake applications are data lakes, data warehousing, data marts, data mesh, business intelligence, data science, and data augmentation using large language models.

DATA LAKE

A *data lake* is a repository that stores large amounts of structured, semistructured, or unstructured data. Snowflake supports storing data in various table types, including standard tables, external tables, hybrid tables, and more. Many concurrent workloads can access the data without resource contention. Users can query the relational or semistructured data in a data lake with SQL or other languages supported in the Snowpark development environment.

DATA WAREHOUSE

A *data warehouse* is an enterprise repository of data collected from one or more source systems or external sources such as the Snowflake Marketplace. The primary purpose of the data warehouse is to provide harmonized historical data for creating reports, dashboards, and data analyses for business users. It allows business users to derive insights from this data, which are the basis for making business decisions.

DATA MART

Data marts store data for a single business unit, such as marketing or finance. The business units can access data in data marts quickly and easily, rather than searching for the information in the data warehouse. Because building a data warehouse can consume significant resources and take a long time to complete, organizations sometimes opt to build only data marts without a data warehouse.

DATA MESH

Data mesh is a more recently introduced approach to data analytics, where—instead of a monolithic data warehouse—individual domain teams build their data products.

These often resemble data marts. With better tools and easier-to-use platforms and technologies, business users can do their data preparation and exploration. For example, they profile the data quality in their business units. They can also perform simple transformations. Thus, some data engineering activities shift from data engineers to business users.

However, even when data engineering activities are transferred to business users, the need for qualified data engineers will continue. Many technical components of the data engineering pipelines require solid software engineering knowledge, which business users might not have, as they have a different focus. Data engineers still play a significant role in a data mesh approach, but the role may move from the IT function to the relevant business function.

Snowflake can host data products, and Snowflake data sharing can expose data products across the organization.

Business intelligence

Business intelligence is the process of analyzing data and deriving actionable information for decision-making. Business intelligence solutions usually get their data from enterprise data warehouses, data marts, data products, or a combination of these.

Data science

Data science is a broad topic that covers applications such as advanced analytics, AI, and machine learning. Generally, data science aims to uncover actionable insights from data sources used for decision-making and strategic planning in organizations. Snowpark enables data scientists to build their models using a programming language of their choice, especially Python, as it has a rich selection of machine learning libraries. For example, data scientists can build, train, and deploy machine learning models within the Snowpark environment with popular libraries like scikit-learn, TensorFlow, or PyTorch. With the introduction of Snowflake Cortex, some machine learning functionalities, such as time-series forecasting, anomaly detection, or classification, can be accessed from Snowflake SQL commands.

Data augmentation using large language models

Generative artificial intelligence (GenAI) is artificial intelligence that generates text, images, programming code, or other output formats. *Large language models* (LLMs) are a subset of GenAI that create text content. Many business applications, such as text classification, document analysis, sentiment classification, and language translation, rely on LLMs. Outputs from LLMs can augment data in various Snowflake solutions.

Developers can access third-party LLM functionality from Snowpark by calling the related API and storing the result in Snowflake tables for further analysis. With the introduction of Snowflake Cortex, selected LLM functionality, like sentiment scoring, prompt completion, text summarization and translation, extracting an answer from unstructured text, or vector embedding, can be accessed from Snowflake SQL commands.

Summary

- Snowflake is a modern cloud data platform well suited for data-intensive solutions such as data lakes, data warehouses, data marts, data mesh, business intelligence, data science, and data augmentation using LLMs.
- Snowflake data engineers are responsible for designing, building, orchestrating, optimizing, monitoring, and maintaining pipelines that take raw data from source systems and transform it into usable information for downstream consumption.
- The first step in the data engineering pipeline is extracting data from source systems. Data engineers gain access to the source systems and ingest data into Snowflake by custom coding Snowflake commands or using a tool.
- Data ingested from source systems into Snowflake is transformed from its raw state into a shape and format suitable for downstream consumers.
- Data transformations can be performed via tools, SQL statements, or stored procedures or coded in Snowpark using one of the supported programming languages.
- Data for downstream consumers is presented in a shape that makes it readily accessible and efficient to use with third-party reporting tools or other solutions, such as machine learning models in Snowpark.
- A data pipeline is a set of data-processing steps that moves data from source to destination and performs transformations as needed.
- When building data pipelines for Snowflake, data engineers apply underlying components, such as security, data governance, data modeling, orchestration, software engineering, and DataOps.
- The most common data engineering with Snowflake applications are data lakes, data warehouses, data marts, data mesh, business intelligence, data science, and data augmentation using LLMs.

2 Creating your first data pipeline

This chapter covers

- Staging a file for ingesting into Snowflake
- Loading data from a staged file into a staging table
- Merging data from a staging table into the target table
- Transforming data with SQL
- Automating the pipeline with a task

In this chapter, you will learn how to build your first data pipeline in Snowflake. We will ingest data from a CSV file into Snowflake, transform the data from its raw form into a target data model required for reporting, and automate the pipeline. This chapter does not include underlying data pipeline components introduced in chapter 1, such as security, data governance, software engineering, or DataOps, because we want to keep it simple initially. These topics are described in more detail in subsequent chapters. To get started with data engineering, we will create a simple data pipeline that will illustrate the core Snowflake functionality required to ingest data from a file into Snowflake and transform the data.

By the end of this chapter, we will build and automate a data pipeline that ingests and transforms CSV files on schedule. We will learn how to stage a file using a Snowflake internal stage. We will execute the Snowflake `COPY` command to load data from the internal Snowflake stage to a staging table. Next, we will merge data from the staging table into the target table. Then we will transform the data into an appropriate format for the downstream consumers using SQL statements. Finally, we will create a task that automates the pipeline execution process every day. A data pipeline using these steps is shown in figure 2.1.

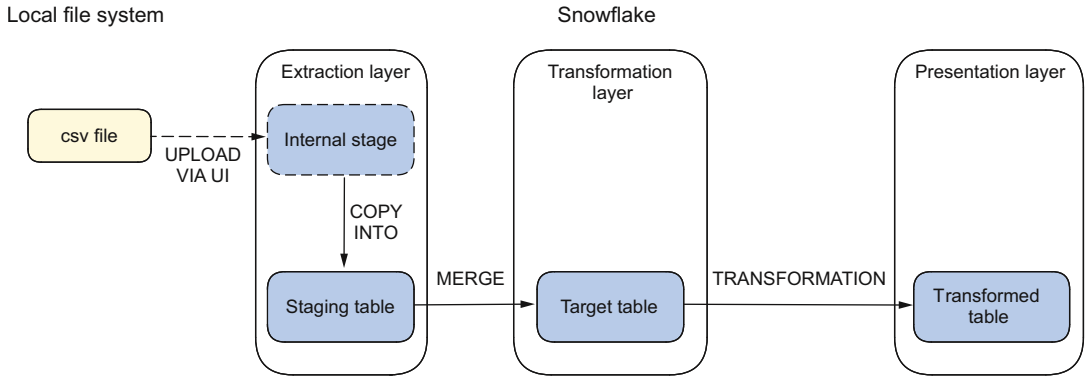


Figure 2.1 Data pipeline that takes CSV files from a Snowflake internal stage, loads the data into a staging table, merges the data from the staging table into the target table, and transforms the data for downstream consumption

Throughout this chapter, we will work with an example to illustrate building the pipeline. We will consider a fictional bakery that makes bread and pastries. The bakery delivers these baked goods to small businesses such as grocery stores, coffee shops, and restaurants in the neighborhood. Since the bakery has no online ordering system, customers order baked goods via email. A bakery employee reads the emails and collects all customer orders into a CSV file saved on the local file system.

The bakery manager summarizes the orders in the CSV file at the end of each working day to determine how much of each baked good the bakery must produce in the coming days. This information is required so the bakery knows how much raw materials to purchase and how many employees are needed to work on a given day to meet the demand.

The bakery wants to automate the data management process by ingesting the data from the CSV file into Snowflake at the end of each day and performing the summarization steps. This chapter walks you through the process of building a data pipeline that supports the bakery's scenario.

Referring to the Snowflake documentation

Throughout this book, we work with examples and execute Snowflake commands as needed to illustrate examples that describe the various data engineering tasks. We don't describe each Snowflake command in detail or provide all possible options and variations of each command. We highlight the syntax and options relevant to the example and give information about executing the most frequently used command options and parameters. Details about all commands are available in the Snowflake documentation at https://docs.snowflake.com/.

Links to specific sections in the documentation are provided where needed.

2.1 Setting up your Snowflake account

To follow along with the example in this chapter, as well as examples in subsequent chapters, you will need access to a Snowflake account. If you already have access to a Snowflake account where you can work through the examples, then feel free to use it. Ensure you have sufficient privileges to create a database and add additional objects to it.

If you don't already have access to a Snowflake account, you can create a free trial Snowflake account at https://signup.snowflake.com/. Appendix A provides more information about creating a free trial Snowflake account and the options available.

NOTE All code and sample data files for this chapter are available in the accompanying GitHub repository in the Chapter_02 folder at https://mng.bz/dZn1.

The first step in the Snowflake account is to create a database, a schema, and a virtual warehouse that we will use to build the pipeline. For now, we will create objects using the `SYSADMIN` role, which is one of the built-in administrative roles in Snowflake. Usually, data engineers use custom roles in the Snowflake account, as we will discuss in later chapters. We will use the built-in roles for now since custom roles don't exist in the free trial Snowflake account.

TIP Even if you are using a Snowflake free trial account where you have the `ACCOUNTADMIN` role, remember that this is the most powerful role in Snowflake and should be used for administration tasks only. Never create objects using the `ACCOUNTADMIN` role; instead, choose a role with fewer privileges that will allow you to perform the required tasks.

To create the required database, schema, and virtual warehouse, we will open a new worksheet in the Snowflake web interface, also known as *Snowsight,* and execute the commands.

TIP If you are not familiar with the Snowsight user interface, please refer to the Snowflake documentation at https://mng.bz/r1zj.

We will use the SYSADMIN role to create a database named BAKERY_DB, a schema named ORDERS, and an extra-small virtual warehouse named BAKERY_WH:

```
use role SYSADMIN;
create database BAKERY_DB;
create schema ORDERS;
create warehouse BAKERY_WH with warehouse_size = 'XSMALL';
```

Using the Snowsight user interface vs. executing commands

Many Snowflake objects, such as databases, schemas, virtual warehouses, and so on, can be created in Snowflake either by executing commands or by using the Snowsight user interface. Throughout this book, we will favor executing commands over the user interface. Data engineering pipelines are typically scheduled to run with minimal user intervention, which means that we must use commands that can be automated rather than clicking through the user interface.

2.2 Staging a CSV file

To start building the data pipeline required by the bakery, we must have a CSV file ready with data to ingest. The bakery has been collecting orders daily and saving them to CSV files. For example, the orders the bakery collected on July 7 are all stored in a CSV file named Orders_2023-07-07.csv. The first few rows of data in this file are shown in table 2.1.

Table 2.1 Orders collected by the bakery on a single day

Customer	Order date	Delivery date	Baked good type	Quantity
Coffee Pocket	2023-07-07	2023-07-10	Baguette	6
Coffee Pocket	2023-07-07	2023-07-10	Bagel	12
Coffee Pocket	2023-07-07	2023-07-10	English muffin	16
Coffee Pocket	2023-07-07	2023-07-10	Croissant	18
Lily's Coffee	2023-07-07	2023-07-10	Bagel	20
Lily's Coffee	2023-07-07	2023-07-10	White loaf	4
Lily's Coffee	2023-07-07	2023-07-10	Croissant	20
Crave Coffee	2023-07-07	2023-07-10	Croissant	50
Best Burgers	2023-07-07	2023-07-10	Hamburger bun	75
...	...	...	...	...

The CSV file is stored on the bakery's local file system. To ingest the data from this file into Snowflake, we will upload the file into a Snowflake *stage*. A stage is a Snowflake

object that points to the location of data files in cloud storage. Snowflake supports two types of stages:

- *External stage*—Data is stored in any of the supported cloud storage providers, including Amazon S3, Google Cloud Storage, or Microsoft Azure.
- *Internal stage*—Data is stored in the cloud storage provider that hosts the Snowflake account.

External stages require connectivity to the cloud storage provider and are discussed in more detail in the following chapter. We will use a Snowflake internal stage to keep our first data pipeline simple. We can stage data files in different types of internal stages:

- User stages that are allocated to each Snowflake user
- Table stages that are allocated to each table created in Snowflake
- Named internal stages

A Snowflake *named internal stage* is a database object created in a schema. It is more flexible than user stages or table stages because it can stage files managed by multiple users and loaded into multiple tables.

Let's create the Snowflake named stage to build the pipeline. In the same worksheet as previously, using the `BAKERY_DB` database and the `ORDERS` schema, we will execute the following command:

```
use database BAKERY_DB;
use schema ORDERS;
create stage ORDERS_STAGE;
```

We can view the contents of the stage by using the `LIST` command:

```
list @ORDERS_STAGE;
```

Because we just created the stage, it is empty, and the `LIST` command returns a message that says, "Query produced no results."

We can now upload the CSV file into the stage we just created. We will perform this step using the Snowsight user interface. To upload this file into the Snowflake internal stage, we will navigate to the main menu in Snowsight, expand the Data option, and click Databases. A second navigation pane appears where we can navigate from the `BAKERY_DB` database to the `ORDERS` schema, where we can expand the Stages folder and choose the `ORDERS_STAGE` internal stage. We will see a screen like in figure 2.2.

After we click `ORDERS_STAGE` in the navigation pane, we will see its properties in the main window. Here, we can click the +Files button to upload a file to the stage. In the window that appears, we can upload the Orders_2023-07-07.csv file.

Let's view the contents of the stage again with the same `LIST` command as earlier:

```
list @ORDERS_STAGE;
```

Now the stage should contain the CSV file we just uploaded. The output of the `LIST` command is shown in table 2.2.

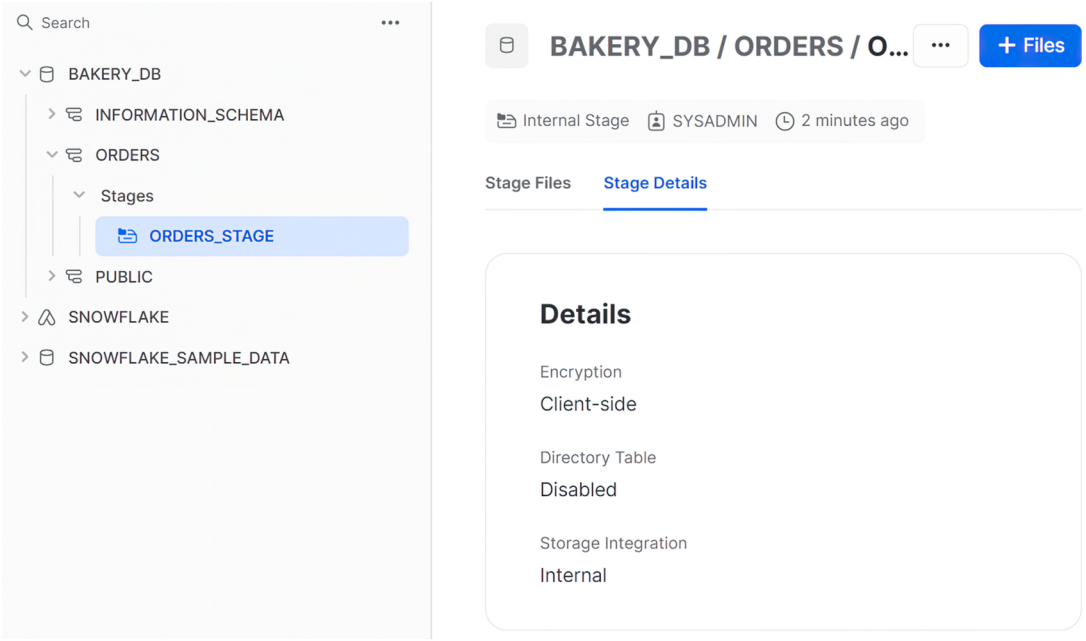


Figure 2.2 Stage details in the Snowsight user interface after expanding the Stages folder in the `ORDERS` schema in the `BAKERY_DB` database

Table 2.2 Output of the `LIST` command showing the CSV file that resides in the internal stage

Name	Size	MD5	Last_modified
`orders_stage/Orders_2023-07-07.csv`	3376	0fe929d77d8d60772d42dcf7482bb5bc	Fri, 7 Jul 2023 17:02:48

Loading files into Snowflake stages via the Snowflake CLI

An alternative method exists to upload files from a local file system into a Snowflake stage. We could have used the Snowflake command line interface to upload the file, but this would require installing and configuring the Snowflake CLI, adding complexity. This method could be challenging for a bakery employee, for whom the Snowsight user interface would be much easier to use.

2.3 Loading data from a staged file into a target table

Now that the CSV file is in the Snowflake internal stage, we can continue with the steps to load the data from the file into a staging table. But first, let's view the data to check that it is available and in the expected format. We will use an SQL command to view the data because we will need the same command later to load it.

The simplest way to view data in a stage is to use the `SELECT` command on the entire stage. One point to remember is that when selecting data from a stage, we don't use the `SELECT *` syntax to denote all columns, like in SQL queries, because Snowflake doesn't know the schema of the data files in the stage. Instead, we use the `$` notation to refer to columns in the file, for example, $1 for the first column, $2 for the second column, and so on.

With semistructured formats, such as Parquet or JSON, Snowflake considers each record as one column. However, when data is in the CSV format, the default field delimiter is the comma, and the default record delimiter is the new line character. Because we are using the same delimiters in our CSV file, Snowflake already recognizes the columns by default. To select everything from the internal stage, knowing that the uploaded CSV file contains five columns, we can execute the following command:

```
select $1, $2, $3, $4, $5 from @ORDERS_STAGE;
```

The output of this command shows the orders collected by the bakery, as in table 2.3 (only the first few rows are shown for illustration).

Table 2.3 The output of the command that selects data from the `ORDERS_STAGE` internal stage

$1	$2	$3	$4	$5
Customer	Order date	Delivery date	Baked good type	Quantity
Coffee Pocket	2023-07-07	2023-07-10	Baguette	6
Coffee Pocket	2023-07-07	2023-07-10	Bagel	12
Coffee Pocket	2023-07-07	2023-07-10	English muffin	16

As shown in table 2.3, Snowflake doesn't differentiate between the header column containing the column names and the data content. When loading data from the staged file into a table, we must indicate to Snowflake that we should exclude the header row with the column names. For this purpose, we will include a *file format* when we load the data.

Snowflake supports structured (CSV, tab-separated values, and similar) and semi-structured (JSON, Avro, ORC, Parquet, and XML) file formats. Together with the file format, we can provide options that specify the data type in the file and other properties that further describe the data format. We can define a file format as a stand-alone named database object in Snowflake, as explained in more detail in chapter 3. To keep it simple, we will provide the file format using individual format options specified in the command that loads, unloads, or views data in staged files.

Staged files include metadata columns that we can also include in a query. A useful metadata column is `metadata$filename`, which tells us the filename from which the data is ingested. Since we currently have only one file in the stage, this information

may be less valuable, but once we start adding more files to the stage, we will want to know which file is the source of each record.

Another useful metadata column is `metadata$file_row_number`, which contains the row number of the record in the file. This column is valuable when we want to ensure that records in the target table retain information about their order in the source data.

2.3.1 Loading data from a staged file into a staging table

We can proceed with loading data from the internal stage into a staging table in Snowflake. First, we must create the staging table. The table will contain the five columns that store data from the CSV file (customer, order date, delivery date, baked goods type, and quantity) and their respective data types (varchar, date, or number).

When ingesting data from files, it is a best practice to add columns that store additional information that helps us track when the data was ingested and where it originated. We will add a column to store the source filename where the data comes from and another to store the ingestion timestamp. We can use the following command to create a table named `ORDERS_STG` in the `BAKERY_DB` database and `ORDERS` schema that we created earlier:

```
use database BAKERY_DB;
use schema ORDERS;
create table ORDERS_STG (
  customer varchar,
  order_date date,
  delivery_date date,
  baked_good_type varchar,
  quantity number,
  source_file_name varchar,
  load_ts timestamp
);
```

Now that the table exists, we can use the Snowflake `COPY` command to copy data from the file in the internal stage into the staging table.

In subsequent chapters, various scenarios of using the `COPY` command are described in more detail. For now, let's briefly review this command. We use the `COPY INTO <table>` command to ingest data from cloud storage providers via external stages or from cloud storage contained in the Snowflake account via internal stages, as in the example in this chapter.

Although the `COPY` command is intended primarily for loading data, not transforming it, we can still apply a few simple transformations. These include selecting individual columns, reordering columns, converting data types, or truncating text strings.

To load data from the `ORDERS_STAGE` internal stage into the `ORDERS_STG` table, we execute the `COPY` command with additional options. The options that we will use in our example are `FILE_FORMAT`, which skips the header row; `ON_ERROR`, which instructs

Snowflake to abort the action in case of any errors; and the PURGE option, which removes the file from the internal stage after ingesting the data.

In the SELECT statement within the COPY command, we will include the five columns available in the staged file, the metadata$filename column, which gives us the name of the staged file, and the CURRENT_TIMESTAMP() function, which provides us with the current timestamp. The COPY command that loads data from the staged file into the Snowflake table is shown in the following listing.

Listing 2.1 COPY command to load data from the staged file into Snowflake

```
use database BAKERY_DB;
use schema ORDERS;
copy into ORDERS_STG
from (
  select $1, $2, $3, $4, $5, metadata$filename, current_timestamp()
  from @ORDERS_STAGE
)
file_format = (type = CSV, skip_header = 1)
on_error = abort_statement
purge = true;
```

After successfully executing the command, we can view the data that was populated into the ORDERS_STG table using a simple SQL statement:

```
select * from ORDERS_STG;
```

As shown in table 2.4, we should see 64 rows of order data from the CSV file (only the first few rows are shown for illustration).

Table 2.4 The output of the command that selects data in the ORDERS_STG staging table

Customer	Order date	Delivery date	Baked good type	Quantity	Source filename	Load timestamp
Coffee Pocket	2023-07-07	2023-07-10	Baguette	6	Orders_2023-07-07.csv	2023-07-07 09:43:47
Coffee Pocket	2023-07-07	2023-07-10	Bagel	12	Orders_2023-07-07.csv	2023-07-07 09:43:47
Coffee Pocket	2023-07-07	2023-07-10	English muffin	16	Orders_2023-07-07.csv	2023-07-07 09:43:47

Since we specified the PURGE option in the COPY statement, the CSV file in the Snowflake internal stage is removed after successful data ingestion.

We can recheck the contents of the stage using the LIST command:

```
list @ORDERS_STAGE;
```

The stage is once again empty because the PURGE option of the COPY command deleted the file after ingestion. The LIST command returns the "Query produced no results" message.

2.3.2 Merging data from the staging table into the target table

The staging table ORDERS_STG contains only the data available in the CSV files present in the internal stage when we executed the COPY command. We will add this data to the target table named CUSTOMER_ORDERS, which stores all historical data from all customer orders the bakery recorded.

The structure of the CUSTOMER_ORDERS table is the same as the staging table. We can create the table, still using the BAKERY_DB database and the ORDERS schema, by executing the following command:

```
use database BAKERY_DB;
use schema ORDERS;
create table CUSTOMER_ORDERS (
  customer varchar,
  order_date date,
  delivery_date date,
  baked_good_type varchar,
  quantity number,
  source_file_name varchar,
  load_ts timestamp
);
```

To load data from the staging table into the target table, we could use the INSERT SQL statement to add all data from the staging table to the target table. However, as data engineers, we must always be conscious of data integrity. Customers could change their orders to new quantities and send a second order for the same baked goods type for the same delivery date that supersedes the previous order.

To ensure only one record for a customer, baked goods type, and delivery date, we will construct a query that merges the data from the ORDERS_STG staging table into the CUSTOMER_ORDERS target table. A MERGE statement includes the following components:

- The MERGE INTO keywords, followed by the name of the target table that will receive the results of the statement
- The USING keyword, followed by the name of the source table where the data originates
- The ON clause, where we specify the primary key of the target table or the columns that ensure uniqueness; in our example, these columns are customer, baked good type, and delivery date
- The WHEN MATCHED THEN clause, executed when the same combination of the unique columns is present in both the source and the target tables; in this case, the values from the source table overwrite the values in the target table

- The `WHEN NOT MATCHED THEN` clause, executed when a combination of the unique columns doesn't exist in the target table; in this case, the record from the source table is inserted into the target table

The query that we will execute is shown in the following listing.

Listing 2.2 Merging orders from the staging table into the target table

```
-- the target table
merge into CUSTOMER_ORDERS tgt
-- the source table
using ORDERS_STG as src
-- the columns that ensure uniqueness
on src.customer = tgt.customer
  and src.delivery_date = tgt.delivery_date
  and src.baked_good_type = tgt.baked_good_type
-- update the target table with the values from the source table
when matched then
  update set tgt.quantity = src.quantity,
    tgt.source_file_name = src.source_file_name,
    tgt.load_ts = current_timestamp()
-- insert new values from the source table into the target table
when not matched then
  insert (customer, order_date, delivery_date, baked_good_type,
    quantity, source_file_name, load_ts)
  values(src.customer, src.order_date, src.delivery_date,
    src.baked_good_type, src.quantity, src.source_file_name,
    current_timestamp());
```

By merging the data from the staging table into the target table, we will always have no more than one row per customer, baked goods type, and delivery date with the most recent value for the quantity in the target table.

> **NOTE** Unlike in other popular relational databases, you don't have to define a primary key when creating a table in Snowflake.

We can view the data that was populated into the `CUSTOMER_ORDERS` target table using a simple SQL statement:

```
select * from CUSTOMER_ORDERS order by delivery_date desc;
```

We should see the 64 rows of order data, just like in table 2.4.

> **NOTE** Using the `MERGE` statement to add data from the staging table into the target table is one of many ways to add new data to a table that stores historical data. Other approaches might also be used, such as an append-only strategy or building SCD (slowly changing dimension) type 2 historical changes. These approaches are discussed in more detail in subsequent chapters.

2.4 Transforming data with SQL commands

The CUSTOMER_ORDERS target table contains all customer orders from the past and future orders that customers have already placed. Users can query this table to examine ordering trends in the past and create reports. The requirement from the bakery manager is that for better planning, the bakery needs to know how many of each type of baked good is ordered per day in the upcoming days.

Because this information is required by the bakery regularly as scheduled, we will construct a summarized table named SUMMARY_ORDERS with three columns: delivery date, baked goods type, and the total quantity. To create the table, still using the BAKERY_DB database and the ORDERS schema, we can execute the following command:

```
use database BAKERY_DB;
use schema ORDERS;
create table SUMMARY_ORDERS(
  delivery_date date,
  baked_good_type varchar,
  total_quantity number
);
```

We will insert summarized data from the CUSTOMER_ORDERS table that we populated in the previous step. To simplify it, we will truncate the SUMMARY_ORDERS table each time and insert the complete summarized data. We can do this because the bakery doesn't store vast amounts of data, and we don't have to consider the compute cost of replacing all data each time or the time it would take to populate the data. In subsequent chapters, we will learn about different ways to transform data, such as materialized views or dynamic tables. But for now, let's replace the data each time.

The query that summarizes the CUSTOMER_ORDERS data is

```
select delivery_date, baked_good_type, sum(quantity) as total_quantity
  from CUSTOMER_ORDERS
  group by all;
```

We are using the `group by all` Snowflake syntax in this query. Instead of specifying all columns to be included in the `group by` clause, Snowflake allows us to use the keyword `all`, indicating all columns that are not part of the aggregation. After executing the previous query, we should see results like in table 2.5 (only the first few rows are shown for illustration).

Table 2.5 The query output that summarizes the CUSTOMER_ORDERS data

Delivery date	Baked good type	Total quantity
2023-07-10	Baguette	30
2023-07-10	Bagel	70
2023-07-10	English muffin	52

Table 2.5 The query output that summarizes the CUSTOMER_ORDERS data *(continued)*

Delivery date	Baked good type	Total quantity
2023-07-10	Croissant	88
2023-07-10	White loaf	44
2023-07-10	Hamburger bun	137
2023-07-10	Whole wheat loaf	25

To prepare summarized data that the bakery manager requires for future planning, we can truncate the target table and insert the summarized data using the previous query. First, we truncate the summary table using the following command:

```
truncate table SUMMARY_ORDERS;
```

Then we insert the summarized data into the summary table using the query in the following listing.

Listing 2.3 Inserting summarized data into the summary table

```
insert into SUMMARY_ORDERS(delivery_date, baked_good_type, total_quantity)
  select delivery_date, baked_good_type, sum(quantity) as total_quantity
  from CUSTOMER_ORDERS
  group by all;
```

The bakery manager can then view the summarized data by selecting from this table using the following command:

```
select * from SUMMARY_ORDERS;
```

The output of this command is the same as in table 2.5 (only the first few rows are shown for illustration).

2.5 Automating the process with tasks

Because the bakery receives new and updated orders from customers regularly every day, they want to automate the data ingestion and transformation process as much as possible.

They will still have to manually collect the order data by reading emails and storing it in CSV files since they don't have any other system available. They must also manually upload the CSV files to the internal Snowflake stage. But from there on, the data engineers can automate the process by combining the previous steps and scheduling them to run periodically, such as every evening. They can create Snowflake *tasks* for this purpose.

A Snowflake task can execute a single SQL statement, a stored procedure, or a set of SQL statements using Snowflake scripting procedural logic. Tasks are often combined

with *streams* that detect changes in source data. In this chapter, we will use tasks without streams in the data pipeline we are building. Streams are explained in more detail in chapter 12.

Tasks require compute resources when they execute. Compute resources can be provided by a virtual warehouse or by Snowflake-managed compute resources. In the data pipeline we are building in this chapter, we will use the `BAKERY_WH` virtual warehouse we created earlier.

We will create a task named `PROCESS_ORDERS` that executes all the previous steps in sequence every day based on a schedule that we provide. The steps that the task will execute are

1. Truncate the staging table.
2. Load data from the internal stage into the staging table using the COPY command.
3. Merge data from the staging table into the target table.
4. Truncate the summary table.
5. Insert summarized data into the summary table.

Initially, for testing, we will schedule the task to execute every 10 minutes to monitor several executions of the task to check that it is working correctly. Later, once we are happy that everything is fine, we will schedule it to run every evening at 11 p.m.

The following code creates the `PROCESS_ORDERS` task (some of the SQL statements are abbreviated for clarity, using code from previous listings):

```
use database BAKERY_DB;
use schema ORDERS;
create task PROCESS_ORDERS
  warehouse = BAKERY_WH
  schedule = '10 M'
as
begin
  truncate table ORDERS_STG;
  copy into ORDERS_STG...
  merge into CUSTOMER_ORDERS...
  truncate table SUMMARY_ORDERS;
  insert into SUMMARY_ORDERS...
end;
```

Code from listing 2.1

Code from listing 2.3

Code from listing 2.4

When we create a task in Snowflake, it is suspended by default. For the task to execute, we must resume it, allowing it to run on schedule. Before we let the task run on schedule, we can test it by manually executing it a single time.

However, before executing the task, we must grant the `EXECUTE TASK` privilege to the role that will be executing the task. While Snowflake allows all roles to create tasks without additional privileges, it doesn't enable tasks to be executed by default. Since we are using the SYSADMIN role in this chapter, we must grant the `EXECUTE TASK` privilege to this role (typically, data engineers use custom roles to create and

execute data pipelines, and we would grant the EXECUTE TASK privilege to the custom role). We can grant the EXECUTE TASK privilege to the SYSADMIN role with the following commands:

```
use role accountadmin;
grant execute task on account to role sysadmin;
use role sysadmin;
```

We must use the ACCOUNTADMIN role to perform the grant. Whenever we use the ACCOUNTADMIN role to grant privileges or perform any other tasks requiring this role, we must always remember to switch back to the role we use for development when we no longer need it. That's why we are switching back to the SYSADMIN role.

After granting the EXECUTE TASK privilege, we can execute the task a single time using the following command:

```
execute task PROCESS_ORDERS;
```

After executing the task, we want to verify that it finished successfully. We can retrieve information about query execution by calling the TASK_HISTORY() table function in INFORMATION_SCHEMA.

Snowflake information schema

The Snowflake information schema contains views with metadata about all objects created in a Snowflake account and table functions containing historical and usage data related to the account. The information schema exists in every database in a schema named INFORMATION_SCHEMA. For more information, refer to the Snowflake documentation at https://mng.bz/V2Jx.

We can execute the command in the following listing to view previous and scheduled task executions.

Listing 2.4 Viewing previous and scheduled task executions

```
select *
  from table(information_schema.task_history())
  order by scheduled_time desc;
```

The output of this command shows the task's execution details, including the code that was executed, the state (SUCCEEDED or FAILED), the error message in case the task failed, the return value, the timestamps of when it started and completed the execution, and more. A sample output of this command is shown in table 2.6.

Table 2.6 Output columns and their values resulting from executing the `TASK_HISTORY()` table function

Column name	Value
QUERY_ID	01b0c4f9-0102-24a5-0001-6d92000bd6b2
NAME	PROCESS_ORDERS
DATABASE_NAME	BAKERY_DB
SCHEMA_NAME	ORDERS
QUERY_TEXT	begin truncate table ORDERS_STG; copy into ORDERS_STG...
STATE	SUCCEEDED
ERROR_CODE	
ERROR_MESSAGE	
SCHEDULED_TIME	2023-07-07T11:37:05.747-08:00
QUERY_START_TIME	2023-07-07T11:37:07.371-08:00
NEXT_SCHEDULED_TIME	2023-07-07T11:47:05.747-08:00
COMPLETED_TIME	2023-07-07T11:37:10.419-08:00

Once we are confident that the task is working correctly, we can allow it to run on the schedule we initially set for testing every 10 minutes so that we can monitor it for a few executions. To enable the task to run on the defined schedule, we resume it using the following command:

```
alter task PROCESS_ORDERS resume;
```

We can use additional CSV files found in the GitHub repository (Orders_2023-07-08.csv and Orders_2023-07-09.csv) to verify that data is loaded correctly each time the task executes. We can manually upload each file to the internal Snowflake stage between task executions and then check that data is loading from these files into the staging, target, and summary tables. We can also monitor the success of task executions and the next scheduled executions by querying the `TASK_HISTORY()` table function like in listing 2.4.

Finally, when we are satisfied that the task correctly ingests data from the staged files into Snowflake tables, we can change the schedule to execute every evening at 11 p.m. To do that, we will schedule the task using CRON syntax.

Before we can make changes to a task, we must suspend it. After we make the change, we must resume the task again. The CRON parameters for scheduling a task to execute at 11 p.m. are 0 (the minute), 23 (the hour in 24-hour time format), * (day of month), * (month), * (day of week). We must also provide a time zone together with the time information so that it is unambiguous when the task must be executed. In our example, we are providing the time zone as UTC.

TIP For more information about scheduling tasks with the CRON syntax, refer to the Snowflake documentation at https://mng.bz/x686.

We can change the schedule of the task by executing the following sequence of commands:

```
alter task PROCESS_ORDERS suspend;
alter task PROCESS_ORDERS
set schedule = 'USING CRON 0 23 * * * UTC';
alter task PROCESS_ORDERS resume;
```

We can again query the `TASK_HISTORY()` table function from listing 2.4 to confirm that the task is scheduled at the specified time. Then we can review the task history again the next day to verify that it was executed successfully.

TIP When we are done testing the task, we can suspend it so that it doesn't needlessly continue to execute and consume resources.

To suspend the task, we can execute the following command:

```
alter task PROCESS_ORDERS suspend;
```

In this chapter, we created our first data pipeline. We kept it simple as a starting point. In the following chapters, we will build on this pipeline to add more functionality and introduce different ways to achieve similar results.

Summary

- We set out to build and automate a data pipeline that ingests data from CSV files, transforms the data, and repeats the task regularly on schedule.
- To ingest data from a CSV file into Snowflake, we upload the file to an internal stage. A Snowflake stage refers to the location of data files in cloud storage.
- We can view the contents of a staged data file by executing a `SQL SELECT` command on the stage. Instead of the usual `SELECT *` to denote all columns, we use the $ notation to refer to columns in the stage.
- We execute the Snowflake `COPY` command to ingest data from the staged CSV file into a Snowflake staging table. To interpret the file structure correctly, we can specify a file format, such as ignoring the header row.
- We can use the `SQL MERGE` command to add new data from the staging table to the target table in Snowflake. However, we must be conscious of data integrity constraints, such as uniqueness.
- We can perform data transformations using SQL commands, for example, to summarize data for reporting and store the summarized data in a new table that will be used by the downstream consumers.
- A Snowflake task can execute a sequence of SQL statements on schedule. The task can be scheduled to execute according to predefined time intervals or at a given time.

Part 2

Ingesting, transforming, and storing data

Now that you have completed your first data engineering pipeline, you're ready to explore the more advanced aspects of Snowflake data engineering.

In chapter 3, you'll learn how to ingest data from a cloud storage provider and create external stages in Snowflake. We will explain and compare different approaches to ingesting files and offer tips on preparing data files in cloud storage for efficient ingestion.

In chapter 4, you'll ingest semistructured data in JSON format and flatten it into a relational structure. You will add exception handling and logging to the data pipeline to ensure its resilience against unintended errors.

In chapter 5, you'll build a new data pipeline that continuously ingests data from files as soon as they appear in the external cloud storage with minimum delay. We will introduce Snowflake features like Snowpipe for continuous data ingestion and dynamic tables for continuous data transformation.

Chapter 6 covers Snowpark, which consists of libraries and code execution environments that allow Python and other programming languages to run natively in Snowflake.

In chapter 7, you'll immerse yourself in generative AI and large language models (LLMs). You will learn how to call external API endpoints from Snowflake and use Snowflake's own Cortex LLM functions to enhance your data pipelines.

In chapter 8, we will utilize Snowflake's query profile tool to understand the mechanics of query execution and identify opportunities for optimizing query performance when dealing with large data volumes.

Chapter 9 will explore what contributes to Snowflake's cost and how to monitor credit consumption. We will explain Snowflake virtual warehouses and how they affect query performance.

Finally, in chapter 10, you will learn about Snowflake role-based access control and data governance features, such as row access policies and masking policies, that limit data access to authorized users.

3 Best practices for data staging

This chapter covers

- Creating external stages to ingest data files from cloud storage
- Viewing stage metadata with directory tables
- Preparing data files for efficient ingestion
- Querying data in external stages with external tables
- Using materialized views to improve query performance

In this chapter, we will build upon the first data pipeline we created in chapter 2. We will add more functionality in the data ingestion step by accessing files from cloud storage rather than the local file system. We will authenticate to the cloud storage provider and create external stages.

Because requirements for data pipelines vary, there is no one-size-fits-all approach to building data pipelines. We will demonstrate and compare different options for ingesting data from cloud storage, including using the `COPY` command on the external stage, creating external tables, and creating materialized views to improve query performance. We will also review how to prepare data files in cloud storage for efficient ingestion.

Our first step will be to create an external stage exposing files from cloud storage to Snowflake. In this chapter, we will continue to ingest data from CSV files. In addition to CSV, Snowflake can ingest many structured and semistructured file formats, such as JSON, Parquet, or XML, which we will discuss in the next chapter.

The syntax and parameters for creating external stages may differ depending on the cloud storage provider (Amazon S3, Google Cloud Storage, or Microsoft Azure Blob Storage) that hosts the files we will ingest. Throughout this chapter, we will use Microsoft Azure as the platform to illustrate examples, but you can follow along with a different cloud storage provider. We will provide links to the Snowflake documentation, where you will find detailed syntax for working with each supported cloud storage provider.

> **TIP** If you already have access to one of the supported cloud storage providers, you can use it to follow along with the examples in this chapter, but ensure you have sufficient privileges to upload files. Otherwise, you can create a free trial account with any supported cloud storage provider.

In this chapter, we will continue using the fictional bakery introduced in chapter 2 as an example to build the pipelines. To briefly recap, the bakery makes bread and pastries. The bakery delivers these baked goods to small businesses, such as grocery stores, coffee shops, and restaurants in the neighborhood. Since the bakery has no online ordering system, the customers place their orders for baked goods via email, and the bakery stores the orders in CSV files on their local file system.

One of the bakery's customers, a nearby restaurant, recently moved its information infrastructure to the cloud. The restaurant can now provide daily order information in CSV files stored in a dedicated container within its cloud storage platform. The bakery will build a data pipeline that ingests data from the restaurant's order files in the cloud storage container and stores it in the Snowflake database.

> **NOTE** All code and sample data files for this chapter are available in the accompanying GitHub repository in the Chapter_03 folder at https://mng.bz/AaJQ.

To build the pipeline that ingests the restaurant's files from cloud storage, the bakery data engineers must gain information about the location and the credentials they will use to access the files. Then they will create an external stage to make the data files available to Snowflake. There are generally two approaches to creating external stages:

- Create an external stage by using a Snowflake *storage integration* object. A storage integration object authenticates Snowflake to the cloud storage provider and is used to create external stages. It encapsulates the credentials to access cloud storage, so developers don't have to provide them. It also gives cloud storage administrators control over the access.
- Create an external stage by providing credentials that authenticate Snowflake to the cloud storage provider when creating the external stage. Data engineers

can use this method to quickly load some data files from cloud storage into Snowflake—for example, for testing or performing a proof of concept without the overhead of creating a storage integration object.

After creating the external stage using either of the two options, the file ingestion process is like ingesting files from internal stages, as explained in chapter 2. The `COPY` command copies data from the external stage to a staging table. After that, data can be loaded from the staging table to the target table, like in the pipeline in chapter 2. Figure 3.1 illustrates the steps required to ingest data from cloud storage to Snowflake.

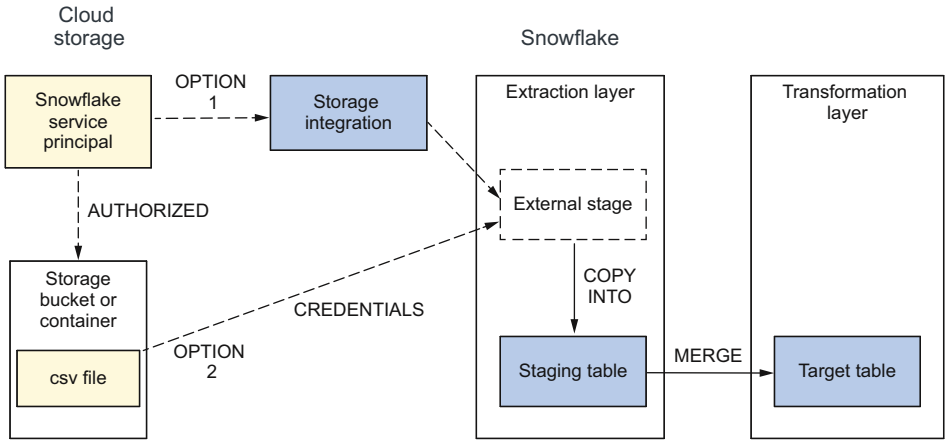


Figure 3.1 A data pipeline that takes CSV files from a cloud storage location makes the files available to Snowflake in an external stage, either by using a storage integration object or by providing credentials, loads the data into a staging table, and inserts or merges the data from the staging table into the target table.

3.1 Creating external stages

An *external stage* is a Snowflake object that exists in a schema. This object stores the location of the files in cloud storage, the parameters used to access the cloud storage account, and additional parameters, such as the options that describe the format of staged files.

Snowflake can ingest data from any supported cloud storage provider (Amazon S3, Google Cloud Storage, or Microsoft Azure Blob Storage), regardless of the cloud platform that hosts the Snowflake account.

For the examples in this chapter, we will assume that the restaurant providing order information to the bakery already has a blob storage container set up in its Microsoft Azure account.

TIP If you prefer using your AWS account, see chapter 4, which describes creating a storage integration object and an external stage in Amazon S3.

Creating a storage integration in Amazon S3 or Google Cloud Storage

You can find details about setting up the storage integration in Amazon S3 and Google Cloud Storage at https://mng.bz/ZVJ9 (Amazon S3) and https://mng.bz/RNJa (Google Cloud Storage).

If you are following along using your Microsoft Azure account, you must create resources in your account by performing the following steps:

1. Create a Resource group, or use an existing Resource group if you already have one and want to use it for the exercises. You can name the Resource group however you wish.
2. Create a Storage account in the Resource group.

IMPORTANT When choosing the name for your Storage account, remember that storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only. Additionally, your storage account name must be unique within Azure. The exercises in this chapter use `bakeryorders001` as the storage account name. If you want to name your storage account similarly but the name is unavailable, change the suffix from `001` to another combination of digits, and modify any code accordingly.

3. Create a container. A container is like a directory in a file system in that it organizes a set of files. You can name the container however you wish, but if you want to name it the same as in the exercises in this chapter, name it `orderfiles`.

Once you have created the resources, you can navigate to the `orderfiles` container and upload one of the sample files, Orders_2023-08-04.csv, from the GitHub repository in the Chapter_03 folder to the container. We will create an external stage using a storage integration to ingest data from CSV files in the storage container.

3.1.1 Configuring a storage integration

A *storage integration* is an object that authenticates Snowflake to the cloud storage provider. It enables Snowflake to read data from and optionally write data to an external stage. Depending on the cloud storage provider, a storage integration object holds access information in different parameters.

In Microsoft Azure, which we will use as an example in this chapter, Snowflake must be registered as an application in the Azure account. Only a user with administrative privileges on Azure can grant the required privileges to the Snowflake application. A storage integration must also specify a storage container that limits the locations where you can create external stages.

We will use the `CREATE STORAGE INTEGRATION` command to create a storage integration with the restaurant's cloud storage location. A Snowflake user with the `ACCOUNTADMIN` role or a different role with the `CREATE INTEGRATION` privilege can create a storage integration.

Before we can execute the command to create a storage integration, we must gather the following information provided by the restaurant that owns the storage account:

- The Azure Tenant ID
- The name of the storage account
- The name of the storage container

In our example, the Azure Tenant ID is `1234abcd-xxx-56efgh78` (this is a fictional Tenant ID for illustrative purposes), the storage account is `bakeryorders001`, and the container is `orderfiles`.

> **TIP** An Azure Tenant ID is a unique identifier assigned to each organization that uses Microsoft services. You can find your Tenant ID by clicking Settings, then Directories + Subscriptions. You will see a list of directories. Find the Default Directory, and look for the value of the Directory ID. This is your Tenant ID.

Using this information (the Tenant ID, storage account, and container), we can create the `BISTRO_INTEGRATION` storage integration using the following command:

```
use role ACCOUNTADMIN;
create storage integration BISTRO_INTEGRATION
  type = external_stage
  storage_provider = 'AZURE'
  enabled = true
  azure_tenant_id = '1234abcd-xxx-56efgh78'
  storage_allowed_locations =
    ('azure://bakeryorders001.blob.core.windows.net/orderfiles/');
```

After executing the command, a storage integration object is created in the Snowflake account. We can view the details of the storage integration by executing the `DESCRIBE STORAGE INTEGRATION` command as follows:

```
describe storage integration BISTRO_INTEGRATION;
```

Table 3.1 shows a sample output of this command (your output will contain your Azure Tenant ID, consent URL, and application name).

Table 3.1 Sample output of the `DESCRIBE STORAGE INTEGRATION` command

Property	Property type	Property value
ENABLED	Boolean	
STORAGE_PROVIDER	String	AZURE
STORAGE_ALLOWED_LOCATIONS	List	azure://bakeryorders001.blob.core.windows.net/orderfiles/

Table 3.1 Sample output of the `DESCRIBE STORAGE INTEGRATION` command *(continued)*

Property	Property type	Property value
STORAGE_BLOCKED_LOCATIONS	List	
AZURE_TENANT_ID	String	1234abcd-xxx-56efgh78
AZURE_CONSENT_URL	String	https://login.microsoftonline.com/1234abcd-xxx-56efgh78z/oauth2/authorize?client_id=1234abcd-xxx-56efgh78z &response_type=code
AZURE_MULTI_TENANT_APP_NAME	String	12abcsnowflakepacint_1234567890

This command returns the properties of the storage integration. We must take note of the values of the following properties:

- `AZURE_CONSENT_URL`—The value of this parameter might look like `https://login.microsoftonline.com/1234abcd-xxx-56efgh78z/oauth2/authorize?client_id=1234abcd-xxx-56efgh78z &response_type=code` (this is a fictional URL for illustrative purposes).
- `AZURE_MULTI_TENANT_APP_NAME`—The value of this parameter might look like `12abcsnowflakepacint_1234567890` (this is a fictional application name for illustrative purposes).

First, we copy the value of the `AZURE_CONSENT_URL` property and paste it as a URL into a new web browser window. An Azure administrator must click the Accept button in the form that appears. Clicking this button will grant an access token to the Snowflake application (also referred to as the service principal in Azure) on the storage container. However, more is needed to allow Snowflake to access the files in the storage container. Permissions on the storage container must also be granted to the Snowflake service principal by the Azure administrator.

To do this, an Azure administrator must log in to the Azure portal and grant a role assignment to the Snowflake service principal. Detailed instructions are available in the Snowflake documentation at https://mng.bz/2gNw.

The Azure administrator can grant either read access using the `Storage Blob Data Reader` role or read and write access using the `Storage Blob Data Contributor` role to the Snowflake service principal.

> **TIP** If you are following along, note that it may take an hour or longer for the Snowflake service principal to appear in Azure. Be patient if it doesn't appear immediately after you accept the consent URL.

In our example, the Azure administrator performs the Azure administration steps, not the data engineer. If you are following along using your Azure account, you will also

be the Azure administrator. Once the storage integration is created and access granted, we can use it to create an external stage.

3.1.2 Creating an external stage using a storage integration

To create an external stage that uses the storage integration, usage of the storage integration object must be granted to the role that will be creating the external stage. We will continue to use the BAKERY_DB database that we created in chapter 2 and work using the SYSADMIN role. Because the SYSADMIN role will require the use of the storage integration, the ACCOUNTADMIN role that created the BISTRO_INTEGRATION storage integration must grant usage on this object to the SYSADMIN role. This can be accomplished by executing the following command:

```
grant usage on integration BISTRO_INTEGRATION to role SYSADMIN;
```

We will now switch to the SYSADMIN role. To keep the objects that are related to externally sourced data separate from the objects that we created in chapter 2, we will create a new schema named EXTERNAL_ORDERS in the BAKERY_DB database using the following commands:

```
use role SYSADMIN;
create warehouse if not exists BAKERY_WH with warehouse_size = 'XSMALL';
create database if not exists BAKERY_DB;
use database BAKERY_DB;
create schema EXTERNAL_ORDERS;
use schema EXTERNAL_ORDERS;
```

Then we will create an external stage named BISTRO_STAGE using the BISTRO_INTEGRATION storage integration. We will provide the location of the files in the storage container we received from the restaurant when we created the storage integration (the name of the storage account is bakeryorders001, and the container is orderfiles). We can execute the command shown in the following listing to create the external stage.

Listing 3.1 Creating an external stage using a storage integration

```
create stage BISTRO_STAGE
  storage_integration = BISTRO_INTEGRATION
  url = 'azure://bakeryorders001.blob.core.windows.net/orderfiles';
```

To view the contents of the external stage, we can execute the following command:

```
list @BISTRO_STAGE;
```

The output from this command will show any files the restaurant has already uploaded into the blob storage container.

3.1.3 Creating an external stage using credentials

Snowflake recommends creating external stages using storage integration objects. These objects encapsulate the credentials to access cloud storage, so data engineers don't have to supply them when creating stages or loading data. Another benefit of maintaining storage integrations between Snowflake and cloud providers is that cloud providers have better control over Snowflake access. They can block access to the Snowflake application from the cloud storage platform for any reason, such as a suspected security breach or no longer required access.

However, creating storage integration objects requires the cooperation of a cloud storage provider administrator, who may not always be available. Sometimes data engineers want to load data files from cloud storage into Snowflake quickly—for example, when testing or performing a proof of concept—without the overhead of creating a storage integration object. Amazon S3 and Microsoft Azure (but not Google Cloud Platform) support creating external stages by providing credentials without a storage integration object.

When working with Azure storage containers, data engineers can generate a *shared access signature* (SAS) token to access objects in the storage account. For example, the restaurant providing order information in CSV files in an Azure storage container can send an SAS token to access the container.

> **TIP** If you are following along using your Microsoft Azure account, you can generate an SAS token according to instructions in the Snowflake documentation at https://mng.bz/1aER.

Once we have the value of the SAS token—for example, `?sv=2023-...%3D` (this is a fictional SAS token for illustrative purposes)—we can create an external stage named `BISTRO_SAS_STAGE` to access the files in the same storage account and container as previously (the name of the storage account is `bakeryorders001`, and the name of the container is `orderfiles`) using the following command:

```
create stage BISTRO_SAS_STAGE
  URL = 'azure://bakeryorders001.blob.core.windows.net/orderfiles'
  CREDENTIALS = (AZURE_SAS_TOKEN = '?sv=2023-...%3D');
```

We can view the contents of the external stage by executing the following command:

```
list @BISTRO_SAS_STAGE;
```

The files uploaded into the blob storage container are visible.

When using a SAS token, remember that it is valid until the expiration date, which is defined when you create it, so you can't use it indefinitely. Compared with a storage integration, where an administrator controls access to the external storage, an administrator can't supervise who has generated a SAS token and thus doesn't know who has access.

From now on, we will use the BISTRO_STAGE external stage created using the storage integration, so we have only one external stage to work with. This choice is arbitrary; we could have used the BISTRO_SAS_STAGE external stage instead because it would behave the same.

3.1.4 Loading data from staged files into a staging table

We use the COPY command to load data from files in external stages into a staging table, just like loading data from files in internal named stages, as explained in chapter 2.

Let's review one CSV file the restaurant provided in the blob storage container. For example, the orders the restaurant placed on August 4 are all saved in a CSV file named Orders_2023-08-04.csv. The contents of this file are shown in table 3.2.

Table 3.2 Orders placed by the restaurant on a single day

Customer	Order date	Delivery date	Baked good type	Quantity
New Bistro	2023-08-04	2023-08-07	Baguette	15
New Bistro	2023-08-04	2023-08-07	Whole wheat loaf	12
New Bistro	2023-08-04	2023-08-07	White loaf	10
New Bistro	2023-08-04	2023-08-07	Croissant	30
New Bistro	2023-08-04	2023-08-08	Baguette	12
New Bistro	2023-08-04	2023-08-08	Whole wheat loaf	12
New Bistro	2023-08-04	2023-08-08	White loaf	10
New Bistro	2023-08-04	2023-08-08	Croissant	20

We will load data from the restaurant's CSV files into a staging table named ORDERS_BISTRO_STG. This table has the same structure as the ORDERS_STG staging table we used in chapter 2. The columns in this table include

- All columns from the CSV file, including customer, order date, delivery date, baked goods type, and quantity.
- The name of the CSV file is derived from the metadata column metadata$filename—like in internal stages, files in external stages include metadata columns that can be queried.
- The timestamp indicating when data was ingested.

We will create this table in the EXTERNAL_ORDERS schema in the BAKERY_DB database with the following command:

```
use database BAKERY_DB;
use schema EXTERNAL_ORDERS;
create table ORDERS_BISTRO_STG (
  customer varchar,
  order_date date,
```

```
  delivery_date date,
  baked_good_type varchar,
  quantity number,
  source_file_name varchar,
  load_ts timestamp
);
```

To load data from CSV files in the `BISTRO_STAGE` external stage into the `ORDERS_BISTRO_STG` staging table, we can execute the command shown in the following listing.

Listing 3.2 Loading data from an external stage into a staging table

```
copy into ORDERS_BISTRO_STG
from (
  select $1, $2, $3, $4, $5, metadata$filename, current_timestamp()
  from @BISTRO_STAGE
)
file_format = (type = CSV, skip_header = 1)
on_error = abort_statement;
```

After loading the data from the external stage to the staging table using the `COPY` command, we can view the data in the table by executing the `SQL SELECT` command:

```
select * from ORDERS_BISTRO_STG;
```

As shown in table 3.2, we should see eight rows of data matching the data in the CSV file.

In chapter 2, when we loaded data from CSV files in an internal stage into the target table, we used the `PURGE = TRUE` option in the `COPY` command. After loading, this option removes the files from the internal stage so we don't accidentally load data from the same file again on the next execution. Additionally, the `PURGE` command ensures that old files no longer needed don't accumulate over time.

We can also use the `PURGE` option with external stages, like in listing 3.2, but this option only works if the Snowflake service principal in the storage integration object was granted the `Storage Blob Data Contributor` role, which allows removing files.

Data engineers often have access to storage integration objects where the Snowflake service principal was granted the `Storage Blob Data Reader` role without the privileges to remove files after ingestion. If this is the case, then the `PURGE` option to remove files from the external stage will not remove any files but will also not report an error. Data engineers must check on their own to see if the files were removed from the stage when using the `PURGE` option.

Snowflake keeps track of files ingested from a stage into a table and does not ingest the same file again. However, when we allow files to accumulate in cloud storage, it will take increasingly longer to ingest data with each subsequent `COPY` execution because Snowflake will have to identify new files against a list of already loaded files each time, and the more files there are, the longer this will take. Therefore, it is still advisable that files that have already been processed are periodically removed from

object storage or moved to an archive location. The developers or administrators of the cloud storage platform usually perform this action.

TIP Snowflake data engineers can remove files from external stages with the `REMOVE` command if they have sufficient privileges on the object storage.

3.1.5 Avoiding duplication when loading data from staged files

When we execute a `COPY` command, Snowflake tracks metadata for each table that stores data from staged files. The metadata includes the filename, file size, entity tag (ETag; an identifier for each version of the file content), the number of rows in the file, the timestamp of the last load, and any errors that occurred during loading.

This ensures that subsequent or simultaneous executions of the `COPY` command don't load data from the same file again, preventing data duplication. Additionally, when loading data from individual files fails for any reason, Snowflake attempts to load data from these files again when the `COPY` command is issued the next time.

To see how this works in practice, let's execute the same `COPY` command as in listing 3.2 again. Because the CSV file in the external stage has already been loaded into the staging table, Snowflake will not load the same file again. The output of the `COPY` command when we execute it for the second time is

```
Copy executed with 0 files processed
```

Loading files with expired metadata

The file metadata for each table into which data is loaded from staged files is retained for 64 days. When Snowflake encounters a file modified more than 64 days ago or the data loaded from this file into a table more than 64 days ago, it no longer knows whether the data from this file was loaded successfully. In this case, it doesn't load the file again to avoid accidental duplication.

Data engineers can specify additional options, such as `FORCE` or `LOAD_UNCERTAIN_FILES` when executing the `COPY` command to control how Snowflake loads files with expired metadata.

When the `FORCE` option is `TRUE`, Snowflake loads all files, regardless of any load metadata. This is particularly useful when a data engineer wants to reload data from all files in the stage. When the `LOAD_UNCERTAIN_FILES` option is `TRUE`, Snowflake attempts to load files with expired load metadata.

To verify that the data was loaded successfully after executing the `COPY` command, we can examine the `LOAD_HISTORY` view in `INFORMATION_SCHEMA`. This view stores the history of data loaded into tables for the past 14 days. Alternatively, if we have sufficient privileges, we can examine the `LOAD_HISTORY` view in the `ACCOUNT_USAGE` schema in the `SNOWFLAKE` database. This view stores the history of data loaded into tables for the past 365 days.

Some interesting columns are the schema name, table name, filename, timestamp of the last load, number of loaded rows, number of parsed rows, and load status. Possible values for the load status are `LOADED`, `LOAD FAILED`, or `PARTIALLY LOADED`. Additional columns provide more information about error messages.

To view the load history for the `ORDERS_BISTRO_STG` table that we loaded with data from files in the `BISTRO_STAGE` external stage using the `COPY` command, we can execute the following query:

```
select *
from information_schema.load_history
where schema_name = 'EXTERNAL_ORDERS' and table_name = 'ORDERS_BISTRO_STG'
order by last_load_time desc;
```

The output of this command should show the list of files in the external stage from which data was loaded into the table. Table 3.3 shows a sample output with a selected set of columns.

Table 3.3 Sample output from the `LOAD_HISTORY` view

Filename	Table name	Last load time	Status
azure://bakeryorders001.blob.core.windows.net/orderfiles/ Orders_2023-08-04.csv	ORDERS_BISTRO_STG	2023-08-04T 10:15:20.052	LOADED
azure://bakeryorders001.blob.core.windows.net/orderfiles/202308/Orders_2023-08-07.csv	ORDERS_BISTRO_STG	2023-08-04T 10:15:20.052	LOADED
azure://bakeryorders001.blob.core.windows.net/orderfiles/202308/Orders_2023-08-08.csv	ORDERS_BISTRO_STG	2023-08-04T 10:15:20.052	LOADED

3.1.6 Using a named file format

In the previous examples, we specified the file format as a parameter in the `COPY` command when loading data from a stage to a table. See, for example, the parameter `file_format = (type = CSV, skip_header = 1)` in listing 3.2. This parameter indicates that the files we are loading are in the CSV format, and they contain a one-line header that must be skipped when loading data from the file.

Instead of specifying the same file format each time we execute the `COPY` command, we can create a named file format and use it rather than writing it out every time we specify it. This is especially useful when we repeatedly load files that share the same format.

For example, we can create a named file format describing the format of the CSV files we are loading in our examples. To create a file format named `ORDERS_CSV_FORMAT` that specifies the format as a CSV file with a comma as the delimiter and one header line, we can execute the following command:

```
create file format ORDERS_CSV_FORMAT
  type = csv
  field_delimiter = ','
  skip_header = 1;
```

Then, when we execute the COPY command, like in listing 3.2, we can use this named file format as a parameter, as shown in the following listing.

Listing 3.3 Loading data using a named file format

```
copy into ORDERS_BISTRO_STG
from (
  select $1, $2, $3, $4, $5, metadata$filename, current_timestamp()
  from @BISTRO_STAGE
)
file_format = ORDERS_CSV_FORMAT
on_error = abort_statement;
```

Specifies the named file format

We can specify the file format of the staged files in more than one place in Snowflake. We can define the file format in any of the following scenarios:

- When we create the stage
- When we ingest data from the stage into the target table
- When we create the target table that stores the ingested data

We should only specify the file format in one of these places—not in all of them. If the file format is specified in more than one place, Snowflake considers the file format in the following order of precedence:

1. COPY command—If the file format is specified in the COPY command, it will be used, overriding any other file formats specified elsewhere.
2. Stage definition—If the file format is specified in the stage definition (but not in the COPY command), it will be used.
3. Table definition—If the file format is specified in the table definition (but not in the COPY command or in the stage definition), it will be used.

Instead of specifying the file format each time we execute the COPY command, we can specify the file format when we create the stage. In that case, we don't have to specify the file format in the COPY command. For example, we can create the external stage with the file format parameter, modifying the command from listing 3.1 as follows:

```
create or replace stage BISTRO_STAGE
  storage_integration = BISTRO_INTEGRATION
  url = 'azure://bakeryorders001.blob.core.windows.net/orderfiles'
  file_format = ORDERS_CSV_FORMAT;
```

Specifies the named file format

After we recreate the external stage with the ORDERS_CSV_FORMAT file format, we can omit the file_format option in the COPY command.

3.2 Viewing stage metadata with directory tables

When creating an internal or an external stage, we can add a directory table that stores metadata about the files in a stage. A directory table is an object associated with a stage but not a separate stand-alone Snowflake object. It can be added to a stage when it is created using the `CREATE STAGE` command or later with the `ALTER STAGE` command.

When files are added to or removed from an external stage location, the metadata in the directory table can be automatically refreshed by configuring the event notification service in the cloud storage provider. Event notification is discussed in more detail in the next chapter.

If event notification is not configured in the cloud storage provider, we can manually refresh the metadata for a directory table on an external stage. Internal stages currently only support the manual refresh of the file metadata.

We can add a directory table to the `BISTRO_STAGE` external stage that we created earlier with the following command:

```
alter stage BISTRO_STAGE
set directory = (enable = true);
```

After adding the directory table, we will refresh it manually using the following command:

```
alter stage BISTRO_STAGE refresh;
```

Finally, after the directory table is refreshed, we can query the list of files in the stage using the following command:

```
select *
from directory (@BISTRO_STAGE);
```

A sample output from this command with a selected set of columns is shown in table 3.4.

Table 3.4 Sample output from the query that lists files in the stage using the directory table

Relative path	Size	Last modified	ETag
202308/Orders_2023-08-07.csv	446	2023-11-18T 01:07:09	"0x8DBE815BE847912"
202308/Orders_2023-08-08.csv	446	2023-11-18T 01:07:09	"0x8DBE815BE847912"
Orders_2023-08-04.csv	448	2023-11-18T 01:06:31	"0x8DBE815A84D8EE0"

The output of this command is a list of files in the stage with columns such as the file's relative path to the stage location, the file size, the timestamp when it was last modified, the file URL, the file MD5 hash, and the ETag. This output is useful when a data

engineer needs more detailed information about files available in a stage than the LIST command.

3.3 Preparing data files for efficient ingestion

In the examples we use in this chapter, we work with relatively small amounts of data where performance and efficiency do not play a significant role. But imagine that instead of working in a small neighborhood bakery, you are working as a data engineer in a large industrial bakery that produces vast volumes of baked goods sold in major supermarkets. In this scenario, you might work with data pipelines that ingest large volumes of data, provided in many files, and the speed and efficiency of data ingestion becomes much more critical.

For such scenarios, Snowflake offers recommendations for efficient data ingestion that help with performance and cost optimization, such as file sizing and organization in cloud storage.

3.3.1 File sizing recommendations

The size and the number of clusters of Snowflake virtual warehouses used for loading data from files play an important role when planning for efficient loading. A virtual warehouse contains nodes consisting of resources, such as CPU, memory, and temporary storage, that are utilized by the COPY command to load data from files into Snowflake tables. A single node can process each file.

> **TIP** Snowflake recommends that files be sized somewhere between 100 MB and 250 MB compressed for optimized loading.

Files that exceed the recommended size should be split into smaller files that fit the sizing recommendation. On the other hand, many small files should be aggregated into larger files according to the file sizing recommendation.

When a large file is loaded, it will be processed on a single node, while the remaining nodes will be idle. Increasing the warehouse size, which will increase the number of available nodes, will not improve performance because a single file will still be processed on only one node, regardless of how many nodes are available. Techniques for working with virtual warehouses to improve performance are discussed in more detail in chapter 9.

When many small files are loaded, they will be processed in parallel on separate nodes, but each file will have a processing overhead. Also, there might be queuing while files wait their turn to be processed until a node becomes available.

Snowflake recommends using a separate virtual warehouse for data loading and another for querying. This allows each warehouse to be sized and configured according to the workload requirements. For example, a warehouse dedicated to loading can be sized optimally for the number of files that are typically available for a load and the required performance. When using the same warehouse for querying, users might notice a decline in performance or queuing of their queries while the warehouse processes file loading.

3.3.2 Organizing data by path

Files in internal or external stages can be organized by path. This means files are stored in logical groupings such as by date, geographical origin, or other criteria. Such file organization enables data engineers to ingest data from individual paths rather than accessing all files in the storage location.

For example, the restaurant provides order information to the bakery and organizes its CSV files by month and year. Instead of storing all files in the `orderfiles` container, the files are stored in a different path each month, for example, `orderfiles/202308`, `orderfiles/202309`, and so on.

If you are following along, you can upload more order files from the GitHub repository to the storage container. In the Chapter_03 folder, there is a folder named 202308 and, in this folder, two additional order files named Orders_2023-08-07.csv and Orders_2023-08-08.csv. When you upload these files to the storage container, make sure that you expand the Advanced options and enter 202308 in the Upload to the folder text box so that the files will end up in the same folder in the container.

We can view files that are now available in the external stage by executing the `LIST` command (alternatively, we can refresh the stage and select from the directory table):

```
list @BISTRO_STAGE;
```

The output of this command shows all files in the external stage with their path as a prefix. In our example, there are three files in the stage, one in the top level of the container and two more in the 202308 folder:

- `azure://bakeryorders001.blob.core.windows.net/orderfiles/202308/Orders_2023-08-07.csv`
- `azure://bakeryorders001.blob.core.windows.net/orderfiles/202308/Orders_2023-08-08.csv`
- `azure://bakeryorders001.blob.core.windows.net/orderfiles/Orders_2023-08-04.csv`

When files are organized by path, we can load data from the external stage by filtering for a chosen path. For example, if we want to load only files from the 202308 folder, then the `COPY` command from listing 3.3 can be rewritten as

```
copy into ORDERS_BISTRO_STG
from (
  select $1, $2, $3, $4, $5, metadata$filename, current_timestamp()
  from @BISTRO_STAGE/202308
)
file_format = ORDERS_CSV_FORMAT
on_error = abort_statement;
```

Adds the path

The `COPY` command can include other options that limit the set of files for loading, such as providing a list of filenames to load or selecting files according to a pattern that matches the filenames.

3.4 Building pipelines with external tables

An alternative way of using the `COPY` command is to load data from cloud storage using *external tables*. An external table is a Snowflake object that allows you to query data from an external stage, just like any other table in Snowflake. Because the data is stored in an external stage, physically located outside of Snowflake, we can only read the data in the external table but not modify it. Additionally, because data is physically located in the external stage, query performance is slower than when querying data from tables inside Snowflake.

Some of the advantages of using external tables are

- Data stored in files in external stages can be queried using standard SQL commands, just like any other table in Snowflake.
- Because data is stored outside of Snowflake, it doesn't consume storage costs and remains available for other systems to use.

On the other hand, there are disadvantages to using external tables, such as

- Query performance is slower than querying data from standard Snowflake tables.
- External tables are read-only, which means that we can't perform data manipulation language (DML) operations, and we can't use them instead of regular tables.

A *materialized view* can persist the data to improve performance when using external tables. A data pipeline can present data from files in an external stage as an external table. Then, a materialized view is created without a `COPY` command, as illustrated in figure 3.2.

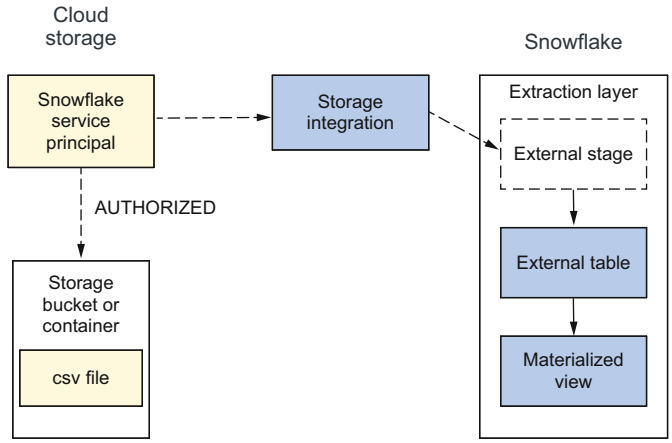


Figure 3.2 Data pipeline that takes CSV files from a cloud storage location, makes the files available to Snowflake in an external stage, presents the data as an external table, and persists the data in Snowflake as a materialized view

3.4.1 Querying data in external stages with external tables

We can create an external table from the BISTRO_STAGE external stage that will allow us to query the data in the external stage using standard SQL queries. Before creating the external table, we must know the file format. Since we already know the file format of the files in the restaurant's stage, we can use the ORDERS_CSV_FORMAT file format we created earlier.

Each external table includes built-in metadata columns we can reference when creating the table. These columns include

- VALUE—This column contains each row of data in the file in the external stage as a VARIANT data type. We can use this column to parse individual columns within the data type by referencing them with c1 for the first column, c2 for the second column, and so on.
- METADATA$FILENAME—The name and path of the file in the external stage.
- METADATA$FILE_ROW_NUMBER—The row number of a record in each file in the external stage.

We can create the ORDERS_BISTRO_EXT external table with the command in the following listing using these metadata columns, the stage name, and the file format.

Listing 3.4 Creating an external table from an external stage

```
create external table ORDERS_BISTRO_EXT (
  customer varchar as (VALUE:c1::varchar),
  order_date date as (VALUE:c2::date),
  delivery_date date as (VALUE:c3::date),
  baked_good_type varchar as (VALUE:c4::varchar),
  quantity number as (VALUE:c5::number),
  source_file_name varchar as metadata$filename
)
location = @BISTRO_STAGE
auto_refresh = FALSE
file_format = ORDERS_CSV_FORMAT;
```

We can query the data in the external table by executing a SQL SELECT command—for example:

```
select *
from ORDERS_BISTRO_EXT;
```

Like directory tables on stages, external tables can be configured to refresh automatically when new files are added or removed from an external stage by configuring the event notification service in the cloud storage provider. If the notification service is not available, external tables can also be refreshed manually, for example, using the following command:

```
alter external table ORDERS_BISTRO_EXT refresh;
```

Because querying external tables is generally slower than querying regular Snowflake tables, we can create materialized views that persist data from an external table in Snowflake.

3.4.2 Using materialized views to improve query performance

A *materialized view* is a data set created from a query and stored for use in further queries. Because the query result is stored, querying a materialized view is faster than executing the query used to build the materialized view.

Materialized views are beneficial when the query is on an external table, which usually performs slower than queries executed on regular Snowflake tables. Once a materialized view is defined, Snowflake takes care of the maintenance. This means that whenever data in the external table used in the materialized view is updated, the materialized view is updated as well. The materialized view contains the same data as the external table, but query performance is better when users query the materialized view because data is stored in Snowflake.

As explained previously, the data in the external table refreshes automatically when new files are added to the external stage if an event notification service is configured. Otherwise, we must refresh external tables manually. Once the external table is refreshed, automatically or manually, the materialized view is refreshed without additional user action.

We can create a materialized view named `ORDERS_BISTRO_MV` based on the `ORDERS_BISTRO_MV` external table using the following command:

```
create materialized view ORDERS_BISTRO_MV as
select customer, order_date, delivery_date,
  baked_good_type, quantity, source_file_name
from ORDERS_BISTRO_EXT;
```

We can query the data in the materialized view by executing a SQL `SELECT` command—for example:

```
select *
from ORDERS_BISTRO_MV;
```

This `select` statement returns the same data in the external table, but the query executes faster.

Although creating external tables and materialized views is an alternative method of creating pipelines instead of using the `COPY` command, there are some drawbacks, such as

- Materialized views are only available in Snowflake Enterprise Edition or higher. If you have the Standard Edition, you can't use materialized views.
- Maintenance of materialized views adds cost, both in storage required to save the data and in compute resources needed to maintain the materialized view.

Data engineers should work with system architects to find the best approach for loading data from cloud storage based on various factors and user requirements, such as prioritizing faster response times or optimizing costs.

Summary

- An external stage is a Snowflake object that exists in a schema. This object stores the location of the files in cloud storage, the parameters used to access the cloud storage account, and additional parameters, such as the options that describe the format of staged files.
- An external stage can be created using a Snowflake storage integration or by providing credentials that authenticate Snowflake to the cloud storage provider.
- A storage integration object is an object that authenticates Snowflake to the cloud storage provider once and uses this object to create external stages.
- Like loading data from files in internal named stages, as explained in chapter 2, we can use the `COPY` command to load data from files in external stages into a staging table.
- When we execute a `COPY` command, Snowflake tracks metadata for each table into which data is loaded from staged files. This ensures that subsequent or simultaneous executions of the `COPY` command don't load data from the same file again so that data is not duplicated.
- We can create a named file format instead of specifying the same format each time we execute the `COPY` command. This is especially useful when we repeatedly load files that share the same format.
- When creating an internal or an external stage, we can add a directory table that stores metadata about the files in a stage. A directory table is an object associated with a stage but not a separate stand-alone Snowflake object.
- The size and number of clusters of Snowflake virtual warehouses used for loading data from files are important when planning for efficient loading. Snowflake recommends that files be sized somewhere between 100 MB and 250 MB compressed. Files exceeding the recommended size should be split into smaller files, while many small files should be aggregated into larger ones.
- A Snowflake virtual warehouse contains nodes consisting of resources, such as CPU, memory, and temporary storage, that the `COPY` command needs to load data from files into Snowflake tables. A single node can process each file.
- Snowflake recommends using a separate virtual warehouse for data loading and another for querying. This allows each warehouse to be sized and configured according to the workload requirements.
- Files in internal or external stages can be organized by logical groupings such as by date, geographical origin, or other criteria. This enables data engineers to ingest data from individual paths rather than by accessing all files in the storage location.

- An alternative way of using the `COPY` command is to load data from cloud storage using external tables. An external table is a Snowflake object that allows you to query data from an external stage, just like any other table in Snowflake.
- A materialized view is a data set created from a query and stored for use in further queries. Materialized views are beneficial when the query is on an external table because they perform better than querying the external table directly.

4 Transforming data

This chapter covers

- Ingesting semistructured data from cloud storage
- Flattening semistructured data into relational tables
- Encapsulating transformations with stored procedures
- Implementing exception handling and logging in stored procedures
- Building robust data pipelines

In this chapter, we will enhance the data pipeline that ingests data from cloud storage, which we built in chapter 3. We will add more functionality in the data transformation part of the pipeline. We will also add exception handling and logging, which will aid in building a robust data pipeline that is as resistant to unintentional errors as possible.

The first type of transformation we will add to the pipeline is flattening semistructured data in JSON format into relational tables. Then we will perform additional data transformations using stored procedures. We will add the data transformation steps to the data pipeline.

Next we will enhance the pipeline by adding exception handling and logging. In case of any errors or incomplete executions, we want to ensure that the data pipeline is robust so that it can be restarted and that it will not duplicate data when executed a second time after fixing an error.

To illustrate the examples used to build the pipeline, we will continue using the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes several kinds of bread and pastries and delivers these baked goods to small businesses, such as grocery stores, coffee shops, and restaurants in the neighborhood.

Since the bakery has no online ordering system, customers order baked goods via email. The bakery stores these orders in CSV files on their local file system. One of the bakery's customers, a nearby restaurant, provides daily order information in the form of CSV files that are stored in a dedicated blob storage container within its cloud storage platform. The bakery built two data pipelines to ingest these files, as described in chapters 2 and 3:

- A bakery employee uploads the CSV files from the local file system into a Snowflake internal named stage. From here, the data pipeline ingests data from the files into Snowflake tables.
- A second data pipeline accesses the restaurant's files via an external stage that points to the restaurant's cloud storage container. The pipeline then ingests data from the files into Snowflake tables.

A small hotel in the neighborhood wants to start buying baked goods from the bakery. The hotel has an information system in place. One of its functionalities is to gather order information and create files with the orders in JSON format in cloud storage.

The bakery will ingest the files in JSON format from the hotel's cloud storage and build a data pipeline like the pipeline that ingests orders from the restaurant's cloud storage described in chapter 3. The main difference between these two pipelines is that the restaurant places orders in CSV format that can be readily converted to a relational format, whereas the hotel places orders in JSON format that we must flatten to a relational data structure.

> **NOTE** All code and sample data files for this chapter are available in the accompanying GitHub repository in the Chapter_04 folder at https://mng.bz/PNJ5. The SQL code is stored in multiple files whose names start with Chapter_4_Part*, where * indicates a sequence number. Please follow the exercises by reading the files sequentially.

To build the pipeline that ingests data from the hotel's files from cloud storage, the bakery data engineers will create an external stage to make the data files available to Snowflake. They will use a storage integration object with the hotel's cloud storage provider, which they will set up initially.

Once the external stage is created, we will use the `COPY` command to copy data from the stage to a staging table. Figure 4.1 illustrates the three data pipelines that

ingest CSV files from the bakery's local file system, CSV files from the restaurant's cloud storage provider, and JSON files from the hotel's cloud storage provider. This chapter describes the third pipeline that ingests data from files in the hotel's cloud storage.

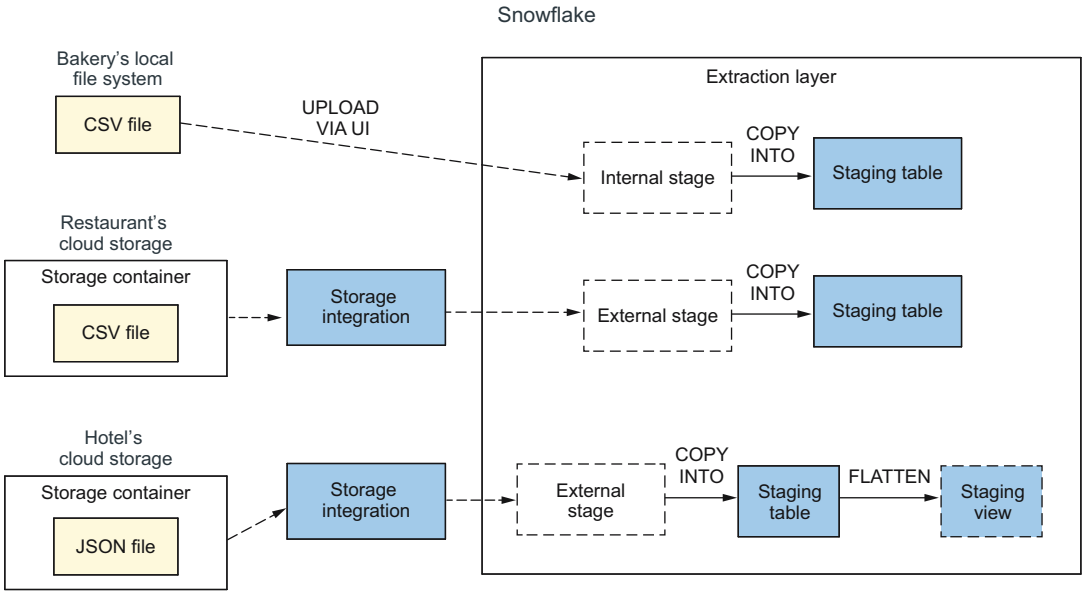


Figure 4.1 Three data pipelines that ingest order information data from CSV files in the bakery's local file system, CSV files in the restaurant's cloud storage, and JSON files in the hotel's cloud storage

4.1 Ingesting semistructured data from cloud storage

To build a data pipeline that ingests files in JSON format from the hotel's cloud storage, we will start by creating a storage integration object, as we did in chapter 3 when we built the data pipeline that ingests the restaurant's orders from cloud storage. Unlike the restaurant, which uses Microsoft Azure as the cloud storage provider, the hotel uses Amazon S3. We will use Amazon S3 for the examples in this chapter.

> **TIP** If you are following along using your Microsoft Azure account, you can continue to use it for the examples in this chapter. Following the description in chapter 3, you can create a storage integration object and an external stage in Microsoft Azure.

We will prepare files in an Amazon S3 bucket for the exercises in this chapter. You will start creating resources in your AWS account by performing the following steps:

1. Create an IAM Policy and an IAM Role by following the instructions in the Snowflake documentation at https://mng.bz/JNJQ. Take note of the Role ARN

value on the role summary page in AWS. For example, the Role ARN may have a value such as `arn:aws:iam::567890987654:role/Snowflake-demo`.

2. Create an S3 bucket.

NOTE When choosing the name for your S3 bucket, remember that bucket names must be between 3 and 63 characters in length and may contain only lowercase letters, numbers, dots, and hyphens. Additionally, your storage account name must be unique across all AWS accounts in all the AWS regions. The exercises in this chapter use `parkinnorders001` as the bucket name. If you want to name your bucket similarly but the name is unavailable, change the suffix from `001` to another combination of digits, and modify any code accordingly.

Once the resources are set up, you can navigate to the `parkinnorders001` bucket and upload one of the sample files, Orders_2023-08-11.json, from the GitHub repository in the Chapter_04 folder to the bucket.

4.1.1 Creating a storage integration

To create a storage integration, the hotel must provide the following information related to its storage account:

- The Role ARN
- The name of the S3 bucket

In our example, the Role ARN is `arn:aws:iam::567890987654:role/Snowflake-demo` (this is a fictional Role ARN for illustrative purposes), and the name of the S3 bucket is `parkinnorders001`.

Using this information, we can create the `PARK_INN_INTEGRATION` storage integration using the following command:

```
use role ACCOUNTADMIN;
create storage integration PARK_INN_INTEGRATION
  type = external_stage
  storage_provider = 'S3'
  enabled = true
  storage_aws_role_arn = 'arn:aws:iam::567890987654:role/Snowflake-demo'
  storage_allowed_locations = ('s3://parkinnorders001/');
```

After executing the command, a storage integration object is created in the Snowflake account. We can view the details of the storage integration by executing the `DESCRIBE STORAGE INTEGRATION` command as follows:

```
describe storage integration PARK_INN_INTEGRATION;
```

Table 4.1 shows a sample output of this command (your output will contain your AWS IAM User, IAM Role, and external ID).

Table 4.1 Sample output of the `DESCRIBE STORAGE INTEGRATION` command

Property	Property type	Property value
ENABLED	Boolean	
STORAGE_PROVIDER	String	S3
STORAGE_ALLOWED_LOCATIONS	List	s3://parkinnorders001/
STORAGE_BLOCKED_LOCATIONS	List	
STORAGE_AWS_IAM_USER_ARN	String	arn:aws:iam::345678987654:user/nxl3-s-eusc3487
STORAGE_AWS_ROLE_ARN	String	arn:aws:iam::567890987654:role/Snowflake-demo
STORAGE_AWS_EXTERNAL_ID	String	ZY34586_SFCRole=2_NbxDAb2mWHQom1HDCwpeyA8RDsEv

This command returns the properties of the storage integration. We must take note of the values of the following properties:

- `STORAGE_AWS_IAM_USER_ARN`—The value of this parameter might look like `arn:aws:iam::345678987654:user/nxl3-s-eusc3487` (this is a fictional User ARN for illustrative purposes).
- `STORAGE_AWS_EXTERNAL_ID`—The value of this parameter might look like `ZY34586_SFCRole=2_NbxDAb2mWHQom1HDCwpeyA8RDsEv` (this is a fictional external ID for illustrative purposes). It represents the external ID needed to establish a trust relationship between Snowflake and AWS.

We now go back to the AWS console, where we will grant the IAM User the necessary permissions to access files in the S3 bucket. Detailed instructions are available in the Snowflake documentation in the "Grant the IAM User Permissions to Access Bucket Objects" section at https://mng.bz/w5Wg.

In our example, the AWS administrator, not the data engineer, performs the AWS administration steps. If you are following along using your AWS account, you will also be the AWS administrator.

Once we have created the storage integration and completed the authorization steps in AWS, we can use it to create an external stage. We will grant usage on the storage integration object to the `SYSADMIN` role, which we will use to create the external stage and build the pipeline. We will accomplish this by executing the following command:

```
grant usage on integration PARK_INN_INTEGRATION to role SYSADMIN;
```

> **TIP** In these initial chapters, we are using the `SYSADMIN` role to create objects in Snowflake for simplicity. Usually, data engineers use custom roles set up in the Snowflake account, but since we haven't created any custom roles, we will use the built-in roles.

4.1.2 Creating an external stage

We will now switch to the SYSADMIN role. To keep the objects that are related to externally sourced data from JSON files separate from the objects that we created previously, we will create a new schema named EXTERNAL_JSON_ORDERS in the BAKERY_DB database using the following commands:

```
use role SYSADMIN;
use database BAKERY_DB;
create schema EXTERNAL_JSON_ORDERS;
use schema EXTERNAL_JSON_ORDERS;
```

Then we will create an external stage named PARK_INN_STAGE using the PARK_INN_INTEGRATION storage integration. We will provide the location of the files in the storage container we received from the hotel (the name of the storage account is parkinnorders001, and the container's name is orderjsonfiles). Because we know that the files are in JSON format, we will provide the FILE_FORMAT parameter as type = json. We can execute the following command to create the external stage:

```
create stage PARK_INN_STAGE
  storage_integration = PARK_INN_INTEGRATION
  url = 's3://parkinnorders001/'
  file_format = (type = json);
```

To view the contents of the external stage, we can execute the following command:

```
list @PARK_INN_STAGE;
```

The file Orders_2023-08-11.json that you uploaded to the S3 bucket earlier will be visible in the output from this command.

4.1.3 Examining the JSON structure

Let's view the data in the staged file. We can use the SELECT command on the stage, but remember that when selecting data from a stage, we don't use the SELECT * syntax to denote all columns, like in usual SQL queries. Instead, we use the $ notation to refer to columns in the file—for example, $1 to indicate the first column. We can execute the following command to view the data in the staged file:

```
select $1 from @PARK_INN_STAGE;
```

The output of this command should return the JSON data that contains the hotel's order information. The JSON data looks like the following listing (not all output is shown).

Listing 4.1 JSON data that contains the hotel's order information

```
{  "Customer": "Park Inn",
   "Order date": "2023-08-11",
   "Orders": [
     {  "Delivery date": "2023-08-14",
```

```
        "Orders by day": [
          {  "Baked good type": "English Muffin",
             "Quantity": 30
          },
          ...
          {  "Baked good type": "Croissant",
             "Quantity": 36
          }
        ]

    },
    {  "Delivery date": "2023-08-15",
       "Orders by day": [
          {  "Baked good type": "English Muffin",
             "Quantity": 26
          },
          ...
          {  "Baked good type": "Croissant",
             "Quantity": 30
          }
       ]
    }
  ]
}
```

We can deduce the structure of the JSON data from this output. There are three key-value pairs at the highest level of the hierarchy with the following keys: "Customer," "Order date," and "Orders." The value of the "Orders" key contains two key–value pairs with keys named "Delivery date" and "Orders by day." The value of the "Orders by day" key contains a list of key–value pairs with keys named "Baked good type" and "Quantity." For a better understanding, the JSON structure from listing 4.1 is illustrated in figure 4.2.

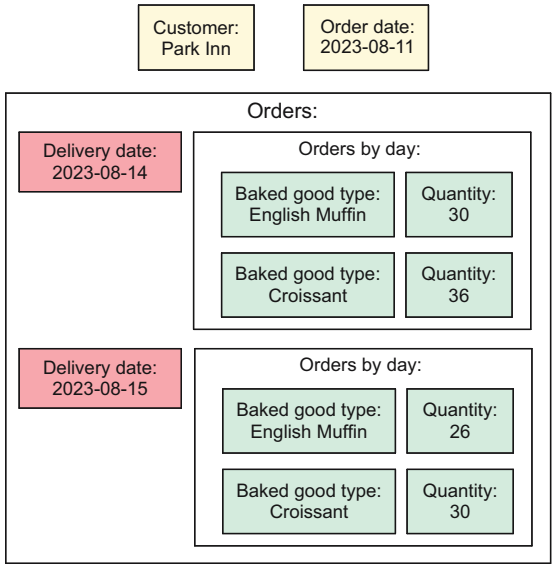


Figure 4.2 Graphical representation of the hotel's JSON structure, illustrating the hierarchical nature of the keys, with "Customer," "Order date," and "Orders" at the highest level, "Delivery date" and "Orders by day" at the next level under the "Orders" key, and finally, "Baked good type" and "Quantity" at the lowest level under the "Orders by day" key

4.1.4 Ingesting JSON data into a VARIANT data type

Let's build a data pipeline that ingests this JSON data into Snowflake and flatten it into a relational table.

> **Snowflake support for semistructured data**
>
> In addition to the JSON format discussed in this chapter, Snowflake can ingest semistructured data from Avro, ORC, Parquet, and XML formats. Semistructured data contains keys or other types of markup that identify entities within the data. These keys are often organized hierarchically by nesting. Unlike structured data stored in relational tables, semistructured data doesn't require a predefined schema, and the schema may also change over time. Snowflake stores semi-structured data in the `VARIANT` data type. This data type may comprise other data types, including `OBJECT`, `ARRAY`, and additional `VARIANT` data types.
>
> More information about semistructured data is available in the Snowflake documentation at https://mng.bz/qOGz.

First, we will create a table in Snowflake that will store the raw JSON data. Here we can use the Snowflake `VARIANT` data type to store semistructured data, such as JSON. We will create a table named `ORDERS_PARK_INN_RAW_STG` containing a `VARIANT` column named `CUSTOMER_ORDERS` to store the JSON data. We will add two columns: `SOURCE_FILE_NAME`, which will store the file's name in the external stage from where the data originates, and the `LOAD_TS` column, which will indicate the timestamp when we loaded the data into Snowflake. We can use the following command to create this table:

```
use database BAKERY_DB;
use schema EXTERNAL_JSON_ORDERS;
create table ORDERS_PARK_INN_RAW_STG (
  customer_orders variant,
  source_file_name varchar,
  load_ts timestamp
);
```

We can use the `COPY` command to load data from the external stage into this table, as already discussed in previous chapters. We will execute the following command:

```
copy into ORDERS_PARK_INN_RAW_STG
from (
  select
    $1,
    metadata$filename,
    current_timestamp()
  from @PARK_INN_STAGE
)
on_error = abort_statement;
```

Once we execute the `COPY` command, we can view the data in the `ORDERS_PARK_INN_RAW_STG` table by querying it with a `SELECT` statement:

```
select *
from ORDERS_PARK_INN_RAW_STG;
```

As the previous command's output, we should see the JSON structure, like in listing 4.1, and two additional columns: the filename in cloud storage from where the data originates and the load timestamp, as shown in table 4.2.

Table 4.2 Sample output when selecting from the `ORDERS_PARK_INN_RAW_STG` table (the column `Customer orders` is not shown in full)

Customer orders	Source filename	Load timestamp
`{"Customer": "Park Inn", "Order date": 2023-08-11", "Orders": [{"Delivery date": "2023-08-14", "Orders by day": [{"Baked good type": "English Muffin", "Quantity": 30…`	Orders_2023-08-11.json	2023-08-11 07:57:15.882

Now that we have the JSON data in a table in Snowflake, let's flatten it so that we can store it in a relational table for downstream use by the bakery manager, who wants to know how many of each type of baked goods are ordered in the following days.

4.2 Flattening semistructured data into relational tables

Snowflake provides the syntax for querying semistructured data such as JSON. In this chapter, we will use the syntax required to flatten the hotel's JSON files into a relational format that the bakery can use.

> **NOTE** For more information about querying semistructured data and a more detailed description of the syntax, please refer to the Snowflake documentation at https://mng.bz/75r4.

As we already know from listing 4.1, the JSON data provided by the hotel is in a hierarchical structure with three keys at the highest level of the hierarchy: "Customer," "Order date," and "Orders." To view the value of each key, we can use the `:` operator to select the key value from the JSON data stored in the `CUSTOMER_ORDERS` column. For example, to view the value of the "Customer" key, we can use the following syntax:

```
customer_orders:"Customer"
```

The following listing shows the complete SQL query that selects the values of each of the three keys at the highest level of the hierarchy.

Listing 4.2 Selecting values from keys at the highest level of the hierarchy

```
select
  customer_orders:"Customer"::varchar as customer,
  customer_orders:"Order date"::date as order_date,
  customer_orders:"Orders"
from ORDERS_PARK_INN_RAW_STG;
```

The output of the query in listing 4.2 returns data presented in table 4.3 (the column `Orders` is not shown in full).

Table 4.3 Values from JSON keys at the highest level of the hierarchy

Customer	Order date	Orders
Park Inn	2023-08-11	`[{"Delivery date":"2023-08-14","Orders by day":[{"Baked good type":"English Muffin","Quantity":30}...`

When we select a key from a JSON structure, the data type of the returned value is `VARIANT`. To convert the value to the corresponding data type, we must explicitly convert it. In the query in listing 4.2, we use the `::` Snowflake operator to convert the "Customer" key's data type to varchar and the "Order date" key to date. We are not converting the value of the "Orders" key because this value contains a hierarchical structure that we will parse in the next step.

Error-handling versions of data type conversion functions

Source data is not always in the correct format, which can cause data ingestion to fail. For example, if the `Order date` column doesn't contain a valid date, the `::date` operation will throw an error, and ingestion will not succeed. To avoid problems related to data type conversion when ingesting data, Snowflake provides error-handling versions of data type conversion functions, such as `TRY_TO_DATE`, `TRY_TO_NUMBER`, etc. Each function tries to convert the data into the desired data type, but if the data doesn't match the data type, the function returns a `NULL` value instead of an error.

For more information about the error-handling versions of data type conversion functions, refer to the Snowflake documentation at https://mng.bz/mRe0.

The "Orders" key value contains an array we recognize because it is enclosed in square brackets []. Inside the square brackets, there are two keys named "Delivery date" and "Orders by day" (refer to figure 4.2, which shows the entire structure of the JSON document). We want to flatten these two keys into separate rows to store them in a relational table. To do that, we will use the `LATERAL FLATTEN` syntax.

Snowflake provides the FLATTEN function for use on a VARIANT value. This function returns an individual row for each object within the VARIANT value. The LATERAL modifier joins the data produced by the FLATTEN function with columns outside the object. In our example, we are flattening the "Orders" key and joining it with the columns CUSTOMER and ORDER_DATE, which we have already extracted from the JSON structure.

When we use the LATERAL FLATTEN syntax, the output value from this function is a column named VALUE. We can refer to this column to extract the values of the "Delivery date" and "Orders by day" keys that are nested in the value of the "Orders" key, which we are flattening.

We can execute the following query to flatten the "Orders" key into individual rows that each contain the "Orders by day" and the "Delivery date" keys.

Listing 4.3 Selecting values from keys at the second level of the hierarchy

```
select
  customer_orders:"Customer"::varchar as customer,
  customer_orders:"Order date"::date as order_date,
  value:"Delivery date"::date as delivery_date,
  value:"Orders by day"
from ORDERS_PARK_INN_RAW_STG,
lateral flatten (input => customer_orders:"Orders");
```

This query's output returns data presented in table 4.4 (the column Orders by day is not shown in full). The LATERAL FLATTEN syntax in the query produced two rows because the JSON file contains orders for two delivery dates.

Table 4.4 Values from JSON keys at the second level of the hierarchy

Customer	Order date	Delivery date	Orders by day
Park Inn	2023-08-11	2023-08-14	[{"Baked good type":"English Muffin","Quantity":30}...
Park Inn	2023-08-11	2023-08-15	[{"Baked good type":"English Muffin","Quantity":26}...

The value of the "Orders by day" key contains an array, meaning we must use an additional LATERAL FLATTEN function in the query to flatten these orders into relational tables. This second LATERAL FLATTEN function will again return a VALUE column, like the first one. To distinguish between them, we will use an alias for each LATERAL FLATTEN function (CO and DO).

The "Orders by day" key contains a list of keys named "Baked good type" and "Quantity." We will select the values from these keys using the query in the following listing.

Listing 4.4 Selecting values from JSON keys at the third level of the hierarchy

```
select
  customer_orders:"Customer"::varchar as customer,
  customer_orders:"Order date"::date as order_date,
  CO.value:"Delivery date"::date as delivery_date,
  DO.value:"Baked good type":: varchar as baked_good_type,
  DO.value:"Quantity"::number as quantity
from ORDERS_PARK_INN_RAW_STG,
lateral flatten (input => customer_orders:"Orders") CO,
lateral flatten (input => CO.value:"Orders by day") DO;
```

First LATERAL FLATTEN

Second LATERAL FLATTEN

The output of this query returns data that is presented in table 4.5. We can see that the JSON data is now flattened to the lowest level of the hierarchy, and there are no more nested objects or lists.

Table 4.5 Values from JSON keys at the third level of the hierarchy

Customer	Order date	Delivery date	Baked good type	Quantity
Park Inn	2023-08-11	2023-08-14	English muffin	30
Park Inn	2023-08-11	2023-08-14	Whole wheat loaf	6
Park Inn	2023-08-11	2023-08-14	White loaf	4
Park Inn	2023-08-11	2023-08-14	Bagel	25
Park Inn	2023-08-11	2023-08-14	Croissant	36
Park Inn	2023-08-11	2023-08-15	English muffin	26
Park Inn	2023-08-11	2023-08-15	Whole wheat loaf	4
Park Inn	2023-08-11	2023-08-15	Bagel	22
Park Inn	2023-08-11	2023-08-15	Croissant	30

Now that we understand the query that transforms data from JSON format into a relational structure, we can use this query to load data into a Snowflake table. To reduce complexity and storage cost by not copying the data from the `ORDERS_PARK_INN_RAW_STG` raw staging table to a relational staging table, we can create a view representing the raw JSON data in relational format.

Let's create a view named `ORDERS_PARK_INN_STG` using the syntax from listing 4.4 and adding the columns that contain the name of the source file and load timestamp:

```
create view ORDERS_PARK_INN_STG as
select
  customer_orders:"Customer"::varchar as customer,
  customer_orders:"Order date"::date as order_date,
  CO.value:"Delivery date"::date as delivery_date,
  DO.value:"Baked good type":: varchar as baked_good_type,
  DO.value:"Quantity"::number as quantity,
```

The query from listing 4.4

```
  source_file_name,
  load_ts
from ORDERS_PARK_INN_RAW_STG,
lateral flatten (input => customer_orders:"Orders") CO,
lateral flatten (input => CO.value:"Orders by day") DO;
```

This view represents staged data from JSON files flattened into a relational format.

> **Overcoming the 16 MB data size limitation**
>
> The data stored in a `VARIANT` data type in a Snowflake table is limited to 16 MB. When files are larger than 16 MB, we cannot directly load semistructured data into a `VARIANT` data type.
>
> One way to overcome this limitation with JSON is to ask the data provider to create structured files so that each JSON document contains an array of multiple objects with the same structure. In such cases, we can use the `STRIP_OUTER_ARRAY` keyword in the `COPY` command. The command loads each object in the array into a separate row in the target table.
>
> For more information, please refer to the Snowflake documentation at https://mng.bz/5OBq.

4.3 Encapsulating transformations with stored procedures

Let's summarize the staging tables and views we have implemented until now. Three pipelines populate three staging tables or views:

- In chapter 2, we created the `ORDERS_STG` staging table in the `ORDERS` schema that contains data from CSV files that the bakery manually uploads to an internal stage.
- In chapter 3, we created the `ORDERS_BISTRO_STG` table in the `EXTERNAL_ORDERS` schema that contains data from CSV files that are hosted in the restaurant's cloud storage.
- In this chapter, we created the `ORDERS_PARK_INN_STG` view in the `EXTERNAL_JSON_ORDERS` schema that contains data from JSON files that are hosted in the hotel's cloud storage.

All three staging tables or views have the same structure, which includes the following columns: customer, order date, delivery date, baked goods type, quantity, filename, and load timestamp.

Once we have the data ingested in staging tables, we can continue transforming it so downstream consumers can use it. One requirement for downstream consumption is from the bakery manager, who wants to know how many of each type of baked good has been ordered for the following days. This information is needed so the bakery can plan the raw materials for production and staff schedules accordingly.

To support the bakery manager's requirement, we will build additional steps into the data pipeline that will perform the following tasks:

- Combine data from all staging tables and views into a single data structure
- Load the combined staging data into a target table
- Summarize the data by day and by type of baked good

We will first create a new schema named `TRANSFORM` in the `BAKERY_DB` database, containing all the transformation objects. To create the schema, we can execute the following commands:

```
use database BAKERY_DB;
create schema TRANSFORM;
use schema TRANSFORM;
```

Then, using the `TRANSFORM` schema, we will create a view named `ORDERS_COMBINED_STG` that combines all staging tables and views by executing the following SQL statement:

```
create view ORDERS_COMBINED_STG as
select customer, order_date, delivery_date,
  baked_good_type, quantity, source_file_name, load_ts
from bakery_db.orders.ORDERS_STG
union all
select customer, order_date, delivery_date,
  baked_good_type, quantity, source_file_name, load_ts
from bakery_db.external_orders.ORDERS_BISTRO_STG
union all
select customer, order_date, delivery_date,
  baked_good_type, quantity, source_file_name, load_ts
from bakery_db.external_json_orders.ORDERS_PARK_INN_STG;
```

> **TIP** If you have been following along in chapters 2 and 3, you should have the `ORDERS_STG` and `ORDERS_BISTRO_STG` staging tables already created. If you skipped any of the exercises in the previous chapters, you can continue from this point onwards; just modify the `ORDERS_COMBINED_STG` view to include only your staging tables.

As in chapter 2, we will create a target table that stores historical order data. Each time we receive new data from the bakery's CSV files or new files appearing in the restaurant's or the hotel's cloud storage, we will load those files into their corresponding staging tables. Then we will merge the data from the combined staging tables into the target table.

We will create the `CUSTOMER_ORDERS_COMBINED` target table using the following command:

```
use database BAKERY_DB;
use schema TRANSFORM;
create or replace table CUSTOMER_ORDERS_COMBINED (
  customer varchar,
  order_date date,
```

```
  delivery_date date,
  baked_good_type varchar,
  quantity number,
  source_file_name varchar,
  load_ts timestamp
);
```

Then we will merge the data from the staging tables into the target table. Like in chapter 2, we will merge the data to ensure that we have at most one row per customer, baked goods type, and delivery date with the most recent value for the quantity in the target table.

We will execute the statement in the following listing to merge the data from the `ORDERS_COMBINED_STG` staging data into the `CUSTOMER_ORDERS_COMBINED` target table.

Listing 4.5 Merging customer orders from the staging table into the target table

```
merge into CUSTOMER_ORDERS_COMBINED tgt
using ORDERS_COMBINED_STG as src
on src.customer = tgt.customer
  and src.delivery_date = tgt.delivery_date
  and src.baked_good_type = tgt.baked_good_type
when matched then
  update set tgt.quantity = src.quantity,
    tgt.source_file_name = src.source_file_name,
    tgt.load_ts = current_timestamp()
when not matched then
  insert (customer, order_date, delivery_date,
    baked_good_type, quantity, source_file_name, load_ts)
  values(src.customer, src.order_date, src.delivery_date,
    src.baked_good_type, src.quantity, src.source_file_name,
    current_timestamp());
```

Instead of executing the `MERGE` statement to load data from the staging table into the target table each time new data arrives in the external stage, a data engineer can build a stored procedure that encapsulates the logic. This stored procedure can then be executed as a single procedure call by a task or other orchestration mechanisms. In addition, stored procedures can include programming logic such as branching and looping, exception handling, and logging.

We can create stored procedures in Snowflake in any supported programming language, including SQL, Snowflake Scripting, JavaScript, Java, Python, or Scala. Note that Java, Python, and Scala stored procedures are available via the Snowpark API and are discussed in chapter 6.

In this chapter, we will use Snowflake Scripting to create a stored procedure named `LOAD_CUSTOMER_ORDERS` to load data from the staging table to the target table. We will describe the Snowflake Scripting syntax, limited to what is required to build the stored procedure.

NOTE For more information about creating stored procedures in Snowflake Scripting, please refer to the Snowflake documentation at https://mng.bz/6Yd5.

4.3.1 Creating a basic stored procedure

Let's start with a basic stored procedure that we can modify with more functionality later. The code for creating a stored procedure named `LOAD_CUSTOMER_ORDERS` that executes the `MERGE` statement from listing 4.5 contains the following components:

- The `CREATE PROCEDURE` command followed by the name of the stored procedure and a list of parameters, if using (in our example, we are not using any parameters)
- The `RETURNS` clause that defines the return data type (we will return `VARCHAR`)
- The `LANGUAGE` clause, which defines the programming language in the stored procedure (in our case, SQL, which denotes Snowflake Scripting)
- The body of the stored procedure wrapped in `$$` signs
- The `BEGIN` and `END` statements that enclose the query that the stored procedure will execute

The code that creates this stored procedure is shown in the following listing.

Listing 4.6 Creating a basic stored procedure

```
use database BAKERY_DB;
use schema TRANSFORM;
create procedure LOAD_CUSTOMER_ORDERS()
returns varchar
language sql
as
$$
begin
  merge into CUSTOMER_ORDERS_COMBINED tgt...
end;
$$
;
```

The query from listing 4.5

Once the stored procedure is created, we can execute it using the `CALL` command:

```
call LOAD_CUSTOMER_ORDERS();
```

To verify that the procedure executed correctly, we can select data from the `CUSTOMER_ORDERS_COMBINED` target table to see that the data has been loaded by executing the following SQL statement:

```
select * from CUSTOMER_ORDERS_COMBINED;
```

The data ingested from CSV and JSON files into the staging tables should now be available in this target table.

4.3.2 Including a return value in a stored procedure

The basic stored procedure we just created returns no value. It is often more convenient to code a stored procedure so that it returns a value to the caller. The return value can contain information relevant to executing the stored procedure, such as the number of affected rows.

In our stored procedure, we could return a text message that says "Load completed." We can embellish this message by adding the number of records the SQL command processed. Snowflake keeps track of a built-in global variable named `SQLROWCOUNT`, which stores the number of affected rows processed by the last executed query. We can use this variable in a `RETURN` statement to compose the return value like that shown in the following listing.

Listing 4.7 Constructing a `RETURN` statement

```
return 'Load completed. ' || SQLROWCOUNT || ' rows affected.';
```

After executing the `MERGE` statement, we can add the `RETURN` statement as composed in listing 4.7 to the stored procedure. The modified code for creating the stored procedure is shown in the following listing.

Listing 4.8 Creating a stored procedure that returns a value

```
use database BAKERY_DB;
use schema TRANSFORM;
create or replace procedure LOAD_CUSTOMER_ORDERS()
returns varchar
language sql
as
$$
begin
  merge into CUSTOMER_ORDERS_COMBINED tgt...
  return 'Load completed. ' || SQLROWCOUNT || ' rows affected.';
end;
$$
;
```

The query from listing 4.5

The RETURN statement from listing 4.7

We can again execute the stored procedure using the `CALL` command:

```
call LOAD_CUSTOMER_ORDERS();
```

Because the stored procedure executes a `MERGE` statement, which both inserts and updates the rows in the target table, the number of affected rows equals the number of rows in the target table. For example, the return value might be

```
Load completed. 131 rows affected.
```

Now that we have our basic stored procedure, let's add exception handling.

4.3.3 Implementing exception handling in stored procedures

Exception handling is vital to programming because errors, anomalies, or unexpected outcomes can happen. We address exceptions by writing exception-handling code that catches these and performs remediation or notification actions.

In Snowflake Scripting, we handle exceptions by adding an `EXCEPTION` block. If there are no exceptions when we execute the stored procedure, the code completes without executing the `EXCEPTION` block. But if there is an error, the code resumes in the `EXCEPTION` block from the point of failure. In the `EXCEPTION` block, we can perform actions such as reversing a transaction. We can also construct a message that the `EXCEPTION` block returns. The message can include the built-in global variable `SQLERRM`, which stores the error message.

The `EXCEPTION` block can be coded to perform different actions based on different types of exceptions. These can be either Snowflake built-in exceptions or declared as custom exceptions. We use the `WHEN` clause to address each exception.

> **NOTE** For more information about exception handling and defining custom exceptions, refer to the Snowflake documentation at https://mng.bz/o0Oj.

Let's recreate the `LOAD_CUSTOMER_ORDERS` stored procedure by adding an `EXCEPTION` block that will use the `RETURN` statement to return an error message. In the `WHEN` clause, we will not address any individual exceptions; instead, we will use the `OTHER` keyword, which handles all other exceptions. The code that recreates the stored procedure is shown in the following listing.

Listing 4.9 Exception handling in a stored procedure

```
use database BAKERY_DB;
use schema TRANSFORM;
create or replace procedure LOAD_CUSTOMER_ORDERS()
returns varchar
language sql
as
$$
begin
  merge into CUSTOMER_ORDERS_COMBINED tgt...
  return 'Load completed. ' || SQLROWCOUNT || ' rows affected.';
exception
  when other then
    return 'Load failed with error message: ' || SQLERRM;
end;
$$
;
```

The query from listing 4.5

Adds an exception handling block

4.4 Adding logging to stored procedures

When a data pipeline executes on a schedule—for example, every evening—a data engineer usually doesn't monitor it each time to check whether it is error-free.

Instead, data engineers typically maintain logging information to check the log table the next day or periodically to review the execution status.

Data pipelines can send notifications in case of failures or unexpected outcomes; this is described in more detail in chapter 13. For now, we will just add logging to our stored procedure, enabling us to review the status of all previous executions.

One way to add logging to the stored procedure would be to create a custom logging table in our chosen schema. This logging table might contain columns such as the execution timestamp, the stored procedure name, the values of the parameters if used, the caller's username, the outcome status (success or failure), the error message in case of failure, and so on. Then we would add `INSERT` statements to the stored procedure to insert logging data into our custom logging table.

Another way to add logging to stored procedures is to use Snowflake's built-in logging functionality. This feature allows you to record the activity of stored procedures by capturing log messages from the code as it executes. These messages are stored in an event table that we can query with SQL statements.

Table 4.6 summarizes the benefits and drawbacks of each approach to help you decide whether to implement custom logging or use Snowflake's built-in logging.

Table 4.6 Comparison between custom logging and Snowflake built-in logging

Logging type	Benefits	Drawbacks
Custom	Flexibility in what information is needed and when it is logged	Must perform custom development
Snowflake built-in	There is no custom development; perform some initial setup, and call the Snowflake built-in logging constructs.	Shared across the account; therefore, access controls must be in place to allow developers to access their logging messages

In this chapter, we will use Snowflake's built-in logging functionality.

To set up logging for the stored procedure, we will follow these steps:

1. Create an event table. This type of table in Snowflake contains a predefined set of columns.
2. Associate the event table with the account. Each Snowflake account can have only one active event table.
3. Grant privileges to the data engineer's role to allow them to set the logging level.
4. Call the logging constructs from the stored procedure.
5. Use SQL query statements to view the logged data.

NOTE Additional information about Snowflake logging and tracing is available in the documentation at https://mng.bz/n0p8.

The first step when setting up logging is to create an event table. For this exercise, we will create a logging table in one of the existing schemas in the `BAKERY_DB` database.

NOTE In real-world scenarios, we usually create an account-level event table in a dedicated database and schema that stores account-level objects for all applications in the Snowflake account.

We will continue using the SYSADMIN role and the TRANSFORM schema to create an event table named BAKERY_EVENTS by executing the following commands:

```
use role SYSADMIN;
use schema TRANSFORM;
create event table BAKERY_EVENTS;
```

Once the event table is created, we will associate this table with the account. We must use the ACCOUNTADMIN role to perform this action and provide the fully qualified table name. We can do this by executing the following commands:

```
use role ACCOUNTADMIN;
alter account set event_table = BAKERY_DB.TRANSFORM.BAKERY_EVENTS;
```

Next, we must grant the SYSADMIN role used to work with the bakery objects the required privileges to change the log level on the stored procedure. To do that, we will grant the MODIFY LOG LEVEL ON ACCOUNT privilege, which allows the grantee to set the log level on any database, schema, stored procedure, or function in the account. Still using the ACCOUNTADMIN role, we can issue the grant by executing the following command:

```
grant modify log level on account to role SYSADMIN;
```

Now we can switch back to the SYSADMIN role and continue setting up the logging in the stored procedure. We must initially set the desired log level for the stored procedure. The default log level is OFF, meaning nothing will be logged. We can choose to log at any of the following levels, starting from the most restrictive (for example, the FATAL log level will log only fatal errors) to the least restrictive: FATAL, ERROR, WARN, INFO, and DEBUG. Each log level logs messages at that level and above.

In this exercise, let's log all levels of messages. We will, therefore, set the log level for the LOAD_CUSTOMER_ORDERS stored procedure to the least restrictive DEBUG level by executing the following commands:

```
use role SYSADMIN;
alter procedure LOAD_CUSTOMER_ORDERS() set log_level = DEBUG;
```

Then we will add logging to the LOAD_CUSTOMER_ORDERS stored procedure using the function SYSTEM$LOG_DEBUG, which logs messages at the DEBUG level. The code that recreates the stored procedure with logging is

```
use database BAKERY_DB;
use schema TRANSFORM;
create or replace procedure LOAD_CUSTOMER_ORDERS()
returns varchar
```

```
language sql
as
$$
begin
  SYSTEM$LOG_DEBUG('LOAD_CUSTOMER_ORDERS begin ');
  merge into CUSTOMER_ORDERS_COMBINED tgt...
  return 'Load completed. ' || SQLROWCOUNT || ' rows affected.';
exception
  when other then
    return 'Load failed with error message: ' || SQLERRM;
end;
$$
;
```

Adds logging

The query from listing 4.5

We can execute the stored procedure using the `CALL` command as follows:

```
call LOAD_CUSTOMER_ORDERS();
```

> **TIP** Because Snowflake logging functionality works at the account level, it usually takes some time before the logged messages appear in the event table. You must wait a few minutes before checking that the logging information is visible in the event table.

To query the event table, we can execute the following SQL statement:

```
select *
from BAKERY_EVENTS
order by timestamp desc;
```

We should see the logged message as the output of this statement.

For more information about logging, refer to the Snowflake documentation at https://mng.bz/vJmq.

4.5 Building robust data pipelines

Let's review what we have built and combine all the pieces into a pipeline that the bakery will execute regularly. The data pipeline ingests data from CSV files and JSON files into staging tables. It then inserts the data into a target table. The final step is summarizing the data by baked goods type and day.

Let's create a table named `SUMMARY_ORDERS` that will store summarized data using the following command:

```
use database BAKERY_DB;
use schema TRANSFORM;
create table SUMMARY_ORDERS (
  delivery_date date,
  baked_good_type varchar,
  total_quantity number
);
```

Then we will summarize the data in the CUSTOMER_ORDERS_COMBINED table by the delivery date and baked goods type and insert the summarized data into the SUMMARY_ORDERS table. Since the data pipeline executes on schedule—for example, every evening—we must avoid data duplication. Before inserting the summarized data into the table, we must truncate it using the TRUNCATE command. To insert the summarized data into the target table, we will execute the commands shown in the following listing.

Listing 4.10 Truncating the table and inserting summarized data

```
truncate table SUMMARY_ORDERS;
insert into SUMMARY_ORDERS(delivery_date, baked_good_type, total_quantity)
  select delivery_date, baked_good_type, sum(quantity) as total_quantity
  from CUSTOMER_ORDERS_COMBINED
  group by all;
```

Like previously, when populating the staging table, we will build a stored procedure named LOAD_CUSTOMER_SUMMARY_ORDERS() to encapsulate the SQL statements from listing 4.10. We will execute the following command to create the stored procedure:

```
use database BAKERY_DB;
use schema TRANSFORM;
create or replace procedure LOAD_CUSTOMER_SUMMARY_ORDERS()
returns varchar
language sql
as
$$
begin
  SYSTEM$LOG_DEBUG('LOAD_CUSTOMER_SUMMARY_ORDERS begin ');

  truncate table SUMMARY_ORDERS;
  insert into SUMMARY_ORDERS...
  return 'Load completed. ' || SQLROWCOUNT || ' rows inserted.';
exception
  when other then
    return 'Load failed with error message: ' || SQLERRM;
end;
$$
;
```

The insert statement from listing 4.10

We can execute the stored procedure using the CALL command as follows:

```
call LOAD_CUSTOMER_SUMMARY_ORDERS();
```

The bakery manager can use the data from this summary table to determine how many of each type of baked goods the customers ordered in the following days. We can execute the following query to display this information:

```
select *
from SUMMARY_ORDERS
order by delivery_date desc;
```

The output of this query should list the date, baked goods type, and quantity of each baked goods type ordered for each day.

Figure 4.3 shows the data flow in the pipeline from the staging tables to the final summarized table.

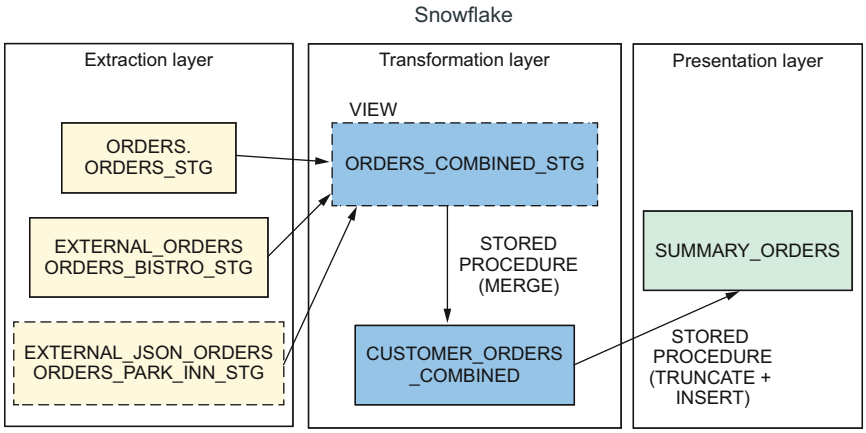


Figure 4.3 Data pipeline that combines data from staging tables, uses a stored procedure to merge the combined staging data into a target table, and summarizes the data for downstream consumption

When building data pipelines, data engineers must carefully consider making them as robust as possible once they start executing on schedule in a production environment. A robust data pipeline is reliable and designed to be resilient to unexpected events requiring restarts. At the same time, it ensures data quality and consistency.

While designing a data pipeline, a data engineer should consider more than just the positive scenario, which is when the pipeline executes error-free. The data engineer must also consider the negative scenarios that can occur for various reasons, such as connection outages, source system failures, unexpected data types, data inconsistencies, timeouts from long-running queries, access control violations, etc.

While designing the data pipeline, the data engineer must consider the course of action in such negative scenarios. For example, they must decide whether the entire data pipeline will be restarted or started from the point of failure after the reason for the failure has been resolved. They must also ensure that data is not duplicated or lost during such adverse scenarios.

Well-designed and robust data pipelines

- Respond to failures without compromising data integrity
- Provide logging and monitoring capabilities to enable visibility into the execution status
- Alert in case of failures or unexpected events

- Perform data validation where feasible, such as validating primary keys
- Are constructed in a modular fashion that allows code reusability
- Are tested in various scenarios before deployment
- Are documented for ease of maintainability

In this chapter, we included some of these aspects when designing the data pipeline. We used a stored procedure as an example of code modularity. We implemented logging to enable visibility into the execution status. We used SQL constructs such as the `MERGE` statement to ensure primary key uniqueness. We truncated the summary table before inserting the summarized data to prevent data duplication. Due to such considerations, we can restart the data pipeline in the event of failure without compromising data integrity.

Summary

- Semistructured data contains key–value pairs that are often organized hierarchically. Unlike structured data, which can be stored in relational tables, semistructured data doesn't require a predefined schema and can be stored in the `VARIANT` data type in Snowflake.
- JSON is a popular semistructured data format for exchanging data in a human-readable format. In this chapter, we demonstrated how a hotel can provide order information to the bakery in JSON format. We can then ingest this data into Snowflake from object storage.
- JSON data can be stored in the `VARIANT` data type in Snowflake. The size limitation is 16 MB.
- Snowflake provides the syntax for querying semistructured data such as JSON. To view the value of each key, we can use the `:` operator to select the key value from the JSON data.
- When we select a key from a JSON structure, the returned value's data type is `VARIANT`. We must use data type conversion functions or operators to convert the value to the corresponding data type.
- Snowflake provides the `FLATTEN` function, which returns an individual row for each object within a JSON document. The `LATERAL` modifier joins the data produced by the `FLATTEN` function with columns outside the object. This syntax allows JSON objects that contain multiple objects, such as a list of orders for a specific date, to be represented as rows in a relational table.
- We can write stored procedures in Snowflake to encapsulate data transformation logic. The procedures can also include programming logic such as branching and looping, exception handling, and logging. We can create stored procedures in Snowflake in any supported programming language, including Snowflake Scripting, JavaScript, Java, Python, or Scala.
- A stored procedure can return a value to the caller. The return value can include an indication of whether the procedure executed successfully as well as

any other information relevant to the procedure's execution, such as the `SQLROWCOUNT` built-in global variable, which stores the number of affected rows processed by the last query that was executed.

- Exception handling is vital to programming because errors, anomalies, or unexpected outcomes can happen. We address exceptions by writing exception-handling code that catches these and performs remediation or notification actions. In Snowflake Scripting, we handle exceptions by adding an `EXCEPTION` block.
- Data engineers usually maintain logging information to check the log table the next day or periodically to review the data pipeline execution status. They can use the Snowflake built-in logging functionality, which allows them to record the activity of stored procedures by capturing log messages from the code as it executes.
- Logging messages are stored in an event table, a type of table in Snowflake with a predefined set of columns. An event table is associated with the account. There can be only one active event table in each Snowflake account.
- Snowflake functions, such as `SYSTEM$LOG_DEBUG`, add logging functionality to stored procedures.
- When building data pipelines, data engineers must carefully consider making them as robust as possible once they start executing on schedule in a production environment. A robust data pipeline is reliable and designed to be resilient to unexpected events requiring restarts. At the same time, it ensures data quality and consistency.

5 Continuous data ingestion

This chapter covers

- Comparing bulk and continuous data ingestion
- Introducing Snowpipe
- Configuring Snowpipe with cloud messaging
- Using and monitoring Snowpipe
- Transforming data continuously with Snowflake dynamic tables

In this chapter, we will build a new data pipeline that continuously ingests data from files whenever they appear in the external cloud storage with minimum delay. We will explain the difference between continuous and bulk data ingestion, which we described in the previous chapters. We will introduce Snowpipe as the Snowflake feature used in data pipelines for continuous data ingestion. Finally, we will use dynamic tables to perform data transformation continuously.

We will build a data pipeline that uses Snowpipe to ingest data from JSON files stored in an external cloud storage location. We will convert the data from JSON format to a relational format. Instead of executing stored procedures as in chapter 4, we will materialize the data by creating dynamic tables.

To illustrate the examples used to build the pipeline in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries and delivers these baked goods to small businesses, such as grocery stores, coffee shops, and restaurants in the neighborhood.

Since the bakery has no online ordering system, the customers place their orders for baked goods via email, and the bakery stores these orders in CSV files on their local file system. Some bakery customers provide daily order information in CSV or JSON files stored in a dedicated container within their cloud storage platform. To ingest all of these files, the bakery data engineers built pipelines that access the files via an internal or external stage. The pipelines ingest data from the files into Snowflake tables and transform them as needed. The pipelines are scheduled to run once daily in the evening, after working hours, so fresh order information is available the following day.

The bakery is expanding its business even more. It signed up with a food delivery service that delivers food orders within the city. Since the delivery service operates during working hours throughout the day, the bakery can't wait for daily file ingestion like in the existing data pipelines because they need the order information from the delivery service whenever a customer orders baked goods. The data engineers must build a new data pipeline that ingests data continuously throughout the day.

The food delivery service has a cloud storage platform that saves orders in files in JSON format. The bakery data engineers will build a data pipeline that resembles the pipeline that ingests order files in JSON format from the hotel's cloud storage described in chapter 4. The main difference between that pipeline is that we ingest the hotel's files once daily but continuously ingest the food delivery service's files using the Snowpipe ingestion service.

> **NOTE** All code and sample data files for this chapter are available in the accompanying GitHub repository in the Chapter_05 folder at https://mng.bz/4plg.

To build the pipeline that ingests data from the food delivery service's files from cloud storage, the bakery data engineers will create an external stage to make the data files available to Snowflake. They will use a storage integration object with the delivery service's cloud storage provider. This step is the same as when ingesting data from cloud storage, described in chapter 4.

Then a Snowpipe will be created and integrated with a notification service in the cloud storage provider. A notification will be sent to a queue whenever a new file arrives in cloud storage, allowing Snowpipe to be aware of and ingest new files. Configuring Snowpipe with cloud messaging will be explained in more detail in the following sections of this chapter. Finally, a dynamic table will materialize the flattened data ingested from files in JSON format. The described data pipeline is illustrated in figure 5.1.

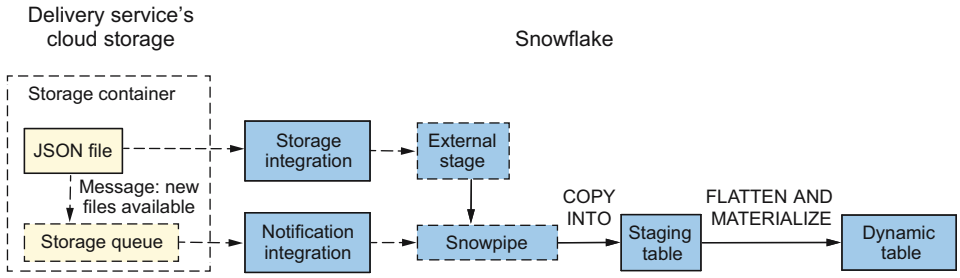


Figure 5.1 Data pipeline that continuously ingests data from JSON files in external cloud storage with Snowpipe. A storage integration object exposes the files in an external stage. A notification integration object configured with cloud messaging notifies Snowpipe when new files arrive so they can be ingested into the staging table with minimum delay. The JSON data in the staging table is flattened and materialized by a dynamic table.

5.1 Comparing bulk and continuous data ingestion

Bulk data loading refers to regularly scheduled data pipelines that ingest data into Snowflake tables. Pipelines are often scheduled to run daily, in the evening or during the night. This type of data loading is useful when new data arrives at known intervals or when data consumers don't require the latest data but can work with data from the previous day.

When consumers require data more frequently, data pipelines can be scheduled to run within shorter intervals—for example, every hour or every few minutes. Such frequent scheduling of data pipelines is often referred to as *micro-batch*. This approach can work if the entire pipeline finishes execution before the next scheduled execution. In practice, this usually doesn't happen quickly enough to satisfy user requirements, since pipelines can't be scheduled in intervals less than the total pipeline execution time.

As an alternative to scheduling data pipelines in micro-batch, Snowflake offers the Snowpipe feature, which is designed to load data from files as soon as they are available in a stage. Snowpipe behaves similarly to micro-batch in that it uses the `COPY` command to load data from files into a Snowflake table, except it is not scheduled by a time interval but by an event trigger. To function correctly, Snowpipe can be configured with cloud messaging, which sends a notification when new files arrive, triggering the pipeline execution.

5.2 Preparing files in cloud storage

To build a data pipeline that ingests files continuously from the food delivery service's cloud storage, we will start by creating a storage integration object, as we did in chapter 4 when we built the data pipeline that ingests the hotel's orders from cloud storage. Like the hotel, the food delivery service uses Microsoft Azure as the cloud storage provider. We will use Microsoft Azure to illustrate the examples in this chapter.

Creating a storage integration in Amazon S3 or Google Cloud Storage

The Snowflake documentation provides details about setting up the storage integration in Amazon S3 and Google Cloud Storage at https://mng.bz/ZVJ9 (Amazon S3) and https://mng.bz/RNJa (Google Cloud Storage).

If you are following along using your own Microsoft Azure account, you can prepare files in blob storage for the exercises in this chapter. You must create resources in your account by performing the following steps:

1. Create a Resource group, or use an existing Resource group if you already have one and want to use it for the exercises. You can name the Resource group however you wish.
2. Create a Storage account in the Resource group.

NOTE When choosing the name for your Storage account, remember that it must be unique within Azure. The exercises in this chapter use `speedyorders001` as the storage account name. If you want to name your storage account similarly but the name is unavailable, change the suffix from `001` to another combination of digits, and modify any code accordingly.

3. Create a Container. You can name the container however you wish, but if you would like to name it the same as in the exercises in this chapter, name it `speedyservicefiles`.

Once the resources are ready, you can navigate to the `speedyservicefiles` container and upload a few sample files named Orders_2023-09-04_12-30-00_12345.json, Orders_2023-09-04_12-30-00_12346.json, and Orders_2023-09-04_12-45-00_12347.json from the GitHub repository in the Chapter_05 folder to the storage container.

TIP Although more files are available in the GitHub repository, upload only a few sample files to the blob storage container at this time. You will upload additional files to the storage container later when we demonstrate how Snowpipe loads new files as soon as they appear in cloud storage.

5.2.1 Creating a storage integration

The food delivery service must provide the following information related to their storage account that is needed to create a storage integration:

- The Azure Tenant ID
- The name of the storage account
- The name of the storage container

In our example, the Azure Tenant ID is `1234abcd-xxx-56efgh78` (this is a fictional Tenant ID for illustrative purposes), the storage account is `speedyorders001`, and the container is `speedyservicefiles`.

Using this information, we can create the SPEEDY_INTEGRATION storage integration using the following command:

```
use role ACCOUNTADMIN;
create storage integration SPEEDY_INTEGRATION
  type = external_stage
  storage_provider = 'AZURE'
  enabled = true
  azure_tenant_id = '1234abcd-xxx-56efgh78'
  storage_allowed_locations =
    ('azure://speedyorders001.blob.core.windows.net/speedyservicefiles /');
```

Like in chapter 4, we will execute the DESCRIBE INTEGRATION command and take note of the AZURE_CONSENT_URL and AZURE_MULTI_TENANT_APP_NAME properties. Then we will work with the food delivery service's Azure administrator, who will accept the Snowflake service principal and grant permissions on the storage container.

Once we have created the storage integration and the Azure administrator completes the authorization steps in Azure, we can use it to create an external stage. We will grant usage on the storage integration object to the SYSADMIN role, which we will use to create the external stage and build the pipeline by executing the following command:

```
grant usage on integration SPEEDY_INTEGRATION to role SYSADMIN;
```

> **TIP** In these initial chapters, we are using the SYSADMIN role to create objects in Snowflake for simplicity. Usually, data engineers use custom roles set up in the Snowflake account, but since we haven't created any custom roles, we will use the built-in roles.

5.2.2 Creating an external stage

We will now switch to the SYSADMIN role. To keep the objects that are related to externally sourced data from the food delivery service separate from the objects that we created previously, we will create a new schema named DELIVERY_ORDERS in the BAKERY_DB database using the following commands:

```
use role SYSADMIN;
use database BAKERY_DB;
create schema DELIVERY_ORDERS;
use schema DELIVERY_ORDERS;
```

Then, we will create an external stage named SPEEDY_STAGE using the SPEEDY_INTEGRATION storage integration. We will provide the location of the files in the storage container we received from the food delivery service (the name of the storage account is speedyorders001, and the name of the container is speedyservicefiles). Because we know that the files are in JSON format, we will provide the FILE_FORMAT parameter as type = json. We can execute the following command to create the external stage:

```
create stage SPEEDY_STAGE
  storage_integration = SPEEDY_INTEGRATION
```

```
  url = 'azure://speedyorders001.blob.core.windows.net/speedyservicefiles/'
  file_format = (type = json);
```

To view the contents of the external stage, we can execute the following command:

```
list @SPEEDY_STAGE;
```

The output from this command will show any files that the food delivery service has already uploaded to the blob storage container. If you are following along, the `LIST` command will show the files you uploaded to the storage container earlier.

We can view the data in the staged files by executing a SQL `SELECT` command:

```
select $1 from @SPEEDY_STAGE;
```

The output of this command shows a variant column with data in JSON format stored in files in external storage. The following listing shows an example of the JSON data structure.

Listing 5.1 JSON data that contains the food delivery service's order information

```
{
    "Order id": "12345",
    "Order datetime": "2023-09-04 12:30:00",
    "Items": [
        {
            "Item": "Croissant",
            "Quantity": 2
        },
        {
            "Item": "Bagel",
            "Quantity": 3
        }
    ]
}
```

As we can see in listing 5.1, there are three JSON key–value pairs at the highest level of the hierarchy with keys named "Order id," "Order datetime," and "Items." The value of the "Items" key is a list of key–value pairs with keys named "Item" and "Quantity." Figure 5.2 shows a graphical representation of the JSON structure.

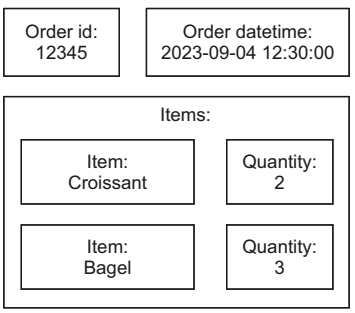


Figure 5.2 Graphical representation of the food delivery service's orders JSON structure. There are three key–value pairs at the highest level of the hierarchy with keys named "Order id," "Order datetime," and "Items." The value of the "Items" key is a list of key–value pairs with keys named "Item" and "Quantity."

We will extract the ORDER_ID and ORDER_DATETIME columns from the JSON structure as individual columns but leave the ITEMS column as a variant data type without parsing. The command that will do that as well as add the metadata$filename metadata column and the current timestamp is shown in the following listing.

Listing 5.2 Extracting values from keys at the first level of the hierarchy

```
select
  $1:"Order id",
  $1:"Order datetime",
  $1:"Items",
  metadata$filename,
  current_timestamp()
from @SPEEDY_STAGE;
```

To store the data from the JSON files, we will create a raw staging table named SPEEDY_ORDERS_RAW_STG using the following command:

```
create table SPEEDY_ORDERS_RAW_STG (
  order_id varchar,
  order_datetime timestamp,
  items variant,
  source_file_name varchar,
  load_ts timestamp
);
```

Preparing the external stage and the target table has been the same as preparing for bulk loading up until now. At this point, we can use the COPY command to bulk-load data from the food delivery service's order files into the staging table. But because we want to load files continuously, as soon as they arrive in cloud storage, we will proceed with building a Snowpipe.

5.3 Configuring Snowpipe with cloud messaging

Snowpipe is a serverless data ingestion service that automates loading data from files hosted in external storage locations into Snowflake. Unlike bulk loading, it uses Snowflake-provided compute resources instead of a user-defined virtual warehouse. Snowflake chooses the required compute resources for Snowpipe ingestion behind the scenes without user intervention.

A Snowpipe is a Snowflake pipe object that contains a COPY statement used to load data from the external stage into a relational table. Like bulk loading, the data in the files can be in various formats, including CSV or semistructured.

To ingest data, Snowpipe can be notified that new files are available in cloud storage in different ways:

- By integrating with cloud messaging, which is described in more detail in this chapter

- By calling Snowpipe `REST` endpoints, which is not covered in this chapter and requires custom coding. More information is available in the Snowflake documentation (see the following sidebar).

> **Calling Snowpipe REST endpoints**
>
> Instead of cloud messaging, an application that calls the Snowpipe `REST` API can notify Snowpipe that new files are available in cloud storage. The application calls a public `REST` endpoint with the name of the Snowpipe object and a list of filenames. These files are queued for loading from the stage into Snowflake tables using the `COPY` statement and any other parameters defined in the Snowpipe object.
>
> More information about calling Snowpipe REST endpoints to load data, including sample code in Java and Python, is available in the Snowflake documentation at https://mng.bz/QVJe.

Cloud messaging, also called event notifications, informs Snowpipe of the arrival of new data files in cloud storage that it must ingest. The event notifications are stored in a queue and are used by Snowpipe to identify new files.

> **NOTE** Event notifications contain only the metadata, such as the names of files that arrived in cloud storage. The files remain in the cloud storage container until Snowpipe ingests them.

The following cloud storage event notifications are supported for loading data continuously into Snowflake tables:

- Amazon S3: SQS (Simple Queue Service) notifications for an S3 bucket
- Google Cloud Storage: Pub/Sub messages for Google Cloud Storage events
- Microsoft Azure blob storage: event grid messages for blob storage events

In this chapter, we will work with Microsoft Azure event grid messages.

> **Configuring cloud messaging in Amazon S3 or Google Cloud Storage**
>
> The Snowflake documentation provides details about setting up cloud messaging in Amazon S3 and Google Cloud Storage at https://mng.bz/yoOe (Amazon S3) and https://mng.bz/EOzj (Google Cloud Storage).

5.3.1 *Configuring event grid messages for blob storage events*

To configure event grid messaging in Microsoft Azure, we must configure the following resources:

- A storage account for the storage queue because the storage queue must reside in a storage account. We can create a new storage account or use the same account we use to store data files. We will use the `speedyorders001` account

already created for the data files and the storage queue to keep the exercises in this chapter simple.

- A storage queue to store notifications in the form of a queue.
- An event grid subscription that defines which events on a specific topic users want to be notified about. When defining the event grid subscription, we will define the event grid system topic that will capture blob storage events, such as new files appearing in the storage container.

The following sections explain in more detail the steps required to configure event grid messaging using your Microsoft Azure portal.

Enabling the Event Grid Resource Provider

Before using the Microsoft Azure event grid, you must check whether the Event Grid Resource Provider is enabled in your Microsoft Azure subscription. If it is not enabled, you must enable it by following these steps:

1. Navigate to your subscription.
2. Choose the Resource providers option.
3. In the list of resources that appears, find "Microsoft.EventGrid."
4. If the status is Registered, you don't have to do anything else. Otherwise, select the "Microsoft.EventGrid" line, and click the Register option at the top of the page to enable the event grid resource provider.

Creating a storage queue

A storage queue stores a set of messages—in this case, event messages from the event grid. We will create it in the `speedyorders001` storage account. To create a storage queue, perform the following steps:

1. Navigate to the storage account.
2. Choose the Queue option.
3. Create a new Queue. You can name the queue however you wish, but if you are following along, we will name it `speedyordersqueue` for this exercise.
4. Please note the queue URL because we will need it later. In our example, the URL is https://speedyorders001.queue.core.windows.net/speedyordersqueue.

Creating the event grid subscription

Subscribing to a topic informs the event grid of which events to track. To create the event grid subscription, follow these steps:

1. Navigate to the storage account.
2. Choose the Events option.
3. Create a new Event Subscription. You can name the event subscription however you wish, but if you are following along, we will name it `speedysubscription`.
4. Add a name for the event grid system topic. We are naming it `speedyordersevents`. A system topic in the event grid represents one or more events published by Azure services. In this example, when a blob is uploaded to the storage account, the Azure Storage service publishes a "Blob Created" event

to the system topic in the event grid, which then forwards the event to the topic's subscribers (in our case, the `speedyordersqueue` queue) that receive and process the event.

5. Expand the "Filter to Event Types" option and ensure that only the "Blob Created" event type is selected. Only these events trigger Snowpipe to load files because only adding new objects to blob storage event types is supported.
6. In the Endpoint Types drop-down list, choose Storage Queues. Then click the "Configure an endpoint" option. Enter the name of the storage container where the queue was created—in our case, `speedyorders001`. Choose the "Select existing queue" option, and choose the `speedyordersqueue` queue in the drop-down list.

A graphical representation of these resources is shown in figure 5.3.

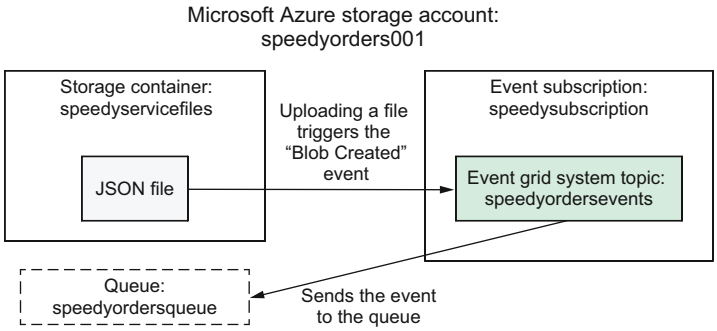


Figure 5.3 Graphical representation of Microsoft Azure resources required to configure event grid messages. A storage account contains a storage container with data files, a message queue, and an event subscription subscribing to an event grid system topic. Loading a new file to the storage container triggers the "Blob Created" event. The event grid system topic sends it to the message queue.

Once these resources are ready, we can create a notification integration in Snowflake. It will enable Snowpipe to load data automatically when Microsoft Azure event grid notifications for an Azure container are triggered.

5.3.2 Creating a notification integration

A *notification integration* is a Snowflake object that provides an interface between Snowflake and a third-party cloud message queuing service such as Azure event grid.

The food delivery service must provide the following information related to their storage account, which we will use when creating a notification integration:

- The Azure Tenant ID
- The storage queue URL

In our example, the Azure Tenant ID is `1234abcd-xxx-56efgh78` (this is a fictional Tenant ID for illustrative purposes), and the storage queue URL that we noted earlier

when creating the storage queue is `https://speedyorders001.queue.core.windows.net/ speedyordersqueue`.

Using this information, we can create the `SPEEDY_QUEUE_INTEGRATION` notification integration using the following command:

```
use role ACCOUNTADMIN;
CREATE NOTIFICATION INTEGRATION SPEEDY_QUEUE_INTEGRATION
ENABLED = true
TYPE = QUEUE
NOTIFICATION_PROVIDER = AZURE_STORAGE_QUEUE
AZURE_STORAGE_QUEUE_PRIMARY_URI =
     'https://speedyorders001.queue.core.windows.net/speedyordersqueue'
AZURE_TENANT_ID = '1234abcd-xxx-56efgh78';
```

Just like when creating the storage integration, we will execute the `DESCRIBE INTEGRATION` command and take note of the `AZURE_CONSENT_URL` and `AZURE_MULTI_TENANT_APP_NAME` properties using the following command:

```
describe notification integration SPEEDY_QUEUE_INTEGRATION;
```

Then we will work with the food delivery service's Azure administrator, who will accept the Snowflake service principal and grant a role assignment for the "Storage Queue Data Contributor" role.

> **TIP** Detailed instructions on the steps required to accept the Snowflake service principal and grant the role assignment in Azure are available in the Snowflake documentation at https://mng.bz/XVJM.

Once the notification integration is created and the authorization steps are completed in Azure, we can use it to create a Snowpipe. We will grant usage on the notification integration object to the role that will create the Snowpipe and build the pipeline. Usually, we use custom roles to perform data engineering activities, but since we are working with an exercise and don't have any custom roles created, we will use the `SYSADMIN` role to create the Snowpipe. Therefore, we will grant usage on the notification integration object to the `SYSADMIN` role by executing the following command:

```
grant usage on integration SPEEDY_QUEUE_INTEGRATION to role SYSADMIN;
```

5.3.3 Creating a pipe object

Now we are ready to create a Snowpipe. The name of the pipe object is `SPEEDY_PIPE`; it uses the `SPEEDY_QUEUE_INTEGRATION` notification integration and executes a `COPY` command that copies data from the staged files in the `SPEEDY_STAGE` external stage into the `SPEEDY_ORDERS_RAW_STG` staging table using the query from listing 5.2. The following command creates the described Snowpipe:

```
use role SYSADMIN;
create pipe SPEEDY_PIPE
  auto_ingest = true
  integration = 'SPEEDY_QUEUE_INTEGRATION'
  as
  copy into SPEEDY_ORDERS_RAW_STG
  from (
    select        <-------- The query from listing 5.2
      $1:"Order id",
      $1:"Order datetime",
      $1:"Items",
      metadata$filename,
      current_timestamp()
    from @SPEEDY_STAGE
  );
```

Once the Snowpipe is created, it will execute the COPY command in the pipe object's definition whenever an event representing new files appearing in blob storage is added to the message queue.

Because we just created the pipe, it will only start ingesting data from new files as they arrive, but it will not detect the existing files already in the cloud storage. To ingest data from files that were in cloud storage before we created the Snowpipe, we can modify the pipe with the REFRESH keyword.

> **NOTE** Modifying the pipe with the REFRESH command detects files that were uploaded to cloud storage within the last 7 days. Files that were uploaded earlier are not detected. Data from older files can be ingested into Snowflake tables as bulk loading by executing the COPY command, as explained in chapter 2.

When we modify the pipe with the REFRESH keyword, data from files that we uploaded to the storage container earlier (`Orders_2023-09-04_12-30-00_12345.json`, `Orders_2023-09-04_12-30-00_12346.json`, and `Orders_2023-09-04_12-45-00_12347.json`) should be ingested into the Snowflake table. We can execute the following command:

```
alter pipe SPEEDY_PIPE refresh;
```

After executing the previous command, we want to verify that data from the files in cloud storage was indeed loaded to the SPEEDY_ORDERS_RAW_STG table. We can do this by querying the table with the following SQL command:

```
select *
from SPEEDY_ORDERS_RAW_STG;
```

The output from this command should return three rows of data, as shown in table 5.1.

> **TIP** The data may take a few minutes to appear in the staging table. If the query returns zero rows when you first execute it, wait a minute or two and try again.

Table 5.1 Data in the staging table after refreshing the pipe object (first four columns shown)

Order id	Order datetime	Orders	Source filename
12345	2023-09-04 12:30:00.000	[{"Item":"Croissant","Quantity":2}, {"Item":"Bagel","Quantity":3}]	Orders_2023-09-04_ 12-30-00_12345.json
12346	2023-09-04 12:30:00.000	[{"Item": "Croissant", "Quantity": 5}]	Orders_2023-09-04_ 12-30-00_12346.json
12347	2023-09-04 12:45:00.000	[{"Item":"Muffin", "Quantity":12}]	Orders_2023-09-04_ 12-45-00_12347.json

When monitoring and troubleshooting pipe objects, you can use the `system$pipe_status` function to check the status of the pipe by executing the following command:

```
select system$pipe_status('SPEEDY_PIPE');
```

The output of this command displays information such as the pipe's execution state, the number of pending files, the timestamp of the last ingestion, the name of the last file that was ingested, and more. This will help you better understand the pipe's status.

Another useful function is the `copy_history` table function. This function displays the loading history for a table resulting from `COPY` commands executed by both bulk loading and Snowpipe. We can execute the following command to view the loading history for the `SPEEDY_ORDERS_RAW_STG` table within the last hour:

```
select *
from table(information_schema.copy_history(
  table_name => 'SPEEDY_ORDERS_RAW_STG',
  start_time => dateadd(hours, -1, current_timestamp())));
```

The output of this command displays information such as the filename, the external stage location, the last load time, the row count, the file size, the error messages, if any, and the pipe information in cases when a Snowpipe executes the `COPY` command.

> **TIP** Snowpipe keeps track of ingested files and doesn't ingest data from the same files again to avoid data duplication. This tracking applies to both the pipe object and the target table. If you are practicing creating and recreating the pipe object, you must recreate the target table to load data from the same files again. Otherwise, even if you recreate the pipe object but not the target table, Snowpipe will not ingest data from the staged files because it assumes that the data is already loaded in the target table.

5.3.4 Ingesting data continuously

Now that we have the pipe object set up and the initial data ingested, we can proceed with the interesting part of ingesting data continuously. We will perform the following scenario:

1. Select data from the `SPEEDY_ORDERS_RAW_STG` staging table to establish how many files have already been loaded into the table.
2. Upload a new file to cloud storage.
3. Wait a minute or two.
4. Select data from the `SPEEDY_ORDERS_RAW_STG` staging table again to verify that Snowpipe ingested the data from the new file in cloud storage and is available in the table.

Go to your Microsoft Azure portal, navigate to the `speedyservicefiles` container, and upload another sample file named Orders_2023-09-04_12-45-00_12348.json from the GitHub repository in the Chapter_05 folder to the storage container.

After a minute or two, select data from the staging table:

```
select *
from SPEEDY_ORDERS_RAW_STG;
```

The data from your uploaded file should appear in the staging table. This confirms that the Snowpipe is continuously loading data and is using cloud messaging to be notified when new files arrive.

Be patient if new data doesn't appear immediately in the staging table; just wait a minute and execute the query again. You can repeat the scenario with another file: upload the file to the cloud storage container, and then check that the data has been ingested in the staging table.

> **TIP** If you are following along and you repeat the scenario of uploading a new file in the cloud storage container several times and run out of files in the GitHub repository, you can create additional files. Copy one of the existing files and rename it to a new, unique name. Be sure that you keep the JSON structure within the file; otherwise, the ingestion step will fail.

5.3.5 Flattening the JSON structure to relational format

Now that data from files in cloud storage is loading continuously, let's examine the JSON structure of the order data in more detail and flatten it to a relational structure. As shown in listing 5.1, there are three JSON key–value pairs at the highest level of the hierarchy with keys named "Order id," "Order datetime," and "Items." The value of the "Items" key is a list of key–value pairs with keys named "Baked good type" and "Quantity."

In the `SPEEDY_ORDERS_RAW_STG` staging table, we extract the values of the three keys at the highest level of the hierarchy into individual columns. We will now flatten the value of the "Items" key into relational columns.

We will use the `FLATTEN` function with the `LATERAL` modifier to flatten the JSON structure into relational columns, as we did in chapter 4 when we flattened the hotel's orders. We can extract the values from the second-level keys named "Item" and "Quantity" from the value of the "Items" first-level key using the query in the following listing.

Listing 5.3 Flattening values from keys at the second level of the hierarchy

```
select
  order_id,
  order_datetime,
  value:"Item"::varchar as baked_good_type,
  value:"Quantity"::number as quantity
from SPEEDY_ORDERS_RAW_STG,
lateral flatten (input => items);
```

The first few rows of the output from this query should look like the data shown in table 5.2.

Table 5.2 Flattened data from the JSON structure

Order id	Order datetime	Baked good type	Quantity
12345	2023-09-04 12:30:00.000	Croissant	2
12345	2023-09-04 12:30:00.000	Bagel	3
12346	2023-09-04 12:30:00.000	Croissant	5
12347	2023-09-04 12:45:00.000	Muffin	12
12348	2023-09-04 12:45:00.000	Muffin	2

The data in the staging table will grow as new orders from the food delivery service are added. This could lead to decreased query performance when executing the query from listing 5.3 that flattens the JSON structure. One way to improve query performance is to materialize the query's results into a table. The bakery data engineers considered different approaches to materializing the data, including

- Creating a view with the flattened data, creating a target table, and merging data from the view into the target table. This approach is used when ingesting order data from the hotel's cloud storage, as described in chapter 4. The hotel's order information is required daily, so a batch-loading schedule is sufficient. Batch scheduling does not work for the food delivery service because the bakery continuously requires order data.
- Creating a materialized view, as described in chapter 3. This could be an option if the query that flattens the data isn't too complex and doesn't select data from more than one table.
- Creating a dynamic table for more flexibility in the types of queries that can be used compared to materialized views and more control over compute resources that will be used in table maintenance.

The bakery data engineers decided to build a dynamic table to materialize the flattened data from the `SPEEDY_ORDERS_RAW_STG` staging table. The next section describes dynamic tables in more detail.

5.4 Transforming data with dynamic tables

Snowflake defines dynamic tables as the building blocks of declarative data transformation pipelines. They enable automated data transformation, which the bakery requires to process data received from the food delivery service.

Instead of creating a target table and inserting or merging transformed data into the table, you can create dynamic tables to materialize the results of a query specified in the dynamic table definition. Data engineers don't have to schedule or manage tasks that populate data in the dynamic table because Snowflake handles the maintenance of a dynamic table in the background.

While dynamic tables have some restrictions, such as not allowing the usage of stored procedures, nondeterministic functions, or external functions, they have fewer restrictions than materialized views and apply to many different use cases.

Let's proceed by creating a dynamic table that flattens and materializes the data in the `SPEEDY_ORDERS_RAW_STG` staging table. In addition to providing the dynamic table name and the query that will be materialized, we have to specify two additional parameters when creating a dynamic table:

- `TARGET_LAG`—This defines the maximum amount of time that the data in the dynamic table lags behind the base tables. For example, if the target lag is specified as one minute, the data in the dynamic table should lag by no more than 1 minute.
- `WAREHOUSE`—This specifies the name of the virtual warehouse that will be used for refreshing the dynamic table.

To create a dynamic table named `SPEEDY_ORDERS` that should lag behind the source data by no more than 1 minute, uses the `BAKERY_WH` virtual warehouse, and materializes data from the query in listing 5.3, we can execute the following command:

```
create dynamic table SPEEDY_ORDERS
  target_lag = '1 minute'
  warehouse = BAKERY_WH
  as
  select        <-- The query from listing 5.3
  order_id,
  order_datetime,
  value:"Item"::varchar as baked_good_type,
  value:"Quantity"::number as quantity,
  source_file_name,
  load_ts
from SPEEDY_ORDERS_RAW_STG,
lateral flatten (input => items);
```

Initially, the dynamic table materializes the data from the query that defines it. When we select data from the dynamic table, we should see the same data as in the results from the query in listing 5.3.

Let's see what happens to the dynamic table when new data arrives in cloud storage. We will perform the following scenario:

1. Select data from the SPEEDY_ORDERS dynamic table to establish how many files have already been loaded into the table.
2. Upload a new file to cloud storage.
3. Wait a minute or two.
4. Select data from the SPEEDY_ORDERS dynamic table again to verify that the data from the new file in cloud storage is available in the table.

Go to your Microsoft Azure portal, navigate to the speedyservicefiles container, and upload another sample file named Orders_2023-09-04_13-00-00_12349.json from the GitHub repository in the Chapter_05 folder to the storage container. After a minute or two, select data from the dynamic table.

Listing 5.4 Viewing order data from the dynamic table

```
select *
from SPEEDY_ORDERS
order by order_datetime desc;
```

The data from your uploaded file should appear in the dynamic table. This confirms that the Snowpipe is continuously loading data and that the dynamic table is materializing the data within the defined target lag of 1 minute.

Be patient if new data doesn't appear in the dynamic table immediately: wait a minute, and execute the query again. You can repeat the scenario with another file: upload the file to the cloud storage container, and then check that the data appears in the staging table.

By executing the query from listing 5.4, which orders the results by the order date and time in descending order, the bakery employees can see the newest order information at the top of the query results.

Like pipe objects, we can monitor and troubleshoot dynamic tables' execution in the Snowsight user interface or by executing Snowflake built-in functions, such as dynamic_table_refresh_history and dynamic_table_graph_history.

To view the dynamic table refresh history, we can execute the following command:

```
select *
from table(information_schema.dynamic_table_refresh_history())
order by refresh_start_time desc;
```

The output of this command lists details about each dynamic table execution according to the target lag frequency. For example, the SPEEDY_ORDERS dynamic table is created with a lag frequency of 1 minute. Therefore, the refresh history shows a row each minute when the dynamic table is scheduled to refresh. Even if there is no new data in the source tables and the dynamic table has nothing to refresh, a record indicates that

the dynamic table attempted a refresh. The refresh history output is beneficial when troubleshooting failed refreshes and investigating errors.

You can do much more with dynamic tables, such as setting up a pipeline by creating a dynamic table that queries other dynamic tables. You can also define dynamic tables that use SQL statements with join conditions between multiple tables.

NOTE For more information about dynamic tables, refer to the Snowflake documentation at https://mng.bz/yoEE.

Snowpipe streaming and the Kafka connector

Continuous data ingestion with Snowpipe and bulk ingestion both work with data files. However, not all data is available in files. Sometimes data is produced in rows in streaming mode—for example, from Apache Kafka topics. When we want to ingest streaming data, we can use the Snowpipe Streaming API with a custom Java application that automatically loads data streams into Snowflake as the data becomes available.

For more information about Snowpipe Streaming, please refer to the Snowflake documentation at https://mng.bz/M1Jm.

Snowflake also offers the Kafka connector, a framework designed to read data from Kafka topics and write the data into Snowflake tables.

For more information about the Kafka connector, please refer to the Snowflake documentation at https://mng.bz/NBD7. For general information about Kafka streaming, a good resource is the book *Streaming Data Pipelines with Kafka* by Stefan Sprenger, available at https://mng.bz/aV5Y.

Summary

- Bulk data loading refers to regularly scheduled data pipelines that ingest data into Snowflake tables. Pipelines can be scheduled to run within shorter intervals—for example, every hour or every few minutes. Such frequent scheduling of data pipelines is referred to as micro-batch.
- As an alternative to scheduling data pipelines in micro-batch, Snowflake offers the Snowpipe feature, which is designed to load data from files as soon as they are available in a stage.
- A Snowpipe is a Snowflake object that contains a `COPY` statement used to load data from the external stage into a relational table. Like bulk loading, the data in the files can be in various formats, including CSV or semistructured.
- Snowpipe uses compute resources provided by Snowflake instead of a user-defined virtual warehouse, as with bulk loading. Snowflake chooses the required compute resources for Snowpipe ingestion behind the scenes, without user intervention.
- To ingest data, Snowpipe can be notified that new files are available in cloud storage by integrating with cloud messaging, also known as event notifications.

An event notification informs Snowpipe of the arrival of new files in cloud storage that must be ingested. The event notifications are stored in a queue.

- Event grid messaging in Microsoft Azure consists of a storage queue for storing notifications, an event grid subscription for defining events, and an event grid system topic for capturing blob storage events, such as new files appearing in the storage container.
- A notification integration is a Snowflake object that provides an interface between Snowflake and a third-party cloud message queuing service such as Azure event grid.
- A Snowpipe executes the `COPY` command in the pipe object's definition whenever an event representing new files appearing in blob storage is added to the message queue. The pipe can be modified with the `REFRESH` keyword to ingest data from files that were already in cloud storage before the Snowpipe was created.
- The `system$pipe_status` function can be used to check the pipe's status. The `copy_history` table function displays the loading history for a table that resulted from `COPY` commands executed by both bulk loading and Snowpipe.
- Different approaches can be used to flatten and materialize data from the staging table, such as creating a target table and merging the data, creating a materialized view, or creating a dynamic table.
- Dynamic tables are the building blocks of declarative data transformation pipelines. Instead of creating a target table and inserting or merging transformed data into it, dynamic tables can materialize the results of a query specified in the dynamic table definition.
- Data engineers don't have to schedule or manage tasks that populate data in dynamic tables because Snowflake handles the maintenance of dynamic tables in the background.
- In addition to providing the dynamic table name and the query that will be materialized, two additional parameters are required when creating a dynamic table. These are the target lag, which defines the maximum amount of time that the data in the dynamic table lags behind the queried tables, and the virtual warehouse, which will be used to refresh the dynamic table.

6 Executing code natively with Snowpark

This chapter covers

- The Snowpark architecture
- Creating functions and stored procedures in Python worksheets
- Using the SQL API from a local development environment
- Transforming data with data frames
- Deploying functions and stored procedures

Data engineers spend significant time developing data ingestion, transformation, and presentation applications. In addition to using SQL, Snowflake's built-in querying language, data engineers can build applications in one of the supported programming languages, including Python, Java, and Scala, that run natively in Snowflake. Snowpark makes this possible, as described in more detail in this chapter.

Snowpark is an umbrella term encompassing features on the server and the client sides. These features are as follows:

- On the server side, Snowpark executes code natively in Python, Java, and Scala. Data engineers don't have to extract data from Snowflake and save it

in their local development environment to work with the data. Instead, they can write code that executes inside the Snowflake processing engine.

- On the client side, Snowpark provides API libraries for each supported programming language. These libraries contain functionality to connect to Snowflake, query data in Snowflake tables, transform the data, and deploy functions and stored procedures to the Snowflake server.

To illustrate the examples in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries and delivers these baked goods to small businesses, such as grocery stores, coffee shops, and restaurants in the neighborhood.

Since the bakery has no online ordering system, customers order baked goods via email. A bakery employee reads the emails and stores order information in a CSV file on the local file system. The employee then uploads the CSV file into an internal Snowflake stage. The bakery data engineers built a data pipeline that ingests data from the files in the internal stage into Snowflake tables. The pipeline also transforms the data so that the bakery manager knows the quantities of baked goods ordered in the future and can plan the production process accordingly. This data pipeline is described in detail in chapter 2.

Now the bakery wants to use Snowpark to streamline data pipeline development in an integrated development environment maintained in a shared code repository. It wants to replace manual steps that must be performed by bakery employees, such as uploading a CSV file to an internal stage, with an automated pipeline. More automation will free the employees to focus on other tasks and reduce the chance of error.

The bakery also looks ahead because it wants to utilize its data for business insights in the future. It wants to enrich its data with information that originates outside of the organization. It also wants to prepare the building blocks required for the enhancements that await in the future, such as augmenting the data with outputs from large language models and using generative AI to save time when interpreting customer order emails. These are described in more detail in chapter 7, but to get ready, we will build the data pipelines and transformations in Snowpark in this chapter.

> **NOTE** All code and sample data files for this chapter are available in the accompanying GitHub repository in the Chapter_06 folder at https://mng.bz/DpGa. The SQL and Python code is stored in multiple files whose names start with Chapter_06_Part*, where * indicates a sequence number and additional information means the file's content. Please follow the exercises by reading the files sequentially.

Python basics

This chapter assumes that you have a basic knowledge of the Python programming language. If not, you can find many Python tutorials by searching the internet. If you prefer learning from a book, a good resource is *The Quick Python Book, Third Edition,* by Naomi Ceder: https://mng.bz/lrKB.

6.1 Introducing Snowpark

The Snowpark architecture supports executing code natively in Python, Java, and Scala on the server side and API libraries for each of these languages on the client side. In this chapter, we will work with Python.

Working with Snowpark API libraries in Java or Scala

In addition to Python, you can use Snowpark libraries in Java or Scala. For more information about the APIs supported in these programming languages, refer to the Snowflake documentation at https://mng.bz/BgOr (Java) and https://mng.bz/dZgv (Scala).

The data structure used in Snowpark is the *data frame*, which represents a relational dataset. The Snowpark API library provides methods for querying and transforming data frames, such as selecting columns, filtering, ordering, grouping, combining, etc. When working with data frames, the queries and transformations are internally translated into SQL statements and sent to the Snowflake server-side SQL engine for execution.

When a data engineer creates functions or stored procedures in Snowpark, the code that defines these objects is serialized and sent to the Snowflake server. It executes in a secure sandbox, adhering to the defined security access policies in Snowflake. The server-side Python code can also use built-in Python packages if needed. The high-level Snowpark Python architecture is illustrated in figure 6.1.

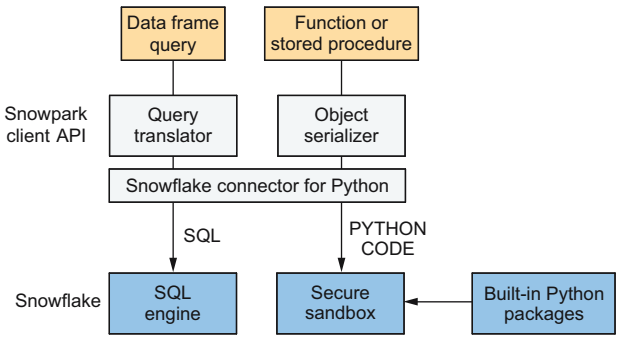


Figure 6.1 The Snowpark Python high-level architecture consists of the client API, which includes methods for querying and transforming data frames and defining functions and stored procedures. Data frame queries are internally translated to SQL and sent to the Snowflake server-side SQL engine for execution. Code that defines functions and stored procedures is serialized and sent to the Snowflake server, where it executes in a secure sandbox, using Python built-in packages if needed.

The Snowpark API also supports querying data from Snowflake tables using SQL statements. Developers can choose how to work with data according to their preferences and prior experience by executing SQL queries or applying data frame methods.

Many open-source Python packages provided by Anaconda are available for use in Snowflake and are maintained in the Snowflake conda channel at https://repo.anaconda.com/pkgs/snowflake/. You can reference these packages in the Snowpark Python code and use them like when coding stand-alone Python applications.

> **NOTE** Before you can start using the packages provided by Anaconda in your Snowflake account, you must accept the third-party terms. Using the `ORGADMIN` role, you must navigate to the Admin menu in Snowsight and enable Anaconda packages under the Billing & Terms option. Detailed instructions on accepting third-party terms are available in the Snowflake documentation at https://mng.bz/r1Ne.

6.2 Creating a Snowpark procedure in a worksheet

To get acquainted with working in Snowpark Python, we will start by building a date dimension containing a flag to indicate whether each date in the dimension is a holiday. The bakery employees will use this dimension by joining with order information to help them prepare for different holiday demand levels. For example, some customer shops may be closed for the holidays, reducing demand, but other customers, such as the hotel, may have a higher demand during the holidays.

We will build the date dimension in Snowpark Python, starting with a stored procedure that takes a date as the input parameter and returns a flag indicating whether the date is a holiday. For this purpose, we can use the Python `holidays` library.

> **TIP** The documentation for the `holidays` library is available at https://python-holidays.readthedocs.io/en/latest/.

The `holidays` library requires a country as a parameter since holidays vary in each country. We will use "US" as the country for all exercises in this chapter. If you are following along, you can use a different country by modifying the code.

To keep all the code we write in Snowpark in one place, we will continue to use the `BAKERY_DB` database that we created in chapter 2. We will create a new schema named `SNOWPARK` using the following commands:

```
use role SYSADMIN;
use database BAKERY_DB;
create schema SNOWPARK;
use schema SNOWPARK;
```

Now we can create a new worksheet in the Snowsight user interface, but instead of an "SQL Worksheet," we will choose "Python Worksheet." In this sheet, we will write a Python function named `IS_HOLIDAY` that will execute in Snowflake and return `True` or `False` depending on whether a given date is a holiday.

TIP Whenever you create a new worksheet in Snowsight, be it an SQL or a Python worksheet, remember to give it a meaningful name by renaming it from the default, a timestamp. A thoughtful name will help you find it later. When you have many worksheets, you can also organize them in folders.

In the new Python worksheet, we will set up the environment by following these steps:

1. Use the `SYSADMIN` role and the `BAKERY_WH` warehouse (we created this warehouse in chapter 2).
2. Set the worksheet context by choosing `BAKERY_DB` as the database and `SNOWPARK` as the schema.
3. Under Settings, change the handler to `IS_HOLIDAY` and the return type to `String`, as shown in figure 6.2.

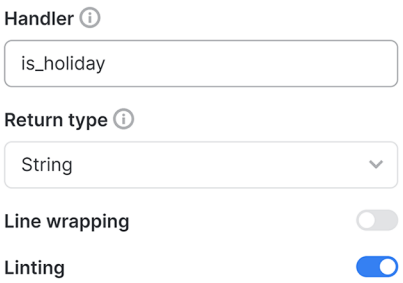


Figure 6.2 The Settings options in a Python worksheet where the handler is set to `is_holiday` and the return type is set to `String`

NOTE A Python function can return a table, variant, or string data type but not Boolean. For the function to return `True` or `False`, the return value must be cast as a string.

4. Under Packages, choose `holidays`, as shown in figure 6.3.

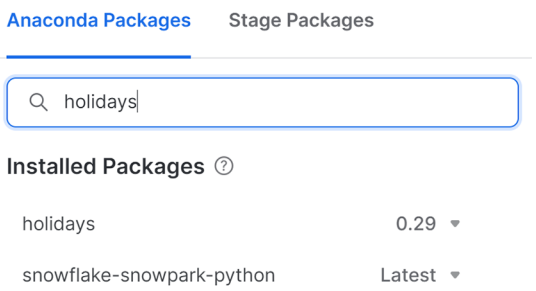


Figure 6.3 The Packages options in a Python worksheet where we search for and select the `holidays` package

TIP Detailed instructions on writing Snowpark code in Python worksheets are available in the Snowflake documentation at https://mng.bz/V2ZW.

After setting the worksheet environment, we will edit the template Python code populated when creating the worksheet. We will import the `holidays` package using the

standard Python import keyword. For this first version, we will hard code the date as January 1, 2024, and "US" as the country. The code after modifying the template code in the Python worksheet is shown in the following listing.

Listing 6.1 Snowpark code that determines whether a date is a holiday

```
# The Snowpark package is required for Python Worksheets
import snowflake.snowpark as snowpark
# Adding the holidays package
import holidays
def is_holiday(session: snowpark.Session):          <-- The handler function
    # get a list of all holidays in the US
    all_holidays = holidays.country_holidays('US')
    # return TRUE if January 1, 2024 is a holiday, otherwise return false
    if '2024-01-01' in all_holidays:
        return True
    else:
        return False
```

We can then click the Run button to execute the code. It should return `True` because January 1, 2024, is a holiday in the United States. You can change the country and the date to different values and execute the code to see which values the code returns by varying the parameters.

The code from listing 6.1 is impractical because the date value is hard coded, and we must change it each time we run the code. To improve this behavior, we will deploy the Python code as a Snowflake stored procedure and change it to accept a date as a parameter.

We can click the Deploy button at the top of the worksheet and enter the following information in the "New Python Procedure" dialog window that appears:

- *Procedure name*—You can name the procedure however you wish, but if you are following along, we are naming it `PROC_IS_HOLIDAY`.
- *Comment*—A brief comment on what the procedure does. In our example, the comment is "The procedure returns True if the date is a holiday."
- *Replace if exists*—Enable the toggle button because it is helpful to replace the procedure if it already exists when developing, as we usually create and recreate it more than once before we get it to work correctly.
- *Handler*—The name of the Python function that the procedure expects in the body of the procedure. In our example, the handler is `IS_HOLIDAY`, the Python function we created in listing 6.1.

The "New Python Procedure" dialog window with the supplied parameters is shown in figure 6.4.

WARNING Do not click the Deploy button in the dialog window in figure 6.4. If you do, you will deploy the function without parameters.

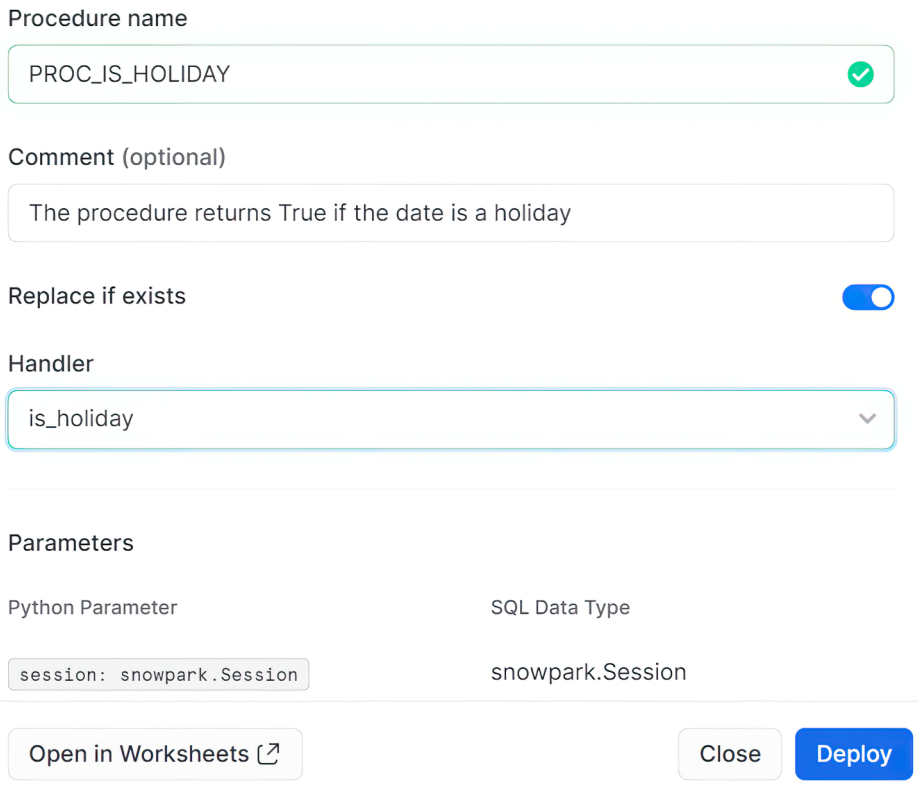


Figure 6.4 "New Python Procedure" dialog window where the Procedure name, Comment, and Handler text boxes are filled with user-supplied values

After entering the information in the dialog text boxes, click the "Open in Worksheets" button, and a new SQL worksheet will appear. The worksheet contains the code that creates the Snowflake stored procedure named `PROC_IS_HOLIDAY` using the Python code we created in listing 6.1. We will now edit this code so that the procedure accepts two parameters: `P_DATE` with a date data type and `P_COUNTRY` with a string data type.

In the first line that contains the `CREATE OR REPLACE` statement, we will add the `P_DATE` and `P_COUNTRY` parameters as follows:

```
create or replace procedure PROC_IS_HOLIDAY(p_date date, p_country string)
```

Then, we will modify the Python function `IS_HOLIDAY()` to accept the `P_DATE` and `P_COUNTRY` parameters in addition to the default `session: snowpark.Session` parameter. In the function's body, we will change the hard-coded values `US` and `2024-01-01` from earlier with the `P_DATE` and `P_COUNTRY` parameters. The updated code of the Python function is shown in the following listing.

Listing 6.2 Python function that determines whether a date is a holiday

```
def is_holiday(session: snowpark.Session, p_date, p_country):
    # get a list of all holidays in the US
    all_holidays = holidays.country_holidays(p_country)
    # return TRUE if p_date is a holiday, otherwise return false
    if p_date in all_holidays:
        return True
    else:
        return False
```

The complete code that creates the stored procedure in the worksheet is

```
create or replace procedure PROC_IS_HOLIDAY(p_date date, p_country string)
    returns String
    language python
    runtime_version = 3.8
    packages =('holidays==0.29', 'snowflake-snowpark-python==*')
    handler = 'is_holiday'
    comment = 'The procedure returns True if the date is a holiday in the
     country'
    as '
# The Snowpark package is required for Python Worksheets
import snowflake.snowpark as snowpark
# Adding the holidays package
import holidays
def is_holiday...        <-- The code from listing 6.2
';
```

Let's execute this code in the worksheet. It should create a stored procedure named `PROC_IS_HOLIDAY` in the `SNOWPARK` schema in the `BAKERY_DB` database.

To test that the procedure is returning the correct `True` and `False` values for the provided parameters, you can execute it with different values of some known holiday dates in the US and France—for example:

```
call PROC_IS_HOLIDAY('2024-01-01', 'US'); -- returns True
call PROC_IS_HOLIDAY('2024-07-04', 'US'); -- returns True
call PROC_IS_HOLIDAY('2024-07-14', 'US'); -- returns False
call PROC_IS_HOLIDAY('2024-07-14', 'FR'); -- returns True
call PROC_IS_HOLIDAY('2024-07-04', 'FR'); -- returns False
```

You can test the procedure in more detail by providing the country code of the country where you reside and some known holiday dates in your country.

With this example, we have illustrated how to create a stored procedure in Snowflake using Python as the language. We will return to this stored procedure later because we will reuse the code when we build the date dimension requiring a flag to indicate whether a date is a holiday.

Although working in Snowsight worksheets is convenient because we can quickly create stored procedures, this approach is not sustainable when many developers work on the same codebase that must support source control, versioning, branching,

and automatic deployments. To achieve that, we will learn how to use the Snowpark SQL API from a local development environment, which we can use together with source control, such as Git.

6.3 Using the SQL API from a local development environment

We will begin by setting up a local development environment, which will enable us to build client applications with Snowpark Python.

6.3.1 Installing and configuring the local development environment

To install and configure your local development environment, follow these steps:

1. Install Python locally on your computer, or use an existing Python installation if you already have one. Check the Snowflake documentation to see if you have one of the supported versions.
2. Depending on which Python version you use locally, create a virtual environment for it using `conda` or `venv`.
3. Install the `snowflake-snowpark-python` package using `conda` or `pip`, depending on which Python you use locally.

NOTE Detailed instructions for performing these steps are available in the Snowflake documentation at https://mng.bz/ZV65.

You can use Snowpark with an integrated development environment (IDE). A popular IDE is Microsoft Visual Studio Code, but you can use other development environments. Ensure that the development environment uses the supported Python version (see instructions in the documentation for your preferred development environment for associating the correct version with your development environment).

6.3.2 Creating a Snowflake session

The first step when executing Snowpark Python in your local development environment is establishing a connection to your Snowflake account. To create a session with Snowflake, follow these steps:

1. Import the Session module from the `snowflake.snowpark` package that you installed earlier when setting up your environment.
2. Create a dictionary data structure in Python where you provide the connection information, such as the Snowflake account name, your username, your password, and, optionally, the role, the virtual warehouse, the database, and the schema used in the session.
3. Call the `create()` method of the Session module to return a named session object.

The code in listing 6.3 creates a Python dictionary named `connection_parameters_dict` that specifies your Snowflake account, username, password, the `SYSADMIN` role,

the BAKERY_WH warehouse, the BAKERY_DB database, and the SNOWPARK schema. It then uses this dictionary as a parameter when creating a session object named my_session.

Listing 6.3 Creating a session with the Snowflake account

```
# import Session from the snowflake.snowpark package
from snowflake.snowpark import Session
# create a dictionary with the connection parameters
connection_parameters_dict = {
    "account": "pqrstuv-ab12345", # replace with your Snowflake account
    "user": "my_user", # replace with your username
    "password": "my_pass", # replace with your password
    "role": "SYSADMIN",
    "warehouse": "BAKERY_WH",
    "database": "BAKERY_DB",
    "schema": "SNOWPARK"
}
# create the session object
my_session = Session.builder.configs(connection_parameters_dict).create()
# close the session
my_session.close()
```

In the following exercises, you will use the session object to submit commands to Snowflake. We will not submit any commands for now, but we will adhere to good programming practices by closing the session so it doesn't remain active with Snowflake.

WARNING The code in listing 6.3 can be used for testing, but it should never be used in production environments because the credentials required to establish a Snowflake connection are visible in the code. Never save credentials in a code repository or make them available to other parties. Instead, use configuration files or secrets to provide connection credentials.

6.3.3 Providing credentials in a configuration file

Different methods can be used to provide credentials for a Snowflake connection in Snowpark Python. For example, you can store the credentials in your cloud provider secrets manager. Alternatively, you can create a configuration file that stores the credentials and read the file to retrieve the information. Another approach might be defining credentials as your operating system's environment variables.

Alternative ways of creating a session instead of a username and password

Instead of using credentials, such as the username and password, you can create a Snowpark session using single sign-on, multifactor authentication, key pair authentication, or OAuth. For more information about different ways of creating a Snowpark session, refer to the Snowflake documentation at https://mng.bz/RNrD.

In the examples in this chapter, we will use a configuration file. The configuration file can be in any structured format, such as CSV, JSON, or YML, that can be interpreted

by Python, converted into a dictionary structure, and subsequently used for the connection parameters.

Let's illustrate this example with a JSON file. We will create a file named connection_parameters.json with the following contents (be sure to replace your Snowflake account, username, and password):

```
{
    "account"                 : "ab12345",
    "user"                    : "my_user",
    "password"                : "my_pass",
    "role"                    : "SYSADMIN",
    "warehouse"               : "BAKERY_WH",
    "database"                : "BAKERY_DB",
    "schema"                  : "SNOWPARK"
}
```

We will read the contents of this file using the `json` library in Python and call the `json.load()` method to read the contents of the file. Then we will construct a Python dictionary named `connection_parameters_dict` and call the `create()` method of the Session module, just like in listing 6.3. The code that reads the configuration file and creates a session with Snowflake is shown in the following listing.

Listing 6.4 Creating a session using a credentials file

```
# import Session from the snowflake.snowpark package
from snowflake.snowpark import Session
# import the json library
import json
# read the credentials from a file
credentials = json.load(open('connection_parameters.json'))
# create a dictionary with the connection parameters
connection_parameters_dict = {
    "account": credentials["account"],
    "user": credentials["user"],
    "password": credentials["password"],
    "role": credentials["role"],
    "warehouse": credentials["warehouse"],
    "database": credentials["database"],
    "schema": credentials["schema"]  # optional
}
# create the session
my_session = Session.builder.configs(connection_parameters_dict).create()
# close the session
my_session.close()
```

As shown in the code in listing 6.4, no configuration parameters are visible, so you can safely save this code to a code repository.

WARNING Don't save the connection_parameters.json file in your code repository. You can save this file in a location other than your code project files so

it doesn't commit to the code repository, or you can add the connection_parameters.json filename to the `.gitignore` file to ensure it is ignored when committing to the code repository.

6.3.4 Querying data and executing SQL commands

Once the session object exists in your Python code, you can send SQL commands to Snowflake using the `session.sql()` method. For example, to select the current timestamp on the Snowflake server and print the output to the Python console, you can execute the following commands:

```
# select the current timestamp
ts = my_session.sql("select current_timestamp()").collect()
# print the output to the console
print(ts)
```

NOTE When executing SQL commands using Snowpark, remember that the returned data type is a data frame that is lazily evaluated. The SQL statement is executed on the Snowflake server only when you send an action, such as `collect()`, `count()`, `show()`, or `save_as_table()`.

6.4 Generating a date dimension in Snowpark Python

With our knowledge so far, we can continue our Snowpark Python learning journey by combining some of the code we developed in previous sections. To start, we will build the Python code locally on the client, using the Snowpark API to save the data to a data frame and then to a table in a Snowflake database and schema.

We will build a date dimension, including dates and flags indicating whether each date is a holiday in the country where the bakery operates. The bakery will use this dimension later, along with order information, to better plan demand around the holidays.

We will start by importing the Python packages we need for the code. We will use the following packages:

- `snowflake.snowpark`—To establish a connection to Snowflake from Snowpark Python
- `snowflake.snowpark.types`—To define the data types (date for the day in the dimension and Boolean for the flag to indicate whether the day is a holiday) as well as the `StructType` and `StructField` types that are required to specify the schema when saving the data to Snowflake
- `json`—To read the connection parameters
- `datetime`—To work with dates and time intervals in Python
- `holidays`—To determine whether a day is a holiday. Depending on which Python you use, you must install this package locally using `pip` or `conda`.

NOTE In the example in listing 6.1, where we created the `IS_HOLIDAY` Python function in a Snowsight Python worksheet, installing the `holidays` package

was unnecessary because the code was executed on the Snowflake server side, where the packages were already available. We only had to indicate that we would use this package under the Packages drop-down list in the worksheet. In this section, we are writing the code locally on the client side, meaning we must install the package locally.

The following listing shows the code that imports all packages.

Listing 6.5 Importing Python packages for building the date dimension

```
# import Session from the snowflake.snowpark package
from snowflake.snowpark import Session
# import data types from the snowflake.snowpark package
from snowflake.snowpark.types
  import StructType, StructField, DateType, BooleanType
# import json package for reading connection parameters
import json
# import date and timedelta from the datetime package for generating dates
from datetime import date, timedelta
# install the holidays package using pip or conda
# import the holidays package to determine whether a given date is a holiday
import holidays
```

The next step is to create a function in Python, like the one we built in listing 6.2, except instead of tailoring the function so that it executes in a Python worksheet, we want the function to run locally in the Python code. When running locally, the function doesn't require the session parameter. The code that uses the `holidays` package to return a flag indicating whether the date in the parameter is a holiday is shown in the following listing.

Listing 6.6 Python function that determines whether the date is a holiday

```
# define a function that returns True if p_date is a holiday in p_country
def is_holiday(p_date, p_country):
    # get a list of all holidays in p_country
    all_holidays = holidays.country_holidays(p_country)
    # return True if p_date is a holiday, otherwise return false
    if p_date in all_holidays:
        return True
    else:
        return False
```

To build a date dimension, we must generate a list of dates. We will use the Python modules `date` and `timedelta` from the `datetime` package. Eventually, the date dimension will contain historical dates starting from when the bakery first started operating up to a year or more in the future, depending on how far into the future the bakery wants to plan.

To make developing and testing the code in this example more manageable, we will start with January 1, 2023 and include five days in the dimension. Later, once we finalize and test the code so that we are confident that it is working correctly, we will rerun it to include more dates.

The next listing shows the Python code that generates a list of dates starting from a starting date as defined by the `start_dt` variable and continuing for as many days as defined by the `no_days` variable.

Listing 6.7 Generating a list of dates in Python

```
# generate a list of dates starting from start_date followed by as many days
# as defined in the no_days variable
# define the start date
start_dt = date(2023, 1, 1)
# define number of days
no_days = 5
# store consecutive dates starting from the start date in a list
dates = [(start_dt + timedelta(days=i)).isoformat()
  for i in range(no_days)]
```

To this list of dates, we will add a flag to indicate whether it is a holiday by using the `is_holiday()` function we created earlier. The output will be a list of lists, as illustrated in the code in the following listing.

Listing 6.8 Combining the date and the output of the `is_holiday()` function

```
# create a list of lists that combines the list of dates
# with the output of the is_holiday() function
holiday_flags = [[d, is_holiday(d, 'US')] for d in dates]
```

At this point, we can print the `holiday_flags` list of lists to the console to check that the data is as expected using the following code:

```
print(holiday_flags)
```

There should be five dates starting from January 1, 2023, each indicating whether the date is a holiday. The following snippet shows how the output looks (both January 1 and January 2 were holidays in the United States in 2023, while the remaining dates were not holidays):

```
[['2023-01-01', True], ['2023-01-02', True], ['2023-01-03', False],
 ['2023-01-04', False], ['2023-01-05', False]]
```

6.5 Working with data frames

Now that we have the data for the date dimension generated in a Python list of lists, we want to save it to a table in Snowflake. To do this, we will use the Snowpark API and the data frame data structure.

A data frame is Snowpark's core abstraction. It is like a table in a relational database environment. A data frame represents a data set, and Snowpark provides methods to operate on it.

The data frame methods are executed lazily on the server. Lazy execution means that you can delay executing the methods until late in the data pipeline by batching several operations to run at the same time. Execution happens when you perform an action, such as `collect()`, `count()`, `show()`, or `save_as_table()`. By batching methods, less data is transferred between the server and the client, which generally results in better performance.

To work with data frames, you need a connection to Snowflake. We will create a connection object named `my_session`, like in listing 6.4, and use the credentials file we created earlier.

Next we will construct a data frame named `df` from the `holiday_flags` list of lists we created earlier. We will use the `create_dataframe()` method, to which we will provide two parameters:

- The `holiday_flags` list of lists.
- The schema definition, including column names and data types. Since the dimension has two columns, we will provide a `StructField` for each. The first column will be called `day` and will be of data type date. The second column will be called `holiday_flg` and will be of the Boolean data type.

The code that constructs the data frame is shown in the following listing.

Listing 6.9 Constructing a data frame and defining the data structure

```
# create a data frame from the holiday_flags list of lists
# and define the schema as two columns:
# - column named "day" with data type DateType
# - column named "holiday_flg" with data type BooleanType
df = my_session.create_dataframe(
    holiday_flags,
    schema = StructType(
        [StructField("day", DateType()),
         StructField("holiday_flg", BooleanType())])
    )
```

Once the data frame is constructed, we can write it to a Snowflake table. But before we do that, we can print the output to the console to ensure the data is as expected. Remember that the data frame is lazily evaluated, so just printing the `df` data frame using the `print(df)` command will not print anything. You must also add an action, such as `collect()`:

```
print(df.collect())
```

The output of the print command should look like the following:

```
[Row(DAY=datetime.date(2023, 1, 1), HOLIDAY_FLG=True),
 Row(DAY=datetime.date(2023, 1, 2), HOLIDAY_FLG=True),
 Row(DAY=datetime.date(2023, 1, 3), HOLIDAY_FLG=False),
 Row(DAY=datetime.date(2023, 1, 4), HOLIDAY_FLG=False),
 Row(DAY=datetime.date(2023, 1, 5), HOLIDAY_FLG=False)]
```

The final step is to write this data frame to a table in Snowflake. The table will be created in the BAKERY_DB database and the SNOWPARK schema because we supplied these values in the connection parameters when connecting to Snowpark. We will store the data in a table named DIM_DATE. We will also use the "overwrite" mode so that it will be overwritten if the table already exists.

> **NOTE** When developing, we change the code frequently for testing, so it's convenient to overwrite the table each time we execute the code. In a production environment, you must consider the requirements and decide whether to overwrite a table, append or merge data, or throw an error if the table already exists.

The following listing shows the code that saves the df data frame to the DIM_DATE Snowflake table.

Listing 6.10 Saving the data frame to a Snowflake table

```
# save the data frame to a Snowflake table named DIM_DATE and overwrite the
    table if it already exists
df.write.mode("overwrite").save_as_table("DIM_DATE")
```

Finally, close the connection at the end of your code. The complete Python code combining listing 6.5 and all subsequent listings until and including listing 6.10 is in the GitHub repository, in the Chapter 06 folder, in a file named Chapter_06_Part6_Snowpark_date_dim.py.

Writing modular code

The code we wrote in this section's example is simple for illustration purposes. All the code resides in a single Python file, which is convenient for quick prototyping but can become challenging to manage when adding more functionality and when more than one developer is working on the code.

Following good programming practices, data engineers must strive to write code that is easy to maintain and modularized as much as possible so that they can test individual pieces of the code and reuse them when needed.

For example, we could create the `is_holiday()` function in a separate file and import it into the main code, which should start with `__main__`. Connecting to Snowflake could also be built as a stand-alone function that returns the session object, which we can call from the main program. We could also write other parts of the code as individual Python functions, methods, or classes.

After executing the code, you can verify that the DIM_DATE table was created in Snowflake in the SNOWPARK schema in the BAKERY_DB database. You can view the data in the table by executing the following SQL command in an SQL worksheet:

```
select * from BAKERY_DB.SNOWPARK.DIM_DATE;
```

The result should be five rows of data, each containing a date starting from January 1, 2023, and each indicating whether the date is a holiday. A sample output is shown in table 6.1.

Table 6.1 Sample output when selecting data from the `DIM_DATE` table

Day	Holiday flag
2023-01-01	True
2023-01-02	True
2023-01-03	False
2023-01-04	False
2023-01-05	False

Now that we have created the sample date dimension and are confident that the code is working as expected, we want to generate a date dimension with dates for 2023 and 2024. To do that, we can adjust the `no_days` parameter in listing 6.5 to a value representing the total number of days in both years, which is 731 (365 days in 2023 and 366 days in 2024 since it is a leap year).

Let's do that now to have the date dimension ready for use in the following sections. Change the `no_days` parameter to 731. Consider commenting the `print()` statements in the code we used for testing so that the data doesn't print to the console. Then rerun the code that generates the date dimension.

Finally, check that the `DIM_DATE` table contains 731 rows with dates for 2023 and 2024, along with flags indicating whether each date is a holiday.

We will need this table to join with order data, but first we must ingest order data into Snowflake. Let's do that next.

6.6 Ingesting data from a CSV file into a Snowflake table

In chapter 2, the bakery employees created CSV files with order information that they manually uploaded to a Snowflake internal stage for ingestion. This step can be automated using Snowpark.

We will write Snowpark Python code that reads a CSV file from the local file system and saves the file in an internal stage in Snowflake. Like in chapter 2, we can execute a `COPY` command that loads data from the staged file into a staging table. Since we are using Snowpark, we can run the `COPY` command from Snowpark rather than executing it as an SQL command, as in chapter 2.

Figure 6.5 illustrates and compares the two pipelines:

- The first pipeline includes a manual step where a bakery employee uploads a file to an internal stage, as described in chapter 2.

- The second pipeline is built by using Snowpark to automate uploading a file into an internal stage and executes the `COPY` command to load data from the staged file into the staging table in Snowflake.

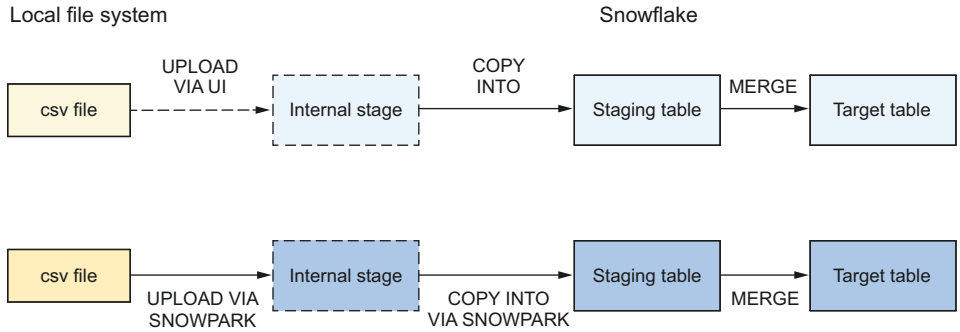


Figure 6.5 Comparison of two pipelines: the first pipeline includes a manual step where a bakery employee uploads a file to an internal stage; the second pipeline uses Snowpark to automate uploading a file into an internal stage and executes the `COPY` command to load the data into the staging table in Snowflake.

To follow along with this example, we will use the Orders_2023-07-07.csv file, which is in the GitHub repository in the Chapter_06 folder.

We will create an internal stage named `ORDERS_STAGE` in the `SNOWPARK` schema in the `BAKERY_DB` database by executing the following SQL commands in a SQL worksheet:

```
use database BAKERY_DB;
use schema SNOWPARK;
create stage ORDERS_STAGE;
```

In the Snowpark code, we will need the following Python packages:

- `snowflake.snowpark`—To establish a connection to Snowflake from Snowpark Python.
- `snowflake.snowpark.types`—To define the data types used in the CSV file, we will need `DateType`, `StringType`, and `IntegerType`, as well as the `StructType` and `StructField` types that are required to specify the schema when saving the data to Snowflake.
- `json`—To read the connection parameters.

We will also define a variable named `source_file_name` and assign the filename as the variable's value. The code that imports packages and assigns the filename to a variable is shown in the following listing.

Listing 6.11 Importing Python packages required for reading CSV files

```
# import Session from the snowflake.snowpark package
from snowflake.snowpark import Session
# import data types from the snowflake.snowpark package
from snowflake.snowpark.types import StructType, StructField, DateType,
     StringType, DecimalType
# import json package for reading connection parameters
import json
# assign the source file name to a variable
source_file_name = 'Orders_2023-07-07.csv'
```

Then we will connect to Snowflake by creating a connection object using a credential file like in listing 6.4.

Next we will use the `file.put()` method of the session object to upload the file as defined in the `source_file_name` parameter to the `ORDERS_STAGE` internal stage. The code that performs this step is shown in the following listing.

Listing 6.12 Uploading the file to an internal stage

```
result = my_session.file.put(source_file_name, "@orders_stage")
print(result)
```

We must define a schema for the CSV file. As we already know from chapter 2, the order files contain five columns: customer, order date, delivery date, baked goods type, and quantity. The data types of these columns are date for the two dates, decimal for the quantity, and string for the customer and baked goods type. The code for constructing the schema is shown in the following listing.

Listing 6.13 Defining the schema

```
# define the schema for the CSV file
schema_for_csv = StructType(
        [StructField("Customer", StringType()),
         StructField("Order_date", DateType()),
         StructField("Delivery_date", DateType()),
         StructField("Baked_good_type", StringType()),
         StructField("Quantity", DecimalType())
        ])
```

Detecting the schema in staged files

When loading data from CSV files and other semistructured data into Snowflake tables, we don't necessarily have to supply the schema definition. Instead, we can use the `INFER_SCHEMA` function, which automatically detects column definitions in staged files.

In this chapter, we supply the schema definition because we want to control the columns that will be ingested. More information about the `INFER_SCHEMA` function is available in the Snowflake documentation at https://mng.bz/2gR9.

We will use the `read.schema()` method to read the contents of the staged CSV file into a data frame named `df`. Then we will use the `copy_into_table()` method to save the data from the data frame to a Snowflake table named `ORDERS_STG`.

> **NOTE** We don't have to create the `ORDERS_STG` staging table before calling the `copy_into_table()` method. The table will be created if it doesn't exist.

Because the order files contain headers, we must skip the first line in the file, which we do by providing the `format_type_options = {"skip_header": 1}` option. The code that reads the CSV file into a data frame using the defined schema and copies the data into a Snowflake table is shown in the following listing.

Listing 6.14 Reading the file into a data frame and copying it to a table

```
# COPY data from the CSV file to the staging table
# using the session.read method
df = my_session.read.schema(schema_for_csv).csv("@orders_stage")
result = df.copy_into_table("ORDERS_STG",
  format_type_options = {"skip_header": 1})
```

You can check that the data was copied to the staging table by selecting data from the `ORDERS_STG` table from an SQL worksheet using the following command:

```
use schema SNOWPARK;
select * from ORDERS_STG;
```

Sample output from this command is shown in table 6.2.

Table 6.2 Sample output when selecting data from the `ORDERS_STG` table (not all data is shown)

Customer	Order date	Delivery date	Baked good type	Quantity
Coffee Pocket	2023-07-09	2023-07-13	Baguette	5
Coffee Pocket	2023-07-09	2023-07-13	Bagel	10
Coffee Pocket	2023-07-09	2023-07-13	English muffin	12
Coffee Pocket	2023-07-09	2023-07-13	Croissant	13
Lily's Coffee	2023-07-09	2023-07-13	Bagel	15

The complete Python code that reads a CSV file and saves the data to a Snowflake table is in the GitHub repository in the Chapter 06 folder in the Chapter_06_Part7_Snowpark_ingest_CSV.py file.

6.7 Transforming data with data frames

The Snowpark library contains many methods for working with data frames corresponding to SQL query commands. For example, we will join the order data in the `ORDERS_STG` staging table with data from the `DIM_DATE` date dimension. The result of

this join will contain the order information with an additional column that indicates whether the delivery date is a holiday.

Thinking in SQL terms, we want to left-join the ORDERS_STG table with the DIM_DATE table using the DELIVERY_DATE column from the ORDERS_STG table and the DAY column from the DIM_DATE table and selecting all columns from both tables. If we were using SQL, we could write the statement in the following listing.

Listing 6.15 Joining the orders table with the date dimension

```
use schema SNOWPARK;
select
  customer, order_date, delivery_date, baked_good_type, quantity,
  day, holiday_flg
from ORDERS_STG
left join DIM_DATE
  on delivery_date = day;
```

Since we are working in Snowpark, we will perform the SQL statement in listing 6.15 using data frame functionality. We will start by importing the Python packages and establishing a connection to Snowflake by creating a session object, as in listing 6.4.

Then we can use the table() method of the my_session object to retrieve each of the two tables into a data frame:

```
df_orders = my_session.table("ORDERS_STG")
df_dim_date = my_session.table("DIM_DATE")
```

We can use the join() method to join the two data frames, naming the resulting data frame DF_ORDERS_WITH_HOLIDAY_FLG:

```
df_orders_with_holiday_flg = df_orders.join(
  df_dim_date,
  df_orders.delivery_date == df_dim_date.day,
  'left'
)
```

We could save the resulting data frame as a table in Snowflake using the save_as_table() method like listing 6.10. But instead of creating a physical table, let's create a view from this data frame using the create_or_replace_view() method:

```
df_orders_with_holiday_flg.create_or_replace_view("ORDERS_HOLIDAY_FLG")
```

The complete Python code that joins the data frames and creates the ORDERS_HOLIDAY_FLG view is in the GitHub repository in the Chapter 06 folder in the Chapter_06_Part8_Snowpark_data_frames.py file.

To check that the view was created successfully, we can expand the BAKERY_DB database in the Databases pane in the Snowsight user interface and then expand the SNOWPARK schema and look for the ORDERS_HOLIDAY_FLG view in the Views folder as shown in figure 6.6.

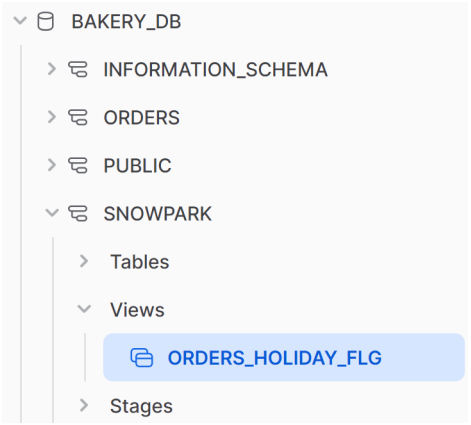


Figure 6.6 Expanding the `BAKERY_DB` database, the `SNOWPARK` schema, and the Views folder to verify that the `ORDERS_HOLIDAY_FLG` view was created

We can also select data from the `ORDERS_HOLIDAY_FLG` view by executing the following SQL command:

```
use schema SNOWPARK;
select * from ORDERS_HOLIDAY_FLG;
```

Sample output from this command is shown in table 6.3.

Table 6.3 Sample output when selecting data from the `ORDERS_HOLIDAY_FLG` view (not all data is shown)

Customer	Order date	Delivery date	Baked good type	Quantity	Day	Holiday flag
Coffee Pocket	2023-07-09	2023-07-13	Baguette	5	2023-07-13	FALSE
Coffee Pocket	2023-07-09	2023-07-13	Bagel	10	2023-07-13	FALSE
Coffee Pocket	2023-07-09	2023-07-13	English muffin	12	2023-07-13	FALSE
Coffee Pocket	2023-07-09	2023-07-13	Croissant	13	2023-07-13	FALSE
Lily's Coffee	2023-07-09	2023-07-13	Bagel	15	2023-07-13	FALSE

In this chapter, we reviewed some of the Snowpark Python features to get a better feel for working with Snowpark. The next chapter will expand on these features to build an end-to-end data pipeline in Snowpark Python.

Summary

- Snowpark is an umbrella term encompassing features on the server side and the client side. On the server side, Snowpark executes code natively in Python, Java, and Scala. On the client side, Snowpark provides API libraries for each supported programming language.

- The data structure in Snowpark is the data frame. The Snowpark API library provides methods for querying and transforming data frames. These queries are internally translated into SQL statements and executed in Snowflake.
- The code that defines functions and stored procedures in Snowpark is serialized and sent to the Snowflake server, where it executes in a secure sandbox. You can also use built-in Python packages if needed.
- You can quickly and conveniently create, execute, and deploy Snowpark Python code from a Python worksheet in the Snowsight user interface.
- When many developers work on the same codebase that must support source control, versioning, branching, and automatic deployments, it is better to use the Snowpark SQL API from a local development environment. This API can be used together with source control, such as Git.
- When executing Snowpark Python in your local development environment, you must first establish a connection to your Snowflake account. You can provide the connection information, such as the Snowflake account name, username, password, and optionally, role, virtual warehouse, database, and schema in a Python dictionary object.
- To avoid exposing credential information, we can use a configuration file in any structured format, such as CSV, JSON, or YML, that can be interpreted by Python and converted into a dictionary to be passed as the connection parameters when establishing a session.
- We can work with data frames in Snowpark Python by using methods such as creating a data frame from a Python list and writing the data to a Snowflake table.
- We can use Snowpark Python to read a CSV file from the local file system and save the file in an internal stage in Snowflake, avoiding uploading the file manually. We can also run the `COPY` command to load data from the staged file into a Snowflake table.
- The Snowpark library contains many methods for working with data frames corresponding to SQL query commands, such as joining tables or creating views.

7 Augmenting data with outputs from large language models

This chapter covers

- Understanding external network access
- Configuring external network access
- Calling API endpoints from Snowpark
- Retrieving customer reviews from websites like Yelp
- Deriving customer review sentiments
- Interpreting emails using large language models to save time

Snowflake's *external network access* functionality enables data engineers to access network locations external to Snowflake. This opens countless opportunities to access data by calling API endpoints from Snowflake stored procedures or user-defined functions (UDFs). For example, data engineers can retrieve external data that augments the data already stored in Snowflake. With the proliferation of generative AI capabilities and large language models (LLMs), data engineers can incorporate the outputs of such models by calling the Snowflake Cortex LLM functions.

In this chapter, we will learn how to retrieve customer reviews from an external website and store them in Snowflake tables for further analysis. We will call an LLM

to help us understand the reviews and classify their sentiment. We will also use an LLM to help us interpret unstructured text from the body of an email and derive structured information from it.

To illustrate the examples in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries and delivers these baked goods to small businesses, such as grocery stores, coffee shops, and restaurants in the neighborhood. The bakery also signed up with a food delivery service that delivers orders throughout the day. The food delivery service's website allows customers to leave reviews regarding the service and quality of the delivered food. The bakery employees read the reviews on the food delivery's website, but this takes a lot of time. They want to ingest the reviews into a Snowflake table and then use an LLM to classify them as positive, negative, or neutral. They can then choose to read only a specific type of review. For example, they could read only the negative reviews to look for areas of improvement.

Since the bakery has no online ordering system, customers order baked goods via email. A bakery employee reads the emails and stores the order information in CSV files on their local file system. This involves manual work because the employee must read the email, understand the content, extract the structured information, such as how many of each type of baked goods the customer ordered on which date, and save the data to a CSV file. The employee could instead use an LLM to interpret the body of each received email and return the order information in a structured format, saving time.

> **NOTE** All code for this chapter is available in the accompanying GitHub repository in the Chapter_07 folder at https://mng.bz/1aZy. The SQL and Python code is stored in multiple files whose names start with Chapter_07_Part*, where * indicates a sequence number and additional information means the file's content. Please follow the exercises by reading the files sequentially.

Creating an external access integration object in Snowflake is a prerequisite for building pipelines that ingest data from external API endpoints. This object provides a secure and governed method for ingesting data from external APIs, complying with the best data management and security practices described in the next section.

> **NOTE** External network access is currently not supported for Snowflake trial accounts. You must have access to a paid Snowflake account to call external API endpoints.

7.1 Configuring external network access

Before a data engineer can call external API endpoints from data pipelines, a Snowflake administrator or a user with sufficient privileges must enable access to the external network location. The access is encapsulated in a Snowflake object called an *external access integration*. The data engineer is then granted access to the external access integration to use it in a data pipeline. A Snowflake administrator can monitor the requests made to external network locations.

The following steps are required to create an external access integration:

1. Create a network rule to represent one or more external network locations.
2. Create a secret to store credentials for accessing the external network location. The secret is then used instead of providing a credential value, such as a token, key, or password.
3. Create an external access integration using the network rule and the secret.

The described external network access architecture is illustrated in figure 7.1.

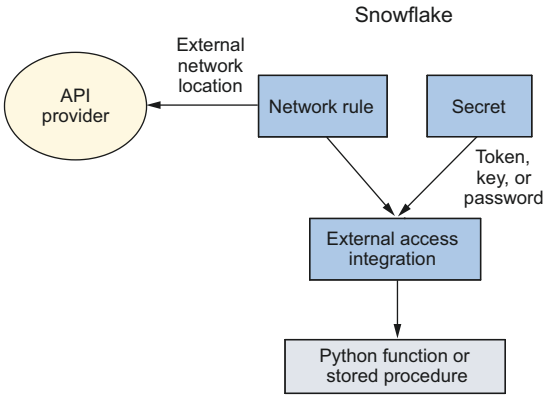


Figure 7.1 Snowflake's external network access architecture consists of a network rule pointing to an external network location and a secret containing a token, key, or password. The network rule and secret are included as parameters when creating an external access integration object. This object is passed as a parameter to a Python function or stored procedure so that the function or procedure can call the external API.

To demonstrate how to use external access, we will create an external access integration to an external website that hosts customer reviews. Since we are working with a fictional bakery, there are no reviews for this bakery. To simulate how the bakery might accomplish this task in the real world, we will read customer reviews on the Yelp.com website. This website publishes crowd-sourced reviews about businesses worldwide.

We will randomly choose a bakery on Yelp's website and ingest the associated reviews into a Snowflake table. We can only retrieve three customer reviews using Yelp's free API endpoints. In an actual situation, the bakery might pay to use the partner API from the food delivery service, enabling them to retrieve all available customer reviews.

Getting information about public API endpoints

Many websites provide access via public API endpoints. An API endpoint is a URL that accepts requests and sends back responses. Documentation that describes the endpoints is usually available, including the input parameters that can be specified when calling each endpoint and the format of the output that the endpoint returns.

To call API endpoints, you need a method of authentication, such as an API key or token. The website's documentation should also provide instructions on authenticating to the API endpoint and generating the required credentials.

To start with our example, we must research how to access Yelp's website API. A simple internet search for "Yelp API" leads us to the "Getting Started with the Yelp Fusion API" documentation at https://docs.developer.yelp.com/docs/fusion-intro.

Here we learn that all API endpoints are under `https://api.yelp.com/v3`. We also find a list of available API endpoints with their descriptions. For this exercise, we will use the "Reviews (up to 3 excerpts)" endpoint, which returns up to three review excerpts for a business whose path is `/businesses/{business_id_or_alias}/reviews`.

The documentation then directs us to the authentication guide, which describes creating an app and obtaining a private API key. The authentication guide is available at https://mng.bz/PNBg. If you are following along with this exercise, please read the instructions in the authentication guide to register on Yelp's developer website, create your app, and obtain your private API key.

Once you know the API endpoint URL and have your API key, you can create an external access integration. For the exercises in this chapter, we will use the `BAKERY_DB` database that we created in chapter 2 and continue working using the `SYSADMIN` role. We will create a new schema named `REVIEWS` in the `BAKERY_DB` database by executing the following commands:

```
use role SYSADMIN;
use database BAKERY_DB;
create schema REVIEWS;
use schema REVIEWS;
```

Because a user needs privileges to create objects required for the external access integration, we will grant the `CREATE NETWORK RULE`, `CREATE SECRET`, and `CREATE INTEGRATION` privileges to the `SYSADMIN` role:

```
use role ACCOUNTADMIN;
grant create network rule on schema REVIEWS to role SYSADMIN;
grant create secret on schema REVIEWS to role SYSADMIN;
grant create integration on account to role SYSADMIN;
```

Now we can use the `SYSADMIN` role to create a network rule named `YELP_API_NETWORK_RULE`. Creating a network rule requires a few parameters, such as `EGRESS` as the mode (meaning "going out," as opposed to `INGRESS` or "coming in"), `HOST_PORT` as the type, and a list of external locations as the value list. Since we need this network rule to access Yelp's website, we will provide the `api.yelp.com` external location in the value list. The command that creates the network rule is

```
use role SYSADMIN;
create network rule YELP_API_NETWORK_RULE
  mode = EGRESS
  type = HOST_PORT
  value_list = ('api.yelp.com');
```

Next we will create a secret to store the API key we obtained earlier. We will name the secret `YELP_API_TOKEN`; we will specify `GENERIC_STRING` as the type (the type used with

API keys) and supply the value of our API key in the secret string, using `'ab12DE...89XYZ'` as the API key (this is a fictional API key for illustrative purposes). The command that creates the secret is

```
create secret YELP_API_TOKEN
  type = GENERIC_STRING
  secret_string = 'ab12DE...89XYZ';
```

In our example, we are using the `SYSADMIN` role to create the secret. The same role will also use the secret because we don't have any custom roles. In general, a Snowflake administrator creates the secret using an administrative role and then grants usage of the secret to the custom role used by the data engineer. We can execute the following command to grant usage on the secret to a custom role:

```
grant read on secret YELP_API_TOKEN to role <custom_role>;
```

> **TIP** A best practice is to use a secret to store access credentials rather than specifying the credentials in your code. This helps to protect the credentials from being unintentionally exposed and enables Snowflake administrators to manage usage of the credentials because they must grant the privilege to use the secret to individual data engineers.

Finally, we can create an external access integration named `YELP_API_INTEGRATION` by providing the `YELP_API_NETWORK_RULE` network rule and the `YELP_API_TOKEN` secret as the parameters using the following command:

```
create external access integration YELP_API_INTEGRATION
  allowed_network_rules = (YELP_API_NETWORK_RULE)
  allowed_authentication_secrets = (YELP_API_TOKEN)
  enabled = TRUE;
```

A data engineer needs the `USAGE` privilege on the external access integration and the `READ` privilege on the secret to use the integration. In this exercise, we use the `SYSADMIN` role to create and use the objects, so we don't have to explicitly grant these privileges because the `SYSADMIN` role owns the objects.

> **NOTE** For more information about creating external network access and various options when specifying the parameters to each command, refer to the Snowflake documentation at https://mng.bz/JNWP.

7.2 Calling an API endpoint from a Snowpark function

Now that we have configured external access integration to Yelp's API, we can use it in a Snowpark UDF. We will create a Snowpark Python UDF that calls the API and returns customer reviews from Yelp's website. We will store the returned data in a Snowflake table for further processing.

Go to Yelp's website, and find a random bakery to retrieve its reviews. In the search box on https://www.yelp.com/, type "bakery" in the first search box and a city of your

choice in the second search box (we chose Paris, France, in this example). You will get a list of bakeries in your selected city. Scroll to find a bakery with many reviews (it should have at least three reviews because the API endpoint that we will use returns up to three review excerpts). Click the bakery name to open the bakery's page (we chose Boulangerie Julien). Examine the URL of the bakery's page; in our example, it looks like

```
https://www.yelp.com/biz/boulangerie-julien-paris-3?osq=bakery
```

We need the business alias name because we will provide it as a parameter to the API endpoint. In our example, the business alias name is `boulangerie-julien-paris-3`; the text follows the `biz/` part of the URL and comes before the "?" part. If you chose a different bakery, your business alias name should follow the same pattern.

7.2.1 Constructing the UDF that retrieves customer reviews

We will use the `requests` Python package to perform the API request from the body of the UDF. More information about this package is available in the documentation at https://requests.readthedocs.io/en/latest/.

With the information we have gathered so far, we can create a UDF named `GET_CUSTOMER_REVIEWS` with the following definition:

- The function accepts a parameter named `business_alias`.
- It returns a result as a variant data type.
- It uses Python as the language.
- The Python version is 3.10 (any of the supported Python versions would work).
- The name of the handler code is `get_reviews` (this is the name of the Python function in the body of the UDF).
- It uses the `YELP_API_INTEGRATION` as the external access integration.
- It uses the `YELP_API_TOKEN` secret.
- It uses the `requests` Python package.

The code that creates the definition of the UDF (we will add more code later) is shown in the following listing.

Listing 7.1 The definition of the UDF

```
create or replace function GET_CUSTOMER_REVIEWS(business_alias varchar)
returns variant
language python
runtime_version = 3.10
handler = 'get_reviews'
external_access_integrations = (YELP_API_INTEGRATION)
secrets = ('yelp_api_token' = YELP_API_TOKEN)
packages = ('requests')
AS
```

More code to follow in listing 7.2

In the body of the UDF, written in Python, we must import the required packages:

- `_snowflake`
- `requests`

NOTE Although we must import two packages in the Python code (`_snowflake` and `requests`), we only list the `requests` package in the packages parameter when defining the Snowpark UDF in listing 7.1. The reason is that `_snowflake` is a module exposed to Python UDFs that execute within Snowflake and is known to Snowpark without explicitly listing it.

We then create the `get_reviews` Python handler function with a `business_alias` parameter that performs the following:

- Retrieves the API key from the `YELP_API_TOKEN` secret into a variable named `api_key` using the `get_generic_secret_string()` method
- Constructs the API endpoint URL by replacing the `{business_id_or_alias}` placeholder in the `/businesses/{business_id_or_alias}/reviews` template with the value of the `business_alias` parameter
- Calls the `get()` method of the `requests` package to retrieve the result from the API endpoint using the URL and the API key as the parameters
- Returns the response in JSON format

The code in the body of the UDF is shown in the following listing.

Listing 7.2 The body of the UDF

```
$$                                    <-- Continuation from listing 7.1
import _snowflake
import requests

def get_reviews(business_alias):
  api_key = _snowflake.get_generic_secret_string('yelp_api_token')
  url = f'''https://api.yelp.com/v3/businesses/{business_alias}/reviews'''
  response = requests.get(
    url=url,
    headers = {'Authorization': 'Bearer ' + api_key}
  )
  return response.json()
$$;
```

The UDF is created once we execute the combined code from listings 7.1 and 7.2 in a worksheet. We can select data from this UDF just like from any other UDF in Snowflake using the SQL `SELECT` command and providing the value `boulangerie-julien-paris-3` that we chose earlier as the parameter (you can use a different parameter if you chose a different bakery):

```
select GET_CUSTOMER_REVIEWS('boulangerie-julien-paris-3');
```

A sample output from this command is shown in figure 7.2 (only the top few lines are included).

```
[] GET_CUSTOMER_REVIEWS('BOULANGERIE-JULIEN-PARIS-3')

{
  "possible_languages": [
    "fr",
    "en"
  ],
  "reviews": [
    {
      "id": "yk_6bL9SflLCPe-_Ul-5HQ",
      "rating": 5,
      "text": "I stayed in Paris for 3 days. Every day, I stopped by here
to buy my pastries...\n\nThe ladies were very nice. They were very eager
to help you which one you...",
      "time_created": "2023-04-09 03:15:00",
      "url": "https://www.yelp.com/biz/boulangerie-julien-paris-3?
```

Figure 7.2 Sample output by selecting from the UDF (only the top few lines are included)

As we can see, the output is in JSON format. We will extract the customer review, the rating, and the time when the customer created the review from the JSON structure and convert the structured data into a tabular format. You can refer to the API endpoint documentation for a description of the JSON object at https://mng.bz/w5rW.

7.2.2 Interpreting the results from the UDF

By examining the data and reading the API endpoint documentation, we know that the information we want to extract (the customer review, the rating, and the time when the customer created the review) is in the value of an embedded JSON object with the "reviews" key. We can select the value of this key by executing the command in the following listing.

Listing 7.3 Selecting the value of the "reviews" key

```
select GET_CUSTOMER_REVIEWS('boulangerie-julien-paris-3'):"reviews";
```

A sample output from this command is shown in the following listing (some nonrelevant information is abbreviated with . . .).

Listing 7.4 Sample value of the "reviews" key

```
[
  {
    "id": "yk",
```

```
    "rating": 5,
    "text": "I stayed in Paris for 3 days...",
    "time_created": "2023-04-09 03:15:00",
    "url": "https://...",
    "user": {...}
  },
  {
    "id": "FHjCTg26L1NlN615SyGHDA",
    "rating": 4,
    "text": "Stopped in on a Tuesday at around 9...",
    "time_created": "2023-10-14 23:49:59",
    "url": "https://...",
    "user": {...}
  },
  {
    "id": "Hw2k9Q6Ktg3-yStDsL_QVg",
    "rating": 5,
    "text": "After spending hours inside the Louvre...",
    "time_created": "2022-12-22 08:43:28",
    "url": "https://...",
    "user": {...}
  }
]
```


The value of the "reviews" key contains an array, which we recognize because it is enclosed in square brackets []. To store the values from the "rating," "text," and "time_created" keys in a relational table, we must flatten them into separate rows using the `FLATTEN` syntax described in chapter 4.

> **TIP** We only need the `FLATTEN` keyword in this example without the `LATERAL` modifier as in chapter 4 because the purpose of the query in this example is to flatten the data, not to join it to a main table expression.

The SQL command in the following listing flattens the output of the query from listing 7.3 by extracting the values from the "rating," "text," and "time_created" keys.

Listing 7.5 Flattening the values from the selected keys

```
select
  value:"rating"::number as rating,
  value:"time_created"::timestamp as time_created,
  value:"text"::varchar as customer_review
from table(flatten(
  input => GET_CUSTOMER_REVIEWS(
  'boulangerie-julien-paris-3'):"reviews"
));
```

The query from listing 7.3

A sample output from this query is shown in table 7.1 (customer reviews are abbreviated with . . .). Your output may differ if you used a different bakery or if new reviews were added since this chapter was written.

Table 7.1 Sample output after flattening the values from the "reviews" key

Rating	Time created	Customer review
5	2023-04-09 03:15:00	I stayed in Paris for 3 days…
4	2023-10-14 23:49:59	Stopped in on a Tuesday at around 9…
5	2022-12-22 08:43:28	After spending hours inside the Louvre…

After flattening the data, we can store it in a table for further analysis.

7.2.3 Storing the customer reviews in a table

Let's create a table named `CUSTOMER_REVIEWS` in the `REVIEWS` schema that we created earlier using the following command:

```
use schema REVIEWS;
create table CUSTOMER_REVIEWS (
  rating number,
  time_created timestamp,
  customer_review varchar
);
```

Before we insert the data into the `CUSTOMER_REVIEWS` table, we will remove any unprintable characters from the customer reviews using a *regular expression*. We will keep only the characters we will use for further text analyses, such as lowercase letters from a to z, uppercase letters from A to Z, digits from 0 to 9, and a few punctuation symbols, including spaces, periods, commas, exclamation marks, question marks, and dashes. We will not store any other unprintable characters or characters in foreign languages in the table.

> **Regular expressions**
>
> Functions that search for patterns in strings and perform actions such as matching the pattern, replacing a pattern with a different pattern, extracting a pattern, etc., use regular expressions. For more information about regular expression functions supported in Snowflake, refer to the documentation at https://mng.bz/q0EE.

We will use the `REGEXP_REPLACE` regular expression command that keeps only the characters specified in the match pattern:

```
regexp_replace(value:"text"::varchar, '[^a-zA-Z0-9 .,!?-]+')
```

Now we can insert the results of the query from listing 7.5 into the `CUSTOMER_REVIEWS` table using the following command:

```
insert into CUSTOMER_REVIEWS
select
```

```
  value:"rating"::number as rating,
  value:"time_created"::timestamp as time_created,
  regexp_replace(value:"text"::varchar,
    '[^a-zA-Z0-9 .,!?-]+')::varchar
    as customer_review
from table(flatten(
  input => GET_CUSTOMER_REVIEWS('boulangerie-julien-paris-3'):"reviews"
));
```

Regular expression that keeps only the characters specified in the match pattern

To verify that the data was inserted successfully, we can query the `CUSTOMER_REVIEWS` table using the following command:

```
select * from CUSTOMER_REVIEWS;
```

The output of this query should look like table 7.1. Up to three reviews are returned because we are using Yelp's free API.

> **WARNING** In this example, we insert data into the target table without checking whether the data already exists. In an actual situation, the data engineer should check the documentation to determine whether the API endpoint accepts parameters that limit the number of customer reviews returned with each API call. Another precaution to avoid duplication is to merge the data received from the API endpoint into the target table instead of inserting it.

Once the customer review data is retrieved from an external website and stored in a Snowflake table, we can continue exploring how to analyze this data using LLMs.

7.3 Deriving customer review sentiments

An LLM is an artificial intelligence algorithm architected to generate text content. It uses deep learning techniques and is trained on massive data sets to understand, summarize, translate, or generate output.

Calling an LLM via an API

One of the most well-known LLMs is OpenAI's GPT-3. You can use it via an API, but note that it's not free. You must create an account with OpenAI and purchase credits before using it. More information about calling GPT-3 via an API is available at https://platform.openai.com/docs/api-reference/chat.

In this chapter, we will use the Snowflake Cortex LLM functions. Snowflake hosts and manages these functions, giving you access to LLMs trained by researchers at companies like Mistral, Reka, Meta, and Google. You can also use Snowflake Arctic, a model developed by Snowflake.

Some of the available Snowflake Cortex LLM functions are

- `COMPLETE`—Returns a response that completes the prompt
- `EMBED_TEXT_768`—Returns a vector embedding that represents the given text

- `EXTRACT_ANSWER`—Returns the answer to a question based on unstructured data
- `SENTIMENT`—Returns a sentiment score of the given text
- `SUMMARIZE`—Returns a summary of the given text
- `TRANSLATE`—Translates the given text from a supported language to another

NOTE More information about the Snowflake Cortex LLM functions is available in the documentation at https://mng.bz/75ng.

We will use the Snowflake Cortex `SENTIMENT` function to return the customer review's sentiment as positive, negative, or unknown. The `SENTIMENT` function takes a string and returns a floating-point number from –1 to 1, indicating the level of negative or positive sentiment in the text.

To work with Snowflake Cortex LLM functions, a user needs the required privilege, which we will grant to the `SYSADMIN` role by executing the following commands:

```
use role ACCOUNTADMIN;
grant database role SNOWFLAKE.CORTEX_USER to role SYSADMIN;
```

Then we will continue working with the `SYSADMIN` role in the `REVIEWS` schema:

```
use role SYSADMIN;
use database BAKERY_DB;
use schema REVIEWS;
```

Let's call the `SENTIMENT` function by providing the string `'The service was excellent!'` as the parameter:

```
select SNOWFLAKE.CORTEX.SENTIMENT('The service was excellent!');
```

The output of this command is a sentiment score with a value of 0.7691993, which is close to 1, indicating a positive sentiment.

Let's try another example:

```
select SNOWFLAKE.CORTEX.SENTIMENT('The bagel was stale.');
```

The output of this command is a sentiment score with a value of –0.42285544, indicating a negative sentiment.

Here is another example:

```
select SNOWFLAKE.CORTEX.SENTIMENT('I went to the bakery for lunch.');
```

This time, the sentiment score is 0.15711609, indicating a neutral sentiment.

Since we want to classify a customer's review as positive, negative, or neutral, we can define limits for sentiment scores between –1 and 1, where we want to draw the boundaries between the defined sentiment classes. We can choose these values according to our judgment and adjust them after viewing the results. For this example, let's take sentiment scores from –1 to –0.7 as negative, scores from above –0.7 to 0.4 as neutral, and scores above 0.4 as positive.

To derive the sentiment class from the sentiment score, we can use the SQL CASE statement:

```
case
    when sentiment_score < -0.7 then 'Negative'
    when sentiment_score < 0.4 then 'Neutral'
    else 'Positive'
  end
```

To derive the customer sentiment scores from customer reviews in the CUSTOMER_REVIEW table, we can use the SENTIMENT function when selecting from the table, providing the CUSTOMER_REVIEW column as the parameter to the function, and calculating the sentiment class with a CASE statement. The query that returns the sentiment scores and classes is

```
select
  rating,
  time_created,
  customer_review,
  SNOWFLAKE.CORTEX.SENTIMENT(customer_review) as sentiment_score,
  case
    when sentiment_score < -0.7 then 'Negative'
    when sentiment_score < 0.4 then 'Neutral'
    else 'Positive'
  end as sentiment
from CUSTOMER_REVIEWS;
```

The output from this command is shown in table 7.2 (customer reviews are abbreviated with . . .). Your output may be different if the data in your CUSTOMER_REVIEWS table is different.

Table 7.2 Sample output after deriving the sentiment score with the SENTIMENT function from the CUSTOMER_REVIEWS table

Rating	Time created	Customer review	Sentiment score	Sentiment
5	2023-04-09 03:15:00	I stayed in Paris for 3 days…	0.8525086	Positive
4	2023-10-14 23:49:59	Stopped in on a Tuesday at around 9…	0.8410726	Positive
5	2022-12-22 08:43:28	After spending hours inside the Louvre…	0.64034367	Positive

Sentiment classification could help the bakery employees reduce the number of customer reviews they read by only reading those that fit a particular classification. They could also track the sentiment distribution over time to see the proportion of positive and negative reviews, helping them to improve their service.

7.4 Interpreting order emails using LLMs to save time

Another example of how bakery employees can save time using LLMs is by reading and interpreting orders in customer emails. Our fictional bakery has no online ordering system. It receives order information via email, and a bakery employee must read the emails and extract order information into a structured format, such as CSV.

We can construct a prompt that asks an LLM to read and understand the email's contents, extract the order information in a structured format, and write the data to a Snowflake table. Let's walk through this example.

We will create a stored procedure that constructs a prompt instructing the Snowflake Cortex `COMPLETE` function to read the contents of an email and return the data in CSV format. We will then add code that converts the CSV into a data frame and saves the data frame to a table in Snowflake.

7.4.1 Creating a stored procedure that interprets customer emails

The definition of the stored procedure is similar to the definitions of the `GET_CUSTOMER_REVIEWS` and `REVIEW_SENTIMENT` UDFs that we created earlier, with a few differences:

- Instead of a UDF, we will create a stored procedure because we need procedural logic to retrieve the results from the model and transform and save the results to a table.
- The name of the stored procedure is `READ_EMAIL_PROC`, and the procedure accepts a parameter named `email_content` that passes the contents of an email to the procedure.
- The return type is `table()` because the procedure returns structured information in the form of a table.
- The handler function is `get_order_info_from_email`.
- The stored procedure uses the `snowflake-ml-python` package, which provides the Snowflake machine learning and LLM functionality.
- The stored procedure also uses the `snowflake-snowpark-python` package, which we will need to convert the CSV data to a data frame and save it to a Snowflake table.

The code that creates the definition of the stored procedure is

```
create or replace procedure READ_EMAIL_PROC(email_content varchar)
returns table()
language python
runtime_version = 3.10
handler = 'get_order_info_from_email'
packages = ('snowflake-snowpark-python', 'snowflake-ml-python')
```

In the body of the stored procedure, we will import the `_snowflake` package as previously. We will also import the `snowflake.snowpark` package. We will use this package to import the `snowpark` library and the data types required to define the structure

when saving the data to a table. Finally, we will import the `Complete` function from the `snowflake.cortex` package:

```
import _snowflake
import snowflake.snowpark as snowpark
from snowflake.snowpark.types import StructType, StructField, DateType,
  StringType, IntegerType
from snowflake.cortex import Complete
```

After importing the required packages and libraries, we will create the handler function named `GET_ORDER_INFO_FROM_EMAIL`. This function takes two parameters. The first parameter is the `session` variable needed for the stored procedure to work with the Snowpark API library objects, as described in chapter 6. The second parameter, `email_content`, is propagated from the stored procedure definition. The following code represents the definition of the handler function and the parameters:

```
def get_order_info_from_email(session: snowpark.Session, email_content):
```

7.4.2 Constructing the prompt

The next step is to construct the prompt. Here we can be as creative as needed to explain precisely what we need from the LLM. If we are unhappy with the LLM's output, we can refine the prompt in subsequent interactions to achieve better results. The initial value of the prompt is

```
prompt = f"""You are a bakery employee, reading customer emails asking
 for deliveries. Please read the email at the end of this text and
 extract information about the ordered items. Format the information in
 CSV using the following columns: customer, order_date, delivery_date,
 item, and quantity. Format the date as YYYY-MM-DD. If no year is given,
 assume the current year. Use the current date in the format YYYY-MM-DD
 for the order date. Items should be in this list: [white loaf, rye loaf,
 baguette, bagel, croissant, chocolate muffin, blueberry muffin]. The
 content of the email follows this line. \n {email_content}"""
```

> **Prompt engineering**
>
> Prompt engineering is the practice of designing questions or instructions that can be interpreted and understood by generative AI and LLMs. The intent of the question or instruction must be stated unambiguously and with enough detail and background information so that the model understands it. It may take several iterations to refine the prompt when working with AI models until the desired result is achieved.

The next line of code in the body of the stored procedure calls the `COMPLETE` function, providing `snowflake-arctic` as the model and the `prompt` we constructed earlier:

```
csv_output = Complete('snowflake-arctic', prompt)
```

> **NOTE** We are using `snowflake-arctic` as the LLM model in this example. You can use other models, like `mistral-large`, `reka-flash`, `llama2-70b-chat`, or `gemma-7b`. For a list of currently supported models, refer to the documentation at https://mng.bz/mR9M. You can experiment with different models and compare the results if you wish.

If the LLM followed our instructions in the prompt, it should return a CSV output containing five columns named `CUSTOMER`, `ORDER_DATE`, `DELIVERY_DATE`, `ITEM`, and `QUANTITY`. Although we didn't specify the columns' data types, we asked the model to format dates as YYYY-MM-DD so that we can store them as a date data type in a Snowflake table.

7.4.3 Saving the CSV result to a table

Before storing CSV data in a Snowflake table, we must convert it to a data frame and provide the structure. As described in chapter 6, we can provide the structure of a data frame using the `StructType` construct in Snowpark. The following code creates a variable named `schema` that holds the schema definition of the CSV file:

```
schema = StructType([
      StructField("CUSTOMER", StringType(), False),
      StructField("ORDER_DATE", DateType(), False),
      StructField("DELIVERY_DATE", DateType(), False),
      StructField("ITEM", StringType(), False),
      StructField("QUANTITY", IntegerType(), False)
  ])
```

We must manipulate the returned CSV data before passing it to the `create_dataframe` method of the `session` object. Because the `create_dataframe` method accepts a list of lists as the parameter, we must split the CSV data by the line break character (`\n`), then split by comma. We also slice using `[1:]` to remove the header row from the CSV data.

The code that creates a data frame named `ORDERS_DF` from the CSV data stored in the `csv_output` variable using the `schema` schema is

```
orders_df = session.create_dataframe(
  [x.split(',') for x in csv_output.split("\n")][1:], schema)
```

Finally, we can save the contents of the data frame to a Snowflake table named `COLLECTED_ORDERS_FROM_EMAIL`. We use the `append` parameter to indicate that the data is appended rather than overwriting the table. As is customary in Snowpark, the table is created if it doesn't already exist.

The code that saves the data from the data frame into a Snowflake table is

```
orders_df.write.mode("append").save_as_table('COLLECTED_ORDERS_FROM_EMAIL')
```

Although not required, we can return the data frame from the stored procedure so that we can see the results in the COLLECTED_ORDERS_FROM_EMAIL table immediately after executing the stored procedure using this code:

```
return orders_df
```

The complete code of the stored procedure is in the accompanying GitHub repository in the Chapter_07 folder in a file named Chapter_07_Part3_read_emails.sql.

After you execute the complete code, it should create a stored procedure named READ_EMAIL_PROC.

7.4.4 Evaluating the output

We can test the stored procedure and evaluate the output by calling it using the CALL command and providing a sample parameter representing the body of an email. The following is one example of calling the stored procedure with made-up email content:

```
call READ_EMAIL_PROC(
  'Hello, please deliver 6 loaves of white bread on Tuesday, September 5.
  On Wednesday, September 6, we need 16 bagels. Thanks, Lilys Coffee');
```

After calling the stored procedure with the previous command, the output should look like table 7.3.

Table 7.3 Output after calling the READ_EMAIL_PROC stored procedure that returns structured data from a sample customer email

Customer	Order date	Delivery date	Item	Quantity
Lilys Coffee	2023-09-05	2023-09-05	White loaf	6
Lilys Coffee	2023-09-06	2023-09-06	Bagel	16

Additionally, we can check if the data was populated into the COLLECTED_ORDERS_FROM_EMAIL table by selecting from the table using the following command:

```
select * from COLLECTED_ORDERS_FROM_EMAIL;
```

The output from this command should be the same as in table 7.3. The LLM got most of the information right, except it doesn't know the current year and date. It appears it was trained in 2023 and thus considers it the current year.

The following are a few more sample emails that we can use to test the procedure:

```
call READ_EMAIL_PROC (
  'Hi again. At Metro Fine Foods, we are renewing our order for
  Thursday, September 7. We need 20 baguettes, 16 croissants, and
  a dozen blueberry muffins. Have a nice day!');
```

```
call READ_EMAIL_PROC(
  'Greetings! We loved your French bread last week. Please deliver
  10 more tomorrow. Cheers from your friends at Page One Fast Food');

call READ_EMAIL_PROC (
  'Do you deliver pizza? If so, send two this afternoon. If not,
  then some bagels should do. Best, Jimmys Diner');
```

After executing these commands, the `COLLECTED_ORDERS_FROM_EMAIL` table should contain data as in table 7.4.

Table 7.4 Sample data in the `COLLECTED_ORDERS_FROM_EMAIL` table

Customer	Order date	Delivery date	Item	Quantity
Lilys Coffee	2023-09-05	2023-09-05	White loaf	6
Lilys Coffee	2023-09-06	2023-09-06	Bagel	16
Metro Fine Foods	2023-09-07	2023-09-07	Baguette	20
Metro Fine Foods	2023-09-07	2023-09-07	Croissant	16
Metro Fine Foods	2023-09-07	2023-09-07	Blueberry muffin	12
Page One Fast Food	2022-07-19	2022-07-20	White loaf	10
Jimmys Diner	2022-01-01	2022-01-01	Bagel	2

As already mentioned, and as is evident from the data in table 7.4, when compared with the sample emails, the LLM doesn't always get it right. In addition to the incorrect dates appearing in 2023, as in the first and second examples, it took 2022 as the current year in the last two. The model had to guess what "French bread" meant because it was not on the list of baked goods the bakery makes. On subsequent executions of the stored procedure using the same email content and different models, the LLM sometimes guesses that "French bread" means baguette and sometimes white bread.

In their current state, LLMs are proving to be tremendously helpful in mundane tasks, such as extracting structured information from unstructured text. However, we should treat all LLM results cautiously and review them for accuracy. As the areas of generative AI and LLMs are rapidly evolving, we can expect better accessibility to these models and better accuracy of their results.

Snowflake is adding LLM functionality in various ways. In addition to enabling external network access, as described in this chapter, many new features have been announced, such as container services, Cortex functions, SQL natural language assistants, and more.

Summary

- Snowflake's external network access functionality enables data engineers to access network locations external to Snowflake. They can call API endpoints from Snowflake stored procedures or UDFs.
- The external network access architecture consists of a network rule pointing to an external network location and a secret containing a token, key, or password. The network rule and secret are included as parameters when creating an external access integration object. This object is passed as a parameter to a Python function or stored procedure so that the function or procedure can call the external API.
- Before creating an external access integration, you must know the website's API endpoint URL and the API key to access the website. A user with sufficient privileges can create an external access integration.
- To use the external access integration, a data engineer must be granted the `USAGE` privilege and the `READ` privilege on the secret. Snowflake administrators can monitor and limit access to external locations if needed.
- We can use the `requests` Python package to call the API endpoint from the body of the UDF. We must specify this package in the parameters of the UDF definition.
- Responses from API calls are often formatted as JSON objects. If necessary, we must parse and flatten the JSON structure to use the response in an SQL query.
- We can use a regular expression, such as `REGEXP_REPLACE`, to remove any unprintable characters from the result of an API call.
- An LLM is an artificial intelligence algorithm architected to generate text content. It uses deep learning techniques and is trained on massive data sets to understand, summarize, translate, or generate output.
- When calling an LLM, we must provide a prompt that clearly outlines the intent of the question or instruction and provides enough detail and background information so the model understands it. Refining the prompt may take several iterations until the desired result is achieved.
- We can create a stored procedure to call a Snowflake Cortex LLM function when we need procedural logic to retrieve the results from the model and perform additional actions, such as transforming the results and saving them to a Snowflake table.
- We can use the `snowflake-snowpark-python` package to convert the results from the API call to a data frame. We provide the structure of the data frame using the `StructType` construct in Snowpark. Then we can save the data frame as a table in Snowflake.
- LLMs don't always do exactly what we tell them to do and don't always get it right. We should treat all LLM results cautiously and review them for accuracy.

8 Optimizing query performance

This chapter covers

- Getting data from the Snowflake Marketplace
- Performing analysis of geographical data
- Viewing query performance using the query profile
- Understanding Snowflake micro-partitions
- Optimizing storage with clustering
- Improving query performance with search optimization
- General tips for improving query performance

Data engineers must ensure that their data pipelines perform well, especially when dealing with large amounts of data. They should write efficient SQL queries and be familiar with Snowflake optimization techniques to meet user performance requirements.

In this chapter, we will write queries using large amounts of data from the Snowflake Marketplace. We will use Snowflake's query profile tool to understand the mechanics of query execution. We will learn about Snowflake's units of storage, known as *micro-partitions*. We will apply clustering to underlying tables to improve

query performance and identify queries that benefit from this type of optimization. We will also look at search optimization and describe the types of queries that benefit. Finally, we will learn how to identify queries that are candidates for optimization and share tips on improving query performance.

To illustrate the examples in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries. The bakery delivers these baked goods to small businesses, such as grocery stores, coffee shops, and restaurants in the neighborhood. The bakery is now looking to expand its production and sell the baked goods in major retail stores nearby.

The bakery data analysts will research nearby retail stores to find those potentially making the most sales. To perform the analysis, they will find and use retail data from a provider in the Snowflake Marketplace. They will get the data into their Snowflake account and execute queries to help them find nearby retail stores.

Because the dataset from the Snowflake Marketplace contains a large amount of data, queries using this data might execute more slowly than expected by the data analysts. Data engineers will analyze the queries that data analysts execute and find ways to improve query performance.

NOTE All code for this chapter is available in the accompanying GitHub repository in the Chapter_08 folder at https://mng.bz/5O6D.

8.1 Getting data from the Snowflake Marketplace

The bakery data analysts will start their analysis by researching data available on the Snowflake Marketplace and choosing an appropriate listing from a provider. They want a listing containing retail stores, their locations, and products. They are looking for free data for the initial analysis, which may be limited in scope or anonymized. After they are satisfied that the data meets their needs, they might upgrade to a paid listing for more detailed data.

Snowflake Marketplace

The Snowflake Marketplace is where you can find, try, and buy data published by third-party data providers. You can also share your data with consumers, either freely or by monetizing it. Types of data available on the Snowflake Marketplace include historical data for research and machine learning, current data such as weather or traffic conditions, consumer data collected by market research agencies, and much more.

More information about the Snowflake Marketplace is available in the Snowflake documentation at https://mng.bz/6Yo6.

For the examples in this chapter, we assume the bakery data analysts chose the "Sample Harmonized Data for Top CPG Retailers and Distributors" listing on the Snowflake Marketplace. This listing contains representative sample data from retail and distributor portals for consumer packaged goods (CPG) and food brands.

To get data from the Snowflake Marketplace, we will use the Snowsight user interface. We must use the `ACCOUNTADMIN` role to get data because this role has the `CREATE DATABASE` and `IMPORT SHARE` privileges required for this purpose (alternatively, we could use a custom role with these privileges granted).

In the Snowsight user interface, we will set `ACCOUNTADMIN` as the default role and click the Marketplace option under the Data Products menu in the left navigation pane. As shown in figure 8.1, we will see a window to use to search for listings. To search for free listings, we will click the "Instantly Accessible" filter, which displays listings that are instantly accessible, as opposed to listings that you must request from the provider. Then we will type "retail" in the search bar. Because the listings in the Snowflake Marketplace can change, with new listings added by providers all the time, your result list may differ somewhat from what is shown in figure 8.1.

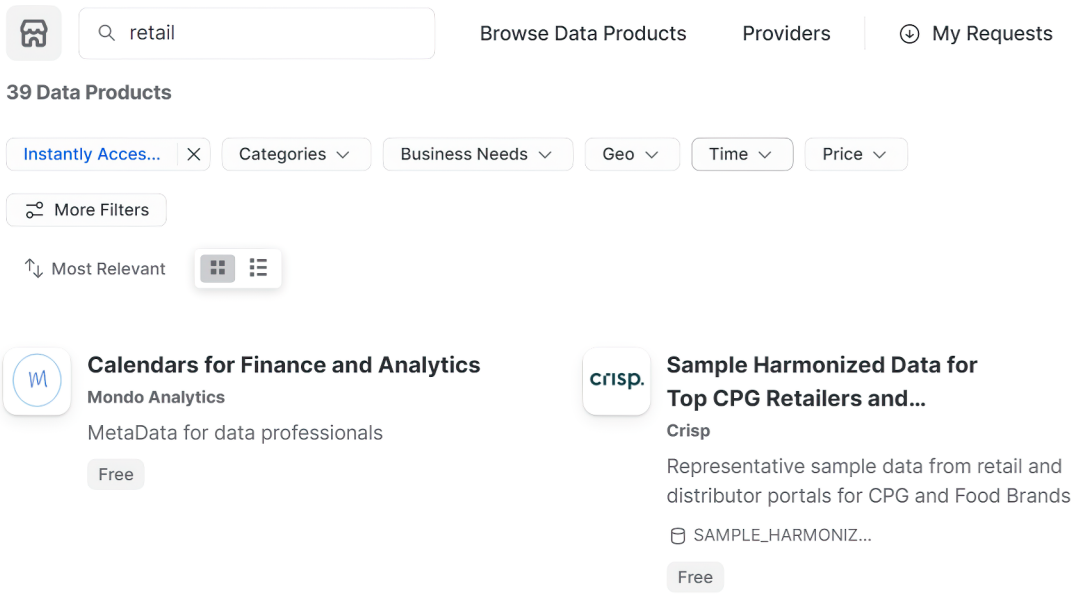


Figure 8.1 Search screen on the Snowflake Marketplace, displaying listings after filtering for "Instantly Accessible" and typing "retail" in the search box

In the filtered result, we should see the "Sample Harmonized Data for Top CPG Retailers and Distributors" listing provided by Crisp. This company collects and shares retail supply chain data. A new window opens after clicking this listing, as shown in figure 8.2. In this window, we can set the following options:

- The database name where the data will be available in our Snowflake account. In our example, we will enter `CPG_RETAILERS_AND_DISTRIBUTORS` as the database name.

- The roles, in addition to `ACCOUNTADMIN`, that will be able to access the data in the database. In our example, we will add `SYSADMIN` as the role that can access the data because we will use this role later to perform queries.

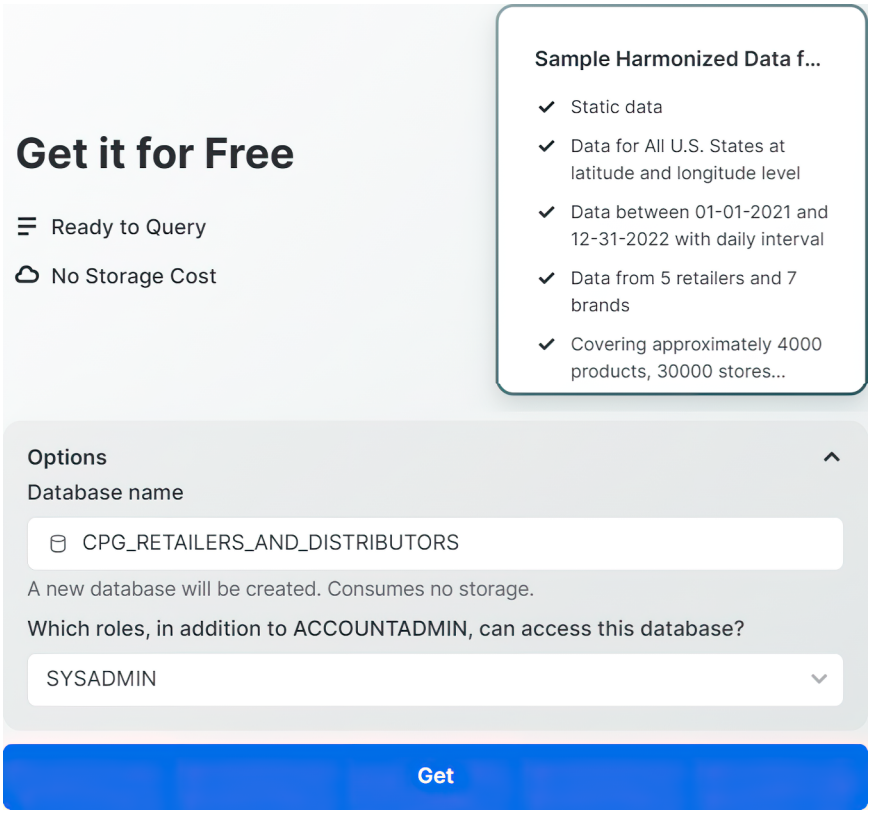


Figure 8.2 Snowflake Marketplace window where we specify the database name and the roles that can access the database before clicking the Get button to get the data into our Snowflake account

> **TIP** If you don't know which roles will access the data, you can omit adding them in this screen. You can grant the privilege later by executing the `grant imported privileges` command to the appropriate roles.

After clicking the Get button, we will see a new database named `CPG_RETAILERS_AND_DISTRIBUTORS` in our Snowflake account. This database contains shared data that does not physically reside in our Snowflake account, but you can query it like any other data in regular databases. When querying data shared from a provider, users often experience a delay in query response times because the query must retrieve the data from the provider's physical location.

For the exercises in this chapter, we will copy parts of the shared data into our Snowflake account so that the data will be physically available, and it will allow us to monitor query performance. In a real-world scenario, users may query the shared data directly or create a copy of the data in their Snowflake account, depending on the requirements.

8.2 Performing analysis of geographical data

After getting the data from the Snowflake Marketplace listing, we will continue with the analysis to identify retail stores in the bakery's vicinity. As we can see in the data description in figure 8.2, the database created from the listing contains data for all US states at latitude and longitude levels from January 1, 2021, through December 31, 2022.

The database contains the following tables:

- `HARMONIZED_RETAILER_INVENTORY_DC`—Representing unified product supply data that spans multiple distributors
- `HARMONIZED_RETAILER_INVENTORY_STORE`—Representing unified product supply data that spans multiple retailers
- `HARMONIZED_RETAILER_SALES`—Representing unified product point of sales data that spans multiple retailers

We will use the `HARMONIZED_RETAILER_SALES` table for the analysis because this table contains the store ID of each store, along with the latitude and longitude of the store's location.

Our fictional bakery doesn't have a location, so for the exercises in this chapter, let's choose a random location in the United States to use in the analysis. Let the bakery's location be Dayton, Ohio, with a latitude of 39.76 and a longitude of –84.19.

8.2.1 Snowflake's geography functions

We can use Snowflake's geography functions to calculate the distance between two points described by their latitude and longitude. We will need the following functions:

- `TO_GEOGRAPHY`—This takes a string as a parameter and returns a geography data type.
- `ST_DISTANCE`—This takes two geography data types as parameters and calculates the distance between them in meters.

NOTE For more information about Snowflake's geospatial data types and functions, refer to the Snowflake documentation at https://mng.bz/o0Wv.

Assuming GeoJSON as the default geography output format, a point is represented as a string in the format `'Point(<longitude> <latitude>)'`. Note that the longitude comes before the latitude in this format. We can record the point representing the bakery's location as

```
'Point(-84.19 39.76)'
```

Using the columns STORE_LATITUDE and STORE_LONGITUDE from the HARMONIZED_RETAILER_SALES table, we can construct the point that describes each store's location in this table by concatenating the values into a string:

```
'Point('||store_longitude||' '||store_latitude||')'
```

To use these points in Snowflake's ST_DISTANCE geography function, we must convert them into a geography data type by applying the TO_GEOGRAPHY function to each string representing a point.

We can now construct a query that selects the columns STORE_ID, STORE_LATITUDE, and STORE_LONGITUDE from the HARMONIZED_RETAILER_SALES table, converts the store's location into a geography data type, and calculates the distance between each store and the bakery's location. Since each store appears in the HARMONIZED_RETAILER_SALES table in multiple rows, we will use the DISTINCT keyword to select each store only once.

> **TIP** While developing the query, we can use the TOP <n> or LIMIT <n> keywords to limit the number of retrieved records so that the queries run quicker and consume fewer resources. We can remove the limitation once we are satisfied that the syntax is correct and the query works as expected.

The following snippet shows the query that selects each store along with its geographical location and uses the ST_DISTANCE function to calculate the distance between the store and the bakery, dividing the result in meters by 1,000 to get kilometers and limiting to five records:

```
select distinct
  store_id,
  store_longitude,
  store_latitude,
  TO_GEOGRAPHY(
    'Point('||store_longitude||' '||store_latitude||')'
  ) as store_loc_geo,
  ST_DISTANCE(
    TO_GEOGRAPHY('Point(-84.19 39.76)'), store_loc_geo
  )/1000 as distance_km
from CPG_RETAILERS_AND_DISTRIBUTORS.PUBLIC.HARMONIZED_RETAILER_SALES
limit 5;
```

The store's location in a geography data type

The distance between the store and the bakery in kilometers

Limits the result to five records

The result from this query is shown in table 8.1.

Table 8.1 Output from the query that selects data from the HARMONIZED_RETAILER_SALES table and calculates each store's geographical location and the distance from the bakery in kilometers, limited to five records (the column Store ID is abbreviated)

Store ID	Store latitude	Store longitude	Store geographical location	Distance in kilometers
5470922…	33.03	–97.06	{"coordinates":[–97.06,33.03],"type":"Point"}	1371.49
2774886…	32.99	–80.12	{"coordinates":[–80.12,32.99],"type":"Point"}	836.09

Table 8.1 Output from the query that selects data from the HARMONIZED_RETAILER_SALES table and calculates each store's geographical location and the distance from the bakery in kilometers, limited to five records (the column Store ID is abbreviated) *(continued)*

Store ID	Store latitude	Store longitude	Store geographical location	Distance in kilometers
4942067…	35.98	–84	{"coordinates":[–84,35.98],"type":"Point"}	420.65
5341435…	37.06	–76.41	{"coordinates":[–76.41,37.06],"type":"Point"}	741.04
3699756	32.58	–92.02	{"coordinates":[–92.02,32.58],"type":"Point"}	1,062.76

NOTE The ST_DISTANCE function returns the result in meters. If you prefer your result in miles, divide the result in meters by 1,609.

Now that we have the query that calculates the distance between the bakery and each store, we will create a table in our Snowflake account that will contain all the data in the HARMONIZED_RETAILER_SALES table in the shared database as well as the geography column representing each store's location and the calculated distance.

8.2.2 Copying data from the shared database

For the exercises in this chapter, we will continue using the BAKERY_DB database and the BAKERY_WH virtual warehouse we created in chapter 2. We will continue working using the SYSADMIN role. We will create a new schema named RETAIL_ANALYSIS in the BAKERY_DB database by executing the following commands:

```
use role SYSADMIN;
use warehouse BAKERY_WH;
use database BAKERY_DB;
create schema RETAIL_ANALYSIS;
use schema RETAIL_ANALYSIS;
```

Then we will create a table named RETAILER_SALES by selecting all columns from the HARMONIZED_RETAILER_SALES table in the shared database and adding columns STORE_ LOC_GEO as the store's location and DISTANCE_KM as the distance in kilometers by executing the following SQL statement:

```
create table RETAILER_SALES as
select *,
  TO_GEOGRAPHY(
    'Point('||store_longitude||' '||store_latitude||')'
  ) as store_loc_geo,
  ST_DISTANCE(
    TO_GEOGRAPHY('Point(-84.19 39.76)'), store_loc_geo
  )/1000 as distance_km
from CPG_RETAILERS_AND_DISTRIBUTORS.PUBLIC.HARMONIZED_RETAILER_SALES;
```

NOTE Because the table HARMONIZED_RETAILER_SALES contains more than 700 million records and the data is physically located outside of our Snowflake account, the query may take several minutes to complete.

We can continue the analysis once we have the data in our Snowflake account in the RETAILER_SALES table. We will look for stores in the bakery's vicinity by executing a query that selects individual stores using the DISTINCT keyword and ordering by distance in ascending order. We will also use the LIMIT keyword to limit the results to the top 100 stores because the bakery is looking for only a few stores in the vicinity, and there is no need to examine all the stores that are further away. The query that selects the top 100 stores in the bakery's vicinity is

```
select distinct
  store_id,
  distance_km
from RETAILER_SALES
order by distance_km
limit 100;
```

The output of this query is shown in table 8.2 (not all records are shown).

Table 8.2 Output when selecting individual stores in the bakery's vicinity ordered by the distance from the bakery in ascending order

Store ID	Distance in kilometers
744128088874118974	2.795243923
4817495515943480738	3.59519146
2782222913875409593	4.207261449
392366678147865718	4.207261449
1874009151117595125	4.207261449

To perform further analysis, the bakery data analysts want to examine a few nearby stores to determine which products they sell. They will look at each of the chosen stores individually. Let's select store ID 392366678147865718 and perform additional analysis.

We want to know which products are sold in the chosen store and in what quantities, which helps the bakery data analysts understand what sells and whether baked goods supplement the store's sales. The query that returns the total quantity of sold products in the chosen store is shown in the following listing.

Listing 8.1 Total quantity of sold products in the chosen store

```
select
  product_id,
  sum(sales_quantity) as tot_quantity
from RETAILER_SALES
where store_id = 392366678147865718
group by product_id;
```

The output of this query is shown in table 8.3 (not all records are shown).

Table 8.3 Output when selecting products and their sold quantities in a chosen store

Product ID	Total quantity
5655945986815814052	251
1752629343736427488	216
7200601990930849010	83

This query may take a while because we are working with large amounts of data, and the data analysts want to optimize the performance.

> **NOTE** We use simple illustration queries to keep the exercises in this chapter easy to follow. In a real-world scenario, queries performed by data analysts may be much more complex and take even longer to execute.

8.2.3 Viewing query execution parameters using the query profile

When investigating query performance, we often look at the query profile. This is a Snowflake tool that provides insight into the mechanics of query execution. It displays the processing plan for the query in a graphical representation and various query statistics relevant to the type of query that was executed.

For example, after executing the query from listing 8.1 in the Snowsight user interface, we will see the Query ID in the Query Details pane at the bottom right, as shown in figure 8.3.

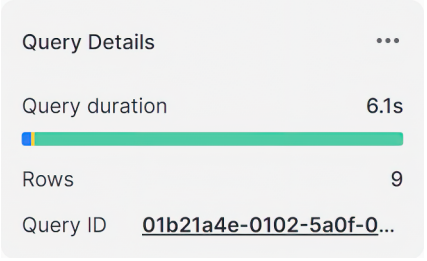


Figure 8.3 Query Details pane in the Snowsight user interface

We can click the hyperlink of the Query ID (alternatively, we can click the three dots at the top right of the pane and choose View Query Profile). A new window with the query profile appears. In this window, we see the graphical representation of the query execution plan, the Most Expensive Nodes pane, the Profile Overview pane, and the Statistics pane, as shown in figure 8.4.

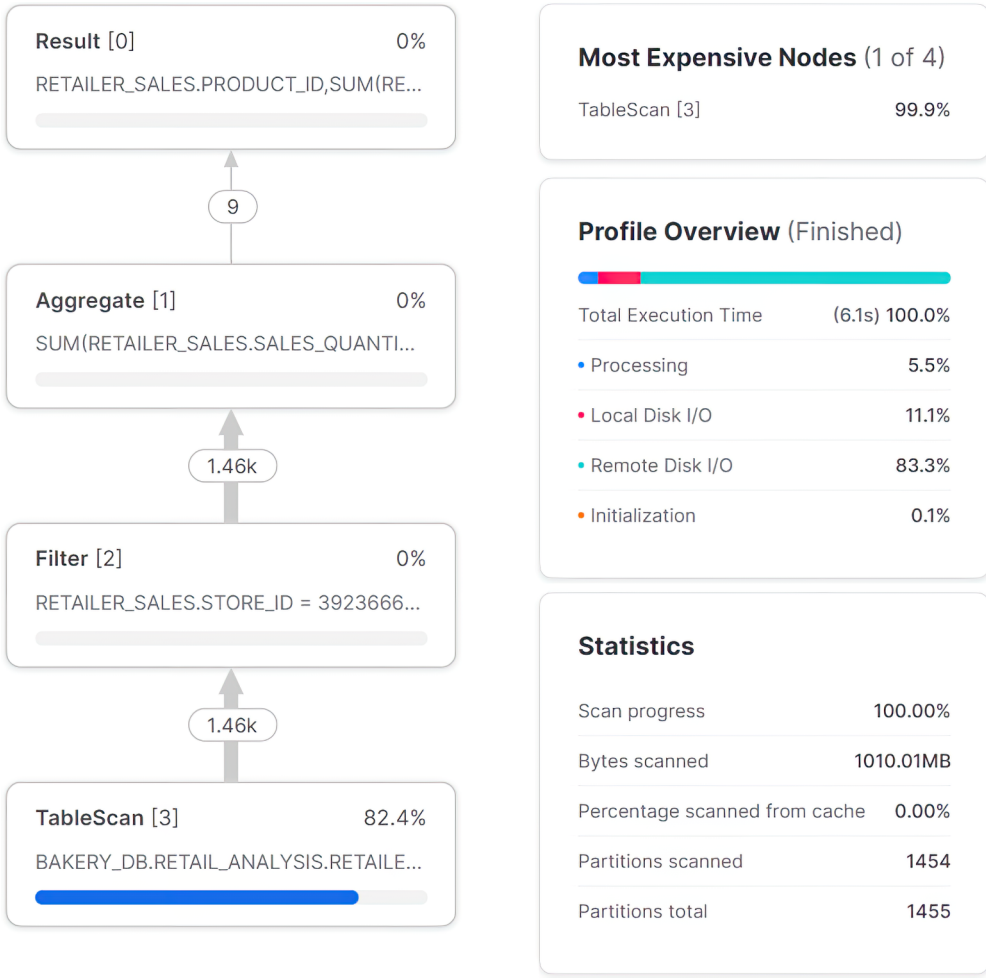


Figure 8.4 Query Profile showing the graphical representation of the query execution plan, the Most Expensive Nodes pane, the Profile Overview pane, and the Statistics pane

The query profile displays different panes and numerous statistics depending on the type and complexity of the query. Out of the many available statistics, we will focus on a few of the most common scenarios that have a significant effect on query performance:

- Micro-partition pruning
- Data caching
- Data spilling

In the remainder of this chapter, we will describe query optimization concerning micro-partition pruning. The following chapter describes query optimization with data caching and reducing data spilling.

NOTE For more information about the query profile tool, refer to the Snowflake documentation at https://mng.bz/n0j4.

8.3 Understanding Snowflake micro-partitions

Before optimizing query performance concerning micro-partition pruning, we must understand how Snowflake stores data in *micro-partitions*. A micro-partition is a contiguous unit of storage that contains somewhere between 50 MB and 500 MB of uncompressed data. The actual size of the stored data is smaller because data is always compressed in Snowflake.

Data in Snowflake micro-partitions is organized in a columnar fashion. This organization differs from traditional databases, which are often organized by rows because traditional databases are designed for reading and writing individual rows efficiently. Many modern databases, like Snowflake, organize data by column so that all data associated with each column is stored adjacently. These databases are optimized for efficiently reading and processing columns.

Snowflake automatically creates micro-partitions when data is inserted into tables. Data is often ordered along natural dimensions such as date, time, or geographical origin. This natural ordering is advantageous when writing data to micro-partitions because rows that share the same dimension value are colocated in micro-partitions whenever possible. Values can overlap between micro-partitions, which reduces skew in the data size stored in each micro-partition.

8.3.1 A conceptual example of micro-partitions

We will use a small amount of data to describe micro-partitioning concepts as an example. Let's use data consisting of nine rows and four columns, including the store ID, the sale date, the product, and the quantity, as shown in table 8.4.

Table 8.4 Example data to illustrate micro-partitions

Store ID	Product	Quantity	Sale date
A	Muffin	16	Jul 10
A	Bagel	12	Jul 10
A	Muffin	16	Jul 11
A	Bagel	18	Jul 11
B	Muffin	20	Jul 10
B	Bun	14	Jul 11
B	Muffin	20	Jul 11
C	Bun	50	Jul 10
C	Bagel	75	Jul 11

This data is stored across three micro-partitions, with the rows divided equally between each. The first three rows are stored in the first micro-partition, the following three rows in the second micro-partition, and the last three in the third micro-partition, as shown in figure 8.5.

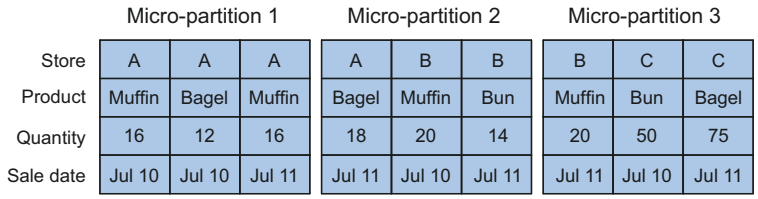


Figure 8.5 Conceptual representation of Snowflake micro-partitions using data from table 8.4

Because the data in table 8.4 is sorted by store, this is how it is sorted in the micro-partitions. Figure 8.5 shows that stores named A are stored in micro-partition 1 and partially in micro-partition 2. Stores named B are stored in micro-partitions 2 and 3. Stores named C are all stored in micro-partition 3. This type of storage is efficient when a query filters by store; for example, the query in listing 8.1 filters for a particular store. In this case, Snowflake only reads the micro-partitions that contain the store in the query filter.

Since Snowflake keeps track of micro-partition metadata, such as the range of values for each of the columns in the micro-partition, it optimizes query performance by reading only from the micro-partitions that contain the required data while excluding the remaining micro-partitions. This exclusion of unneeded partitions is called *pruning* in Snowflake.

Because storage in Snowflake is column-oriented, individual columns can be excluded if they are not used in a query. Thus, query performance can also benefit from pruning by column.

> **TIP** It is advisable to name the columns in the `SELECT` clause in Snowflake queries rather than selecting all columns using `SELECT *`. The reason is that Snowflake can take advantage of micro-partition pruning by reading only micro-partitions that contain the chosen columns instead of reading all micro-partitions when all columns are selected.

Let's investigate what happens when we write a query that filters the sample data by date instead of store. In the micro-partition structure shown in figure 8.5, we see that the two dates, July 10 and July 11, are both stored in all micro-partitions. If we filter for each date, the query must read all micro-partitions. For this type of query, it would be more efficient if the data were stored in micro-partitions sorted by date rather than store.

Figure 8.6 illustrates how micro-partitions could be organized if the data were sorted by the sale date before it was stored in a Snowflake table. In this case, July 10 appears only in micro-partitions 1 and 2, while July 11 appears in micro-partitions 2 and 3. Queries that filter by the sale date can take advantage of micro-partition pruning. In contrast, queries that filter by the store might be less efficient than in the previous micro-partition organization, where the data was organized by store.

	Micro-partition 1			Micro-partition 2			Micro-partition 3		
Store	A	A	B	C	A	A	B	B	C
Product	Muffin	Bagel	Muffin	Bun	Muffin	Bagel	Bun	Muffin	Bagel
Quantity	16	12	20	50	16	18	14	20	75
Sale date	Jul 10	Jul 10	Jul 10	Jul 10	Jul 11	Jul 11	Jul 11	Jul 10	Jul 11

Figure 8.6 Conceptual representation of Snowflake micro-partitions using data from table 8.4, ordered by the sale date

NOTE Snowflake's micro-partition storage is more complex than this conceptual example. For more information, refer to the Snowflake documentation at https://mng.bz/vJ2r.

8.3.2 Micro-partition pruning

Let's revisit the query profile we examined earlier in figure 8.4, specifically the Statistics pane shown again in figure 8.7.

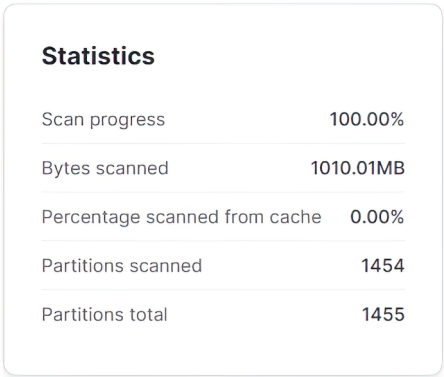


Figure 8.7 Statistics pane from the query profile in figure 8.4, showing statistics after executing the query that returns the total quantity of sold products in the chosen store

We see in figure 8.7 that the number of scanned partitions is 1,454, and the number of total partitions is 1,455. This tells us that Snowflake pruned only one micro-partition when executing the query. However, we know that the query filtered by store, resulting in a subset of data from the table, and it could have pruned many more micro-partitions.

We can verify this assumption by comparing the total number of rows in the table and the number of rows with the filtering by store applied using a query such as

```
select
  'Total rows' as filtering_type,
  count(*) as row_cnt
from retailer_sales
union all
select
  'Filtered rows' as filtering_type,
  count(*) as row_cnt
from retailer_sales
where store_id = 392366678147865718;
```

The output of this query is shown in table 8.5.

Table 8.5 Comparison of row counts in the entire table and when filtering for a single store

Filtering type	Row count
Filtered rows	1,460
Total rows	748,705,395

In such situations where we believe that the query could have taken advantage of micro-partition pruning but we see in the query profile that there was little micro-partition pruning, we can take measures to improve query performance. One way to do that is to order the data in the table by the column used in the query filter, thus reorganizing storage in micro-partitions. This can be accomplished with table clustering, as explained in the next section.

> **WARNING** Not all queries benefit from micro-partition pruning. For example, if a query must read all data from a table, then none of the micro-partitions can be pruned. Only queries that filter out a significant amount of data will see improved performance by micro-partition pruning.

8.4 Optimizing storage with clustering

When data is stored in micro-partitions so that values with the same key are stored in the same micro-partitions and few values overlap, it indicates that data is *well clustered*. Over time, as data in tables is inserted, updated, or deleted, clustering efficiency may degrade, which means that even after initial clustering, we must periodically recluster the data.

8.4.1 Viewing clustering information

We can use the `SYSTEM$CLUSTERING_INFORMATION` function to check how well data is clustered in a table for a given column. The function takes the following parameters:

- The name of the table for which we require clustering information.
- A list of columns for which we require clustering information. Even if we examine clustering information for just one column, we must enclose it in parentheses because the parameter expects a list. This parameter is optional if the table is already clustered because the clustering key is the default.
- An optional parameter with 10 as the default value, which indicates the number of clustering errors the function returns. We will not use this parameter in this exercise.

To check how well the `RETAILER_SALES` table is clustered for the `STORE_ID` column, we can execute the following command:

```
select SYSTEM$CLUSTERING_INFORMATION('retailer_sales', '(store_id)');
```

The output of this command might look something like the one shown in the following listing (some lines are omitted).

Listing 8.2 Output of the `SYSTEM$CLUSTERING_INFORMATION` function

```
"cluster_by_keys" : "LINEAR(store_id)",
  "total_partition_count" : 1455,
  "total_constant_partition_count" : 0,
  "average_overlaps" : 1454.0,
  "average_depth" : 1455.0,
  "partition_depth_histogram" : {
    "00000" : 0,
    ...
    "00016" : 0,
    "02048" : 1455
  },
  "clustering_errors" : [ ]
}
```

NOTE If you see a message saying "Clustering key columns contain high cardinality key `STORE_ID` which might result in expensive re-clustering," you can ignore it for now since we are learning about clustering. However, you should pay attention to such messages in a real scenario.

The interpretation of some of the values in the resulting output in listing 8.2 is as follows:

- The `total_constant_partition_count` is 0, meaning the table is not well clustered (a larger value means the clustering is better). A micro-partition is in a constant state when the clustering can't be improved further. In a well-clustered table, most micro-partitions are in a constant state.
- The `average_overlaps` value is almost the same as the `total_partition_count` value, which means that nearly all micro-partitions overlap, and consequently, the table is not well clustered (a lower value means that clustering is better).

- The `average_depth` value is the same as the `total_partition_count` value, meaning the table is not well clustered. A smaller value means the clustering is better.

NOTE More information about table clustering is available in the Snowflake documentation at https://mng.bz/4pyv.

When data in tables is not clustered well but users need better query performance that would benefit from micro-partition pruning, we can change how the table is clustered by adding a *clustering key*. A clustering key contains one or more table columns or expressions. After adding a clustering key to a table, Snowflake sorts and reorganizes data in micro-partitions behind the scenes.

8.4.2 Adding clustering keys to a table

Before creating a clustering key on a table, remember that clustering incurs cost, and consider whether clustering will be beneficial for your use case. The following are some criteria to keep in mind when deciding on clustering:

- The table contains a large number of micro-partitions, so pruning will be significant.
- Typical queries executed on the table select a small portion of rows, which means that queries will retrieve a small number of micro-partitions.
- Many queries executed by users filter by the same clustering key. Often, users filter data by a date or a time period, which might be a good candidate for a clustering key.
- The table has a large clustering depth.
- Queries that sort data on the clustering key will also perform well if the data is clustered.
- The number of distinct values in a column designated for clustering is large enough to enable micro-partition pruning and small enough to allow Snowflake to colocate rows with the same key in the same micro-partition.

In our previous example, we want to improve the query performance of the query in listing 8.1, which selects from the `RETAILER_SALES` table and filters for a single store ID. From the Statistics pane in the query profile in figure 8.7, we saw that micro-partition pruning was ineffective. We also saw from the output of the `SYSTEM$CLUSTERING_INFORMATION` function in listing 8.2 that the table is not well clustered by store ID.

To improve query performance of queries that filter for a small number of stores, we will add the `STORE_ID` column as the clustering key to the `RETAILER_SALES` table using the following command:

```
alter table RETAILER_SALES cluster by (store_id);
```

The clustering process may take some time because the table has more than 700 million records. Snowflake performs clustering in the background. There is no disruption, and

we can continue to execute queries using the table even while the clustering process is in progress.

8.4.3 Monitoring the clustering process

You can monitor the clustering process using the AUTOMATIC_CLUSTERING_HISTORY table function. This function returns the credits consumed, bytes updated, and rows updated each time a given table is clustered or reclustered within a specified date range.

The function returns results only when executed by a role with the MONITOR USAGE privilege. Since we are using the SYSADMIN role, we must grant this privilege first using the ACCOUNTADMIN role by executing the following commands:

```
use role ACCOUNTADMIN;
grant MONITOR USAGE ON ACCOUNT to role SYSADMIN;
use role SYSADMIN;
```

The AUTOMATIC_CLUSTERING_HISTORY function displays the clustering history within the last 12 hours unless otherwise specified by the date_range_start and date_range_end parameters. To view the clustering history for the RETAILER_SALES table, we must specify the fully qualified name, including the database and schema, to the table_name parameter. The following is a query that returns the clustering history for the RETAILER_SALES table within the last day:

```
select *
  from table(information_schema.automatic_clustering_history(
  date_range_start=>dateadd(D, -1, current_date),
  table_name=>'BAKERY_DB.RETAIL_ANALYSIS.RETAILER_SALES'));
```

When the initial clustering on the RETAILER_SALES table is completed, this query should return a row like what is shown in table 8.6.

Table 8.6 Output of the AUTOMATIC_CLUSTERING_HISTORY function

Start time	End time	Credits used	Num bytes reclustered	Num rows reclustered	Table name
2023-08-04 10:00:00	2023-08-04 11:00:00	2.205658123	71,192,126,358	2,178,694,056	BAKERY_DB. RETAIL_ANALYSIS. RETAILER_SALES

Once a table has a clustering key defined, Snowflake performs the initial clustering and periodic reclustering to ensure that the table is well clustered, even after data manipulation language operations such as inserts, updates, or deletes on table data. Each reclustering process is also included in the results of the AUTOMATIC_CLUSTERING_HISTORY function.

Now that the table is clustered, we can view the clustering information again using the SYSTEM$CLUSTERING_INFORMATION function:

```
select SYSTEM$CLUSTERING_INFORMATION('retailer_sales', '(store_id)');
```

The result of this function is shown in the following listing (some lines are omitted).

Listing 8.3 Output of the SYSTEM$CLUSTERING_INFORMATION function

```
{
  "cluster_by_keys" : "LINEAR(store_id)",
  "total_partition_count" : 1271,
  "total_constant_partition_count" : 25,
  "average_overlaps" : 7.6271,
  "average_depth" : 5.0433,
  "partition_depth_histogram" : {
    "00000" : 0,
    "00001" : 25,
    "00002" : 0,
    "00003" : 0,
    "00004" : 510,
    "00005" : 349,
    "00006" : 210,
    "00007" : 113,
    "00008" : 34,
    "00009" : 23,
    "00010" : 7,
    ...
  },
  "clustering_errors" : [ ]
}
```

As compared to the output of the SYSTEM$CLUSTERING_INFORMATION function in listing 8.2 that was executed before the table was clustered, we now observe a few changes:

- The total_partition_count decreased from 1,455 to 1,271 because the data is stored more efficiently.
- The total_constant_partition_count increased from 0 to 25, indicating better clustering.
- The average_overlaps decreased from 1,454 to 7.6271, indicating significantly fewer overlaps between micro-partitions.
- The average_depth decreased from 1,455 to 5.0433, again indicating significantly better clustering.

8.4.4 Viewing improved query execution after clustering

To see how table clustering improves query performance, let's execute the query that selects products and their quantities for a chosen store from listing 8.1 again:

```
select
  product_id,
  sum(sales_quantity) as tot_quantity
from RETAILER_SALES
where store_id = 392366678147865718
group by product_id;
```

After executing this query in the Snowsight user interface, let's view the query profile by clicking the Query ID hyperlink in the Query Details pane. In the query profile, we can review the Statistics and Profile Overview panes, as shown in figure 8.8.

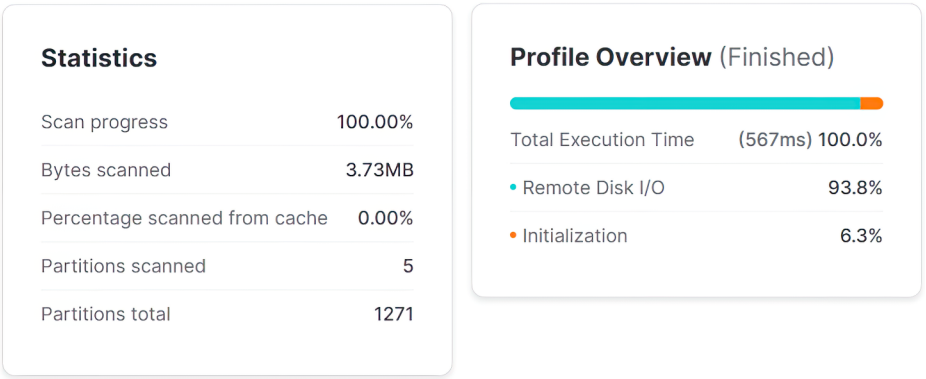


Figure 8.8 Statistics and Profile Overview panes in the query profile after the table is clustered

We can see that micro-partition pruning is very efficient now that the `RETAILER_SALES` table is clustered because only 5 micro-partitions were scanned out of 1,271.

Additionally, query execution time decreased from 6.1 seconds for the query executed before adding clustering (as seen in the Profile Overview pane in figure 8.4) to 567 milliseconds after executing the query on the clustered table. The decreased execution time is expected because fewer micro-partitions were scanned.

> **TIP** Adding clustering keys to tables and allowing Snowflake to maintain them automatically in the background incur cost. Before adding clustering keys to any table, consider the information presented in this chapter and in the Snowflake documentation to ensure that the types of queries most often performed by users and the distribution of the data in the table benefit from clustering.

8.5 Improving query performance with search optimization

The bakery data analysts want to continue with their analysis. The next analysis they wish to perform is to summarize the sales of a chosen competing product in all stores, regardless of the store's distance from the bakery. This will help them understand product sales distribution across the region and identify locations where they might achieve higher baked goods sales.

Because the data we are using is anonymized, we will guess that the competing product they are analyzing has an ID value of 4120371332641752996. In a real scenario, the bakery would purchase nonanonymized data that fits its needs. However, we can work with anonymized data to illustrate the exercises in this chapter.

NOTE The data in the Snowflake Marketplace listing may change over time. If you don't see a product with an ID value of 4120371332641752996 in your `RETAILER_SALES` data, select a different product with an ID value that exists in your data.

The following listing shows a query that returns the sum of the sold quantity of a chosen product in each store in the `RETAILER_SALES` table.

Listing 8.4 Total sold quantity of a chosen product in each store

```
select store_id, sum(sales_quantity) as tot_quantity
from RETAILER_SALES
where product_id = 4120371332641752996
group by store_id;
```

The output of this query is shown in table 8.7 (not all data is shown).

Table 8.7 Output of the query that returns the total sold quantity of a chosen product in each store

Store ID	Total quantity
2690155239960589744	3,858
7214951668181491887	819
1880121760328995629	1,695

After executing the query, we can review the query profile as we did earlier by clicking the Query ID hyperlink in the Query Details pane. In the Statistics pane in the query profile window, we should see no micro-partition pruning (the values of Partitions scanned and Partitions total are the same or nearly the same), which is expected. We know that the `RETAILER_SALES` table is clustered by the `store ID` column and not by the `product ID` column, which is the filter in this query.

To improve the query performance of queries that filter on the `product ID` column, we can't cluster the `RETAILER_SALES` again because we already clustered it by store ID. Instead, we can take advantage of Snowflake's *search optimization* service.

The search optimization service improves query performance when queries filter data. When search optimization is added to a table, Snowflake creates a *search access path* in the background. The search access path keeps track of which values of the table's columns reside in each of its micro-partitions. This helps with micro-partition pruning when querying the table using filters. Snowflake also maintains the search access path when data in the table is inserted, updated, or deleted.

8.5.1 Adding search optimization to a table

We can add search optimization to a table without specifying the columns to which it applies. In this case, Snowflake builds search access paths for all eligible columns in

the table. However, because building and maintaining the search access paths consumes cost, adding search optimization to select columns used most frequently in queries is more worthwhile.

> **NOTE** Search optimization requires Snowflake Enterprise edition or higher. The Snowflake documentation provides more information about the service at https://mng.bz/QV1Q.

In our example, to improve the performance of the query that filters by the product ID in listing 8.4, we can only add search optimization to the `RETAILER_SALES` table for the `product ID` column. We can do this by executing the following command:

```
alter table RETAILER_SALES add search optimization on equality(product_id);
```

After executing this command, creating the search access path may take some time. We can monitor the progress of building the search access path by looking into the output of the `SHOW TABLES` command:

```
show tables like 'RETAILER_SALES';
```

Selected columns from the output of this command are shown in table 8.8.

Table 8.8 Output of the `SHOW TABLES` command where we can monitor the search access path progress

Table name	Search optimization	Search optimization progress	Search optimization bytes
RETAILER_SALES	ON	100	17,498,624

Table 8.8 shows the search optimization progress at 100, which means that the search access path has been built.

8.5.2 Reviewing query performance after adding search optimization

We can now execute the query that returns the total sold quantity of a chosen product in each store from listing 8.4 again to verify whether search optimization improved performance. After executing the query, we see the Statistics pane in the query profile window, as shown in figure 8.9.

As we can see in the statistics pane, the number of scanned partitions is 955, while the total number of partitions is 1,271. This indicates that search optimization improved query performance by using micro-partition pruning.

> **WARNING** Not all queries benefit from search optimization, and not all tables or columns should have search optimization added.

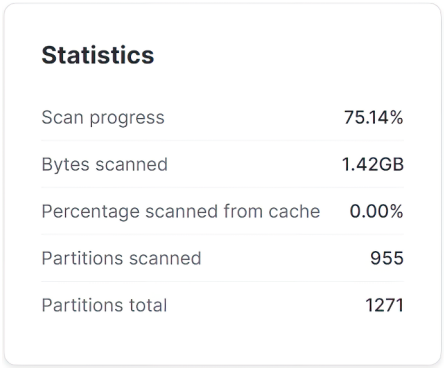


Figure 8.9 Statistics pane after executing the query in listing 8.4 and after adding search optimization to the `RETAILER_SALES` table

The following are some guidelines that can help you decide whether adding search optimization would improve query performance:

- Queries must have equality filters on the chosen columns. Equality means that the operator in the `WHERE` clause is an equal sign or an implicit equal sign such as the `IN` predicate (for example, `where product_id in (12345, 67890)`).
- The table in the query is large and has a large number of micro-partitions.
- The filter is selective, so the query returns a small number of rows.
- If the table is clustered, the queries that are candidates for optimization don't use the clustering key. Otherwise, clustering would improve query performance, and search optimization would not be needed.

TIP Additional tips on identifying queries that can benefit from search optimization are available in the Snowflake documentation at https://mng.bz/XVQp.

8.6 General tips for improving query performance

In addition to micro-partition pruning, which optimizes query performance, there are many more approaches to optimizing query performance in Snowflake. The following chapter describes some of these approaches, such as utilizing data caching and reducing data spilling.

8.6.1 Writing efficient SQL queries

Since Snowflake uses standard SQL, optimizing Snowflake query performance requires following best practices for writing efficient SQL queries that are generally applicable.

The following are some all-purpose SQL performance optimization tips for writing SQL queries in Snowflake. Remember that not all recommendations apply in all situations, so use them based on the relevance of the query you are writing or optimizing:

- Retrieve only the data you need in a query by applying filtering as early as possible to reduce the amount of data a query must process.

- Retrieve the data from the tables first, applying any required joining and filtering; then perform more complex operations on the reduced volume of data.
- To simplify the queries, avoid too many nested subqueries. Consider whether a single join is sufficient when a query joins with the same table more than once.
- Remove unnecessary `ORDER BY` clauses in subqueries.
- Use `UNION ALL` instead of `UNION` if the results of the queries are disjointed.
- Don't use `DISTINCT` "just in case" when each row is unique.
- In data warehousing use cases, we often deduplicate data—for example, to get the last record of the day or to remove duplicate values in descriptions by taking the first value. In such cases, use the `ROW_NUMBER()` analytical function and the `QUALIFY` clause.
- Use the query profile tool to understand the mechanics of query execution.

8.6.2 Identifying queries that are candidates for optimization

Data engineers must sometimes look for bottlenecks in data pipeline execution and identify queries that are candidates for optimization. They can search for the longest-running queries by examining the query history.

> **TIP** The Snowflake documentation explains how to examine past query performance. You can find this information at https://mng.bz/yo6y.

Let's write a query that selects previously executed queries from the query history and orders them by the longest elapsed time in descending order. We can get this information from the `QUERY_HISTORY` view in the `SNOWFLAKE` database. Because the `SYSADMIN` role that we are using doesn't have access to this table, we must grant the `GOVERNANCE_VIEWER` database role, which allows access to this table to the `SYSADMIN` role using the following commands:

```
use role ACCOUNTADMIN;
grant database role SNOWFLAKE.GOVERNANCE_VIEWER to role SYSADMIN;
use role SYSADMIN;
```

Then we can use the `SYSADMIN` role again to write a query that retrieves previously executed queries within the last day, ordered by the query elapsed time in descending order, limited to the top 50 rows:

```
select
  query_id,
  query_text,
  partitions_scanned,
  partitions_total,
  total_elapsed_time
from SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY
where TO_DATE(start_time) > DATEADD(day,-1,TO_DATE(CURRENT_TIMESTAMP()))
order by total_elapsed_time desc
limit 50;
```

The results of this query are shown in table 8.9 (not all rows are shown, and some columns are abbreviated).

Table 8.9 Output of the query that retrieves the longest-running previously executed queries

Query ID	Query text	Partitions scanned	Partitions total	Total elapsed time
01b21fc6-0102-5a0f-0002	`select product_id, sum(sales_quantity) from…`	1,271	1,271	10,883
01b2217d-0102-59c0-0002	`select store_id, sum(sales_quantity) from…`	955	1,271	8,831
01b21fa0-0102-5820-0002	`select product_id, sum(sales_quantity) from…`	1,454	1,455	7,880
01b21fc7-0102-59c0-0002	`select product_id, sum(sales_quantity) from…`	1,271	1,271	4,171

The output helps data engineers identify the longest-running queries by showing the total elapsed time. The number of scanned and total partitions is also shown, which helps them understand whether query pruning was applied. The query text and the query ID of the executed query can be used to explore the query in more detail in the query profile. This is a good starting point where data engineers can look for ways to improve query performance when asked to optimize data pipelines.

Summary

- The Snowflake Marketplace is where you can find, try, and buy data published by third-party data providers. We can use the Snowsight user interface to get data from a Marketplace listing into our Snowflake account.
- Snowflake's geography functions, such as `TO_GEOGRAPHY` and `ST_DISTANCE`, calculate the distance between two points described by their latitude and longitude.
- We can use the `TOP <n>` or `LIMIT <n>` keywords to limit the number of retrieved records in a query to execute more quickly and consume fewer resources.
- When investigating query performance, we often look at the query profile. This is a Snowflake tool that provides insight into the mechanics of query execution. It displays the processing plan for the query in a graphical representation and various query statistics relevant to the type of query that was executed.
- A micro-partition is a contiguous unit of storage in Snowflake that contains somewhere between 50 MB and 500 MB of uncompressed data. Snowflake automatically creates micro-partitions when data is inserted into tables.

- Since Snowflake keeps track of micro-partition metadata, such as the range of values for each of the columns in the micro-partition, query performance can be optimized by reading only from the micro-partitions that contain the required data while excluding the remaining micro-partitions. This exclusion of unneeded partitions is called pruning.
- When data is stored in micro-partitions so that values with the same key are stored in the same micro-partitions and few values overlap, it indicates that data is well clustered. We can use the `SYSTEM$CLUSTERING_INFORMATION` function to check how well data is clustered in a table for a given column.
- When data in tables is not clustered well but users need better query performance that would benefit from micro-partition pruning, we can change how the table is clustered by adding a clustering key.
- You can monitor the clustering process using the `AUTOMATIC_CLUSTERING_HISTORY` table function. This function returns the credits consumed, bytes updated, and rows updated each time a given table is clustered or reclustered within a specified date range.
- The search optimization service improves query performance when queries filter data. When we add search optimization to a table, Snowflake creates a search access path in the background. The search access path keeps track of which values of the table's columns might be found in each of its micro-partitions. This helps with micro-partition pruning when querying the table using filters.
- Since Snowflake uses standard SQL, optimizing Snowflake query performance requires following best practices for writing efficient SQL queries that are generally applicable.
- Data engineers must sometimes look for bottlenecks in data pipeline execution and identify queries that are candidates for optimization. They can search for the longest-running queries by examining the query history.

9 Controlling costs

This chapter covers

- Understanding Snowflake costs
- Sizing virtual warehouses
- Using persisted query results
- Optimizing query performance to reduce spilling
- Optimizing performance with data caching
- Reducing query queuing
- Monitoring compute consumption

Data engineers must understand how Snowflake incurs costs so they can build cost-effective data pipelines. Cost and performance are often intertwined. In some cases, better performance may result in a higher cost. For example, upgrading a virtual warehouse to a larger size means better performance, but the cost of this can quickly add up. In other cases, such as long-running queries, poor performance could result in a higher cost because the query causes the virtual warehouse to be active longer. One of the responsibilities of data engineers is to monitor warehouse consumption and use the findings to strike a good balance between improving performance and controlling costs.

In this chapter, we will write queries using large amounts of data from the Snowflake Marketplace. We will discover what contributes to Snowflake's cost and how to monitor credit consumption. We will explain Snowflake virtual warehouses and how we can resize them to optimize performance. We will learn to minimize spilling during query execution and use persisted query results. We will also describe strategies to reduce query queuing and concurrently running queries.

To illustrate the examples in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries. The bakery delivers these baked goods to small businesses, such as grocery stores, coffee shops, and restaurants in the neighborhood. The bakery is now looking to expand its production and sell the baked goods in major retail stores nearby.

In chapter 8, the bakery data analysts researched nearby retail stores to find those potentially making the most sales using retail data from a provider in the Snowflake Marketplace. They got the data into their Snowflake account and executed queries to help them find nearby retail stores.

Because the table populated from the Snowflake Marketplace contains a large amount of data, queries using this data often execute slower than expected by the data analysts. Data engineers analyzed the queries the data analysts executed and suggested query performance optimization techniques such as clustering or search optimization.

In this chapter, the data engineers will look for additional ways to improve query performance, primarily by monitoring and optimizing virtual warehouses while also considering the cost effect of performance optimization.

NOTE All code for this chapter is available in the accompanying GitHub repository in the Chapter_09 folder at https://mng.bz/M1Ao.

9.1 Understanding Snowflake costs

Before delving into the details of Snowflake's cost, let's briefly discuss where controlling cost is relevant for data engineers. Data engineering pipelines comprise core components that include extracting data from source systems, ingesting the data into Snowflake, performing transformations, and presenting the data to downstream consumers. These components are explained in more detail in chapter 1. Figure 9.1 shows a graphical representation of the core components.

When building data pipelines, data engineers encounter different types of workloads. Ingesting data from source systems is often executed in batch, typically overnight. Data ingestion performance is not crucial in such situations unless it takes too long to complete in a single night. Sometimes users want data to be ingested more frequently than overnight—for example, every few hours or minutes. Performance is more important in this scenario because the pipeline must finish executing the previous ingestion before it starts executing the next one.

When transforming and presenting data to downstream consumers, data engineers must pay attention to the user performance requirements and meet their needs.

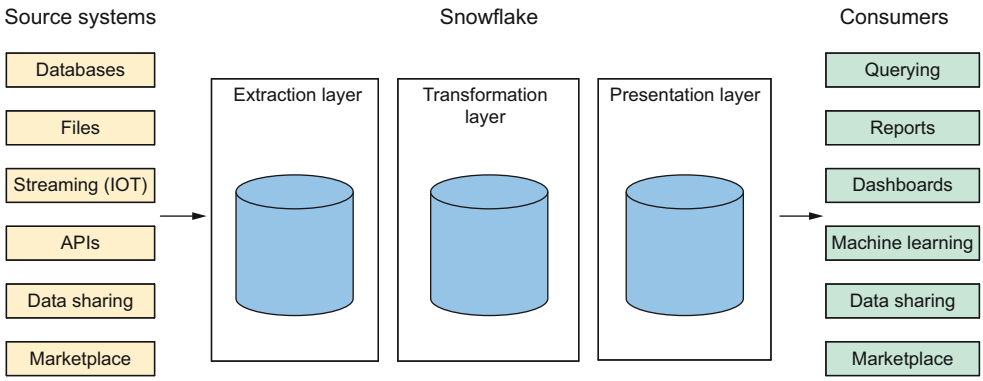


Figure 9.1 Data engineering core components with Snowflake include extracting data from source systems, ingesting the data into Snowflake, performing transformations, and presenting the data to downstream consumers.

Performance improvements usually have a cost implication in Snowflake. This is why data engineers need to understand how costs are incurred.

9.1.1 Total Snowflake cost

Snowflake uses a consumption-based pricing model, where users pay for storage and compute resources. The total cost is calculated as the sum of the costs in the following categories:

- *Compute resources cost*—Using compute resources within Snowflake consumes Snowflake credits. The factors contributing to compute cost are described in more detail in the following sections of this chapter.
- *Storage resources cost*—Storing data in the Snowflake data cloud consumes storage resources. Objects, such as data in database tables and materialized views, staged data files, search optimization paths, and historical data preserved for time travel, contribute to the overall storage cost.
- *Data transfer resources cost*—While Snowflake does not charge for ingesting data into your account, it does charge for exporting data. Data transfer occurs when you unload data to a cloud storage provider, replicate data to a secondary database for high availability or disaster recovery, use external functions, or access external network locations. Since most of the data engineers' work involves building data pipelines that ingest data, we will not discuss data transfer costs further.

NOTE Using external network access as described in chapter 7 incurs data transfer resources costs. Therefore, review the cost structure if you use it in your pipelines. For more information about data transfer costs, refer to the Snowflake documentation at https://mng.bz/aVYo.

9.1.2 Compute resources cost

Snowflake's compute resources include user-managed compute resources represented by virtual warehouses and Snowflake-managed resources. The total cost for using compute resources is calculated as the sum of the costs in the following categories:

- *Virtual warehouse compute*—Virtual warehouses are computing resources that allow users to process and analyze data. They operate independently and can be scaled up or down based on workload requirements. Snowflake bills virtual warehouses for the time they are actively working but not for the time they are suspended. The billing is on a per-second basis with a 60-second minimum.
- *Serverless compute*—Snowflake features, such as search optimization, continuous data loading with Snowpipe, or database replication and failover, use Snowflake-managed compute resources rather than user-managed virtual warehouses. Snowflake automatically resizes and scales these resources up or down as needed.
- *Cloud services compute*—The cloud services layer of the Snowflake architecture performs services such as authentication, metadata management, query parsing, serving cached data, etc. Snowflake bills for the compute resources in this layer only if they exceed 10 percent of the total daily compute consumption.

NOTE For more information about the factors contributing to compute cost, refer to the Snowflake documentation at https://mng.bz/gAjx.

9.1.3 Virtual warehouse credits

A Snowflake virtual warehouse is characterized by its size and additional properties that define and automate warehouse activity. The warehouse size determines the compute resources for executing queries and other processing operations. The cost of the processing time spent by each virtual warehouse is measured in credits according to its size.

NOTE Throughout this chapter, we express virtual warehouse compute cost using credits as the unit of measure. The cost of a credit in monetary value in your currency varies depending on your Snowflake edition, the cloud provider, the cloud provider region, and other commercial arrangements with Snowflake. You can find the Snowflake pricing guide at https://mng.bz/eVKP.

Organization usage data

The Snowflake administrator can find details about the consumption in monetary value for all accounts in the organization in the `ORGANIZATION_USAGE` views in the `SNOWFLAKE` database. More information is available in the Snowflake documentation at https://mng.bz/pxr2.

A list of supported Snowflake warehouse sizes and their credit consumption per hour is shown in figure 9.2.

Virtual warehouse size	X-small	Small	Medium	Large	X-large
Credits per hour	1	2	4	8	16

Virtual warehouse size	2X-large	3X-large	4X-large	5X-large	6X-large
Credits per hour	32	64	128	256	512

Figure 9.2 Snowflake virtual warehouse sizes and their credit consumption per hour

Figure 9.2 indicates that each subsequent warehouse consumes twice the number of credits as the previous one. The compute resources provided by each subsequent warehouse are also roughly twice as much as the previous one, which results in faster query execution time, particularly for complex queries that require a significant amount of resources.

Virtual warehouses, by default, consist of a single cluster of compute resources. When more computing power is needed, we can configure warehouses as *multicluster*. With multicluster warehouses, Snowflake adds clusters to provide a larger pool of compute resources as needed. Clusters are deactivated when no longer needed according to the parameters specified when creating the multicluster warehouse.

NOTE For more information about virtual warehouses, refer to the Snowflake documentation at https://mng.bz/OmRo.

Snowpark-optimized warehouses

In addition to standard virtual warehouses, Snowflake provides Snowpark-optimized warehouses designed for workloads with large memory requirements, such as machine learning use cases. For more information about Snowpark-optimized warehouses, refer to the Snowflake documentation at https://mng.bz/YVXK.

9.2 Sizing virtual warehouses

Let's work with an example to understand how increasing the virtual warehouse size improves query performance. We will continue using the `BAKERY_DB` database we created in chapter 2 for the exercises in this chapter. We will use the `RETAILER_SALES` table in the `RETAIL_ANALYSIS` schema we created in chapter 8 and continue using the `SYSADMIN` role. We can execute the following commands to use these objects:

```
use role SYSADMIN;
use database BAKERY_DB;
use schema RETAIL_ANALYSIS;
```

We will create a set of virtual warehouses of increasing sizes to compare query performance. We can do this in the Snowsight user interface under the Admin menu or execute commands. Like other data engineering tasks, we will create virtual warehouses by executing commands because creating or altering virtual warehouses is sometimes built into data pipelines. Therefore, it's convenient to use commands.

To create four virtual warehouses named BAKERY_WH_<warehouse_size>, sized from extra small to large, using default values for all remaining parameters, we will execute the following commands:

```
create warehouse BAKERY_WH_XSMALL with warehouse_size = 'xsmall';
create warehouse BAKERY_WH_SMALL with warehouse_size = 'small';
create warehouse BAKERY_WH_MEDIUM with warehouse_size = 'medium';
create warehouse BAKERY_WH_LARGE with warehouse_size = 'large';
```

TIP You can specify additional parameters when creating virtual warehouses, such as when to suspend a warehouse due to inactivity, the number of clusters if you are creating a multicluster warehouse, and a few more. For details of the parameters, see the Snowflake documentation at https://mng.bz/GNlR.

Next we need a query to compare performance with warehouses of different sizes. Suppose we choose a simple query that runs quickly. In that case, we might not be able to discern whether it performs differently in different warehouses because the amount of consumed resources would be small enough that warehouses of all sizes would process it quickly and efficiently. Instead, we want a complex query that processes a large amount of data.

We will use the RETAILER_SALES table we created in chapter 8, which is large enough for this exercise since it contains more than 700 million records. The data in this table includes retail stores, their distance from the bakery's location, and sales data for all products sold in each store.

Say the bakery wants to sell its baked goods only in larger retail stores that sell at least 100 different products. The bakery data analysts will construct a query that retrieves the total quantity of products sold in each store, sorted by the distance from the bakery's location, limited to the larger retail stores. This query is shown in listing 9.1.

We will execute the query using each virtual warehouse we created earlier, starting from the smallest. To use the extra-small warehouse, we can choose the BAKERY_WH_XSMALL warehouse from the drop-down list in the context menu at the top right of the worksheet in the Snowsight user interface. Alternatively, we can use the extra-small warehouse by executing the following command:

```
use warehouse BAKERY_WH_XSMALL;
```

Then we will execute the query in the following listing using this virtual warehouse.

Listing 9.1 A query for comparing warehouse performance

```
select
  store_id,
  distance_km,
  product_id,
  sum(sales_quantity) as total_quantity
from RETAILER_SALES
where store_id in (
```

Filter that selects stores that sell more than 100 different products

```
  select store_id
  from (
    select store_id,
      count(distinct product_id) as product_cnt
    from RETAILER_SALES
    group by store_id
    having product_cnt > 100
  )
)
group by store_id, distance_km, product_id
order by distance_km;
```

The output produced by the query from listing 9.1 is shown in table 9.1 (not all data is shown).

Table 9.1 Output of the query from listing 9.1 that selects the total sold quantity of each product in each store, includes only stores that sell more than 100 different products, and sorts the output by the distance from the bakery's location

Store ID	Distance KM	Product ID	Total quantity
4162341910460251133	7.147963876	5043292653147024261	40
4162341910460251133	7.147963876	5680980132714865975	60
4162341910460251133	7.147963876	737353455755519867	103

After executing the query, we can examine the query profile, as in chapter 8. To recap, the query profile is a Snowflake tool that provides insight into the mechanics of query execution. It displays the processing plan for the query in a graphical representation and various query statistics relevant to the type of query.

For example, after executing the query from listing 9.1 in the Snowsight user interface using the `BAKERY_WH_XSMALL` virtual warehouse, we will see the Query ID in the Query Details pane at the bottom right, as shown in figure 9.3.

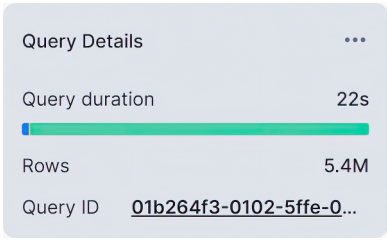


Figure 9.3 Query Details pane in the Snowsight user interface

Let's view the query profile by clicking the Query ID hyperlink in the Query Details pane (alternatively, we can click the three dots at the top right of the pane and choose View Query Profile). In the query profile that opens in a new browser tab, we will review the Profile Overview and the Statistics panes, as shown in figure 9.4.

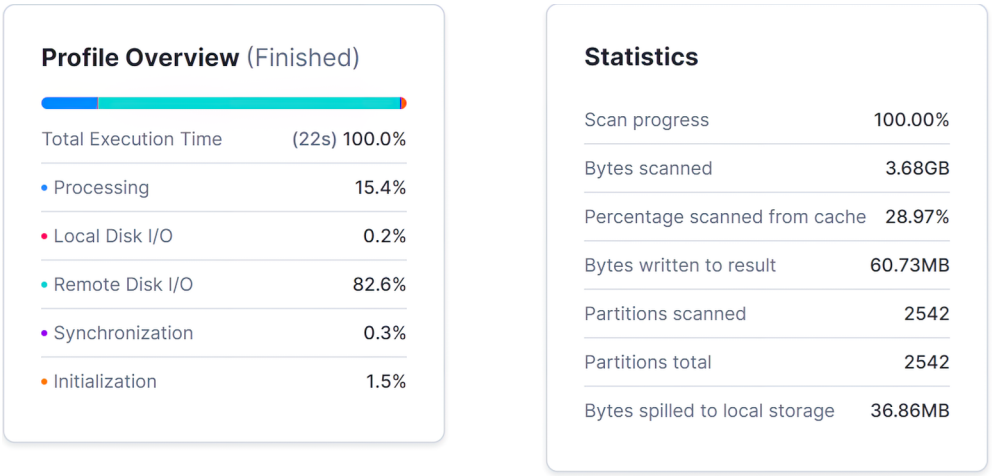


Figure 9.4 The Profile Overview and the Statistics panes from the query profile after executing the query from listing 9.1 using the `BAKERY_WH_XSMALL` virtual warehouse

Let's take a note of two parameters:

- The Total Execution Time from the Profile Overview pane, which is 22 seconds
- The Bytes spilled to local storage from the Statistics pane, which is 36.86 MB

9.2.1 Using persisted query results

To compare query statistics using a different virtual warehouse, we will again execute the query from listing 9.1. This time, we will use the next-larger warehouse named `BAKERY_WH_SMALL`, whose size is Small. To use this warehouse, we will execute the following command:

```
use warehouse BAKERY_WH_SMALL;
```

Then we will execute the query from listing 9.1. This query will likely finish executing almost instantaneously. To understand why this occurred so quickly, we can view the query profile after executing the query by clicking the Query ID hyperlink in the Query Details pane like previously. This time, instead of the graphical representation of the query execution, we see just one step, as shown in figure 9.5. This step indicates that the query results were reused.

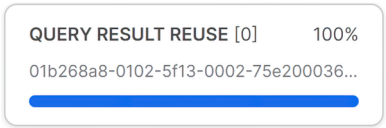


Figure 9.5 Query profile after executing the query from listing 9.1 a second time, indicating that the query results were reused

This query profile explains that Snowflake persists the results of queries for a certain period of time (usually 24 hours, with some exceptions) after they are executed. This helps to optimize performance when the same query is executed many times, such as when many users run the same report or perform the same data transformation. Because the results are persisted, Snowflake retrieves them instead of re-executing the query.

There are some limitations to when Snowflake reuses persisted query results. For example, the query must be exactly the same; even uppercase or lowercase letters must match. The data in the underlying tables from which the query reads must not have changed to guarantee the same result. If the data has changed, Snowflake must re-execute the query and will not use the persisted results.

> **NOTE** For more information about reusing persisted query results, refer to the Snowflake documentation at https://mng.bz/znoX.

Using persisted query results can significantly improve query performance when the same query executes many times and the underlying data doesn't change frequently. Because the query is not re-executed, it doesn't require an active virtual warehouse and doesn't consume any credits.

9.2.2 Comparing query statistics between differently sized warehouses

We want to compare query profile statistics when executing the same query using warehouses of different sizes. In that case, we must temporarily disable using persisted query results to continue the exercise. We will set the `USE_CACHED_RESULT` parameter to `FALSE` in the current session by executing the following command:

```
alter session set use_cached_result = FALSE;
```

After executing this command, query results will not be reused in the current session, which means that Snowflake will execute the query every time, even if the query is exactly the same as previously and the results have already been persisted.

> **WARNING** Normally, to ensure the best possible performance, using persisted query results in Snowflake is desirable. Therefore, we want the `USE_CACHED_RESULT` parameter to be set to `TRUE`, as is the default. We have only set it to `FALSE` in this exercise so that we can examine the query statistics each time we execute the query.

After disabling query result reuse by setting the `USE_CACHED_RESULT` to `FALSE` in the current session, we can continue to execute the query from listing 9.1 using differently sized warehouses. We should already be using the `BAKERY_WH_SMALL` virtual warehouse that we selected earlier. We can now execute the query again from listing 9.1 and examine the query profile.

Taking note of the same statistics as previously, the value of the Total Execution Time statistic is 10 seconds, and the value of the Bytes spilled to local storage statistic is 38.06 MB.

Repeating the process, using the `BAKERY_WH_MEDIUM` and `BAKERY_WH_LARGE` virtual warehouses, executing the query from listing 9.1 each time, and taking note of the Total Execution Time and the Bytes spilled to local storage statistics, we can gather the results in a table. We will add the Credits per hour for each warehouse size to the table, as shown in table 9.2.

NOTE Your results may vary from those shown in table 9.2, but the overall proportions between the results and the interpretation should be similar.

Table 9.2 Comparison of query profile statistics after executing the query from listing 9.1 using differently sized virtual warehouses

Warehouse size	Credits per hour	Total execution time in seconds	Bytes spilled to local storage in MB
X-small	1	22	36.86
Small	2	10	38.06
Medium	4	6	0
Large	8	4	0

Using the values of the Credits per hour and the Total execution time in seconds from table 9.2., we can do a rough calculation to determine the number of credits consumed by the query using each warehouse. Since the Total execution time is expressed in seconds, we must divide the Credits per hour value by 3,600 to get the credits per second and multiply this value by the Total execution time in seconds to get the Credits consumed value, as shown in table 9.3.

Table 9.3 Calculation of the credits consumed by the query from listing 9.1 using differently sized virtual warehouses

Warehouse size	Credits per hour	Total execution time in seconds	Credits consumed
X-small	1	22	(1/3,600) * 22 = 0.0061
Small	2	10	(2/3,600) * 10 = 0.0056
Medium	4	6	(4/3,600) * 6 = 0.0067
Large	8	4	(8/3,600) * 4 = 0.0089

As shown in table 9.3, the Credit consumed was the lowest (0.0056) when the query from listing 9.1 was executed using a small warehouse. However, this is just an approximation of the cost calculation. We must remember that warehouses are billed for the initial 60 seconds whenever they start, so a larger warehouse costs more for this fixed period.

In a real-life scenario, multiple users usually execute queries simultaneously using the same warehouse, meaning the queries compete for resources. A query may run longer in a multiuser environment than in our experiment, where we were the only users of the warehouse. Since virtual warehouses consume credits for the time they are active, regardless of how many queries they execute, the cost of the credits consumed applies to all queries executed simultaneously. We can't attribute the warehouse cost to a single query.

> **TIP** Determining the optimal warehouse size is often an exercise of trial and error, especially when many queries execute simultaneously. Usually you start with a smaller-sized warehouse and use it for some time, and then increase the size to the next-larger size and compare the cost and performance to decide whether the performance gains outweigh the increased cost.

A general rule of thumb when selecting an initial warehouse size, as recommended by Snowflake, is

- In small-scale development and testing environments, choose smaller warehouse sizes, such as extra small, small, or medium.
- In large-scale production environments, choose larger warehouse sizes, such as large, extra large, or larger.

After choosing an appropriate warehouse size, you should continue to monitor performance and credit consumption. You can always change the size of a warehouse as your workloads change over time.

9.2.3 Optimizing query performance to reduce spilling

In addition to Total execution time, table 9.2 records the value of the Bytes spilled to local storage statistics after executing the query from listing 9.1 using differently sized warehouses. As we can see from the recorded values in the table, there is spilling when using extra-small and small warehouse sizes, but no spilling with medium and large warehouses.

Spilling happens when a warehouse runs out of memory while executing a query, causing it to write data to local disk storage. Consequently, query performance decreases because it must fetch data from the disk storage instead of memory. When you execute large and complex queries, even local disk storage may run out, in which case Snowflake writes to remote storage in the cloud provider, decreasing performance even more.

To improve query performance due to spilling, remember that spilling is often caused by grouping and sorting in queries that select data from large tables. If possible, try to rewrite such queries to reduce the amount of data they must process. For example, limiting the result set by using a filtering condition that selects fewer records can reduce spilling.

Let's illustrate this scenario with an example. We will use the query from listing 9.1 again and the extra-small warehouse to execute it. This query selects the total quantity

of each product sold in each store, includes only stores that sell more than 100 different products, and sorts the output by the distance from the bakery's location. The bakery data analysts might only be interested in some stores within the country located less than 1,000 kilometers away from the bakery's location. They can amend the query by adding a filter that limits the stores to those whose distance is less than 1,000 kilometers, as shown in the following listing.

Listing 9.2 Amended query from listing 9.1 with filtered data

```
use warehouse BAKERY_WH_XSMALL;

select
  store_id,
  distance_km,
  product_id,
  sum(sales_quantity) as total_quantity
from RETAILER_SALES
where store_id in (
  select store_id
  from (
    select store_id,
      count(distinct product_id) as product_cnt
    from RETAILER_SALES
    where distance_km < 1000
    group by store_id
    having product_cnt > 100
  )
)
group by store_id, distance_km, product_id
order by distance_km;
```

Filter that selects stores less than 1,000 kilometers away

After executing the query with the filter, we can again examine the Profile Overview and Statistics panes in the query profile, as illustrated in figure 9.6. As expected, because the query must process less data, the Total Execution Time value is less than when the query was executed without the filter; it decreased from 22 seconds to 16 seconds.

The value of the Bytes spilled to local storage statistic also decreased from 36.86 MB to 18.53 MB, indicating that less data was spilled to local storage because there was less data for the query to process.

We can't always limit the amount of data in a query to reduce spilling. Sometimes, the use case requires the query to execute using all data, and limiting the amount of data is impossible. In such cases, we can use a larger warehouse to reduce spilling because it can accommodate more data in memory. As previously mentioned, you should experiment with differently sized warehouses to determine the best size where the increased cost justifies the performance gain.

TIP Generally, you shouldn't increase the warehouse size to accommodate the requirements of a single query or a single user. Always consider the complete workload.

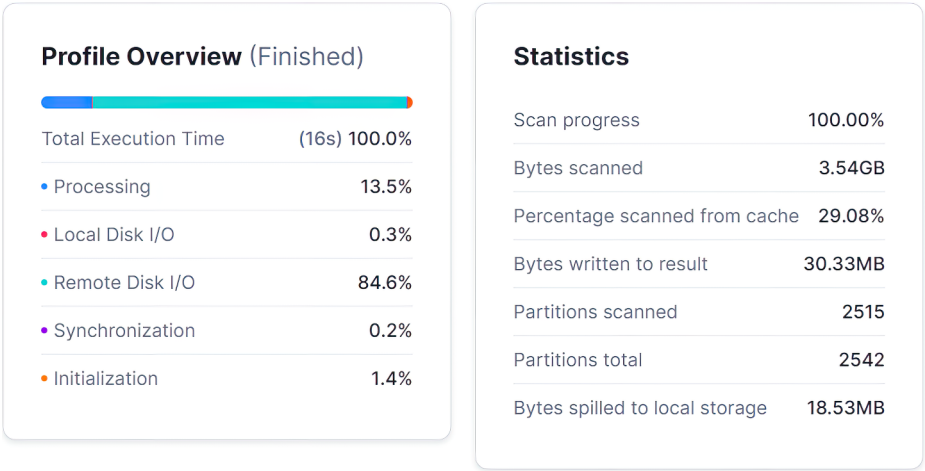


Figure 9.6 The Profile Overview and the Statistics panes from the query profile after executing the query that filters data to include only stores less than 1,000 kilometers away from the bakery's location using the `BAKERY_WH_XSMALL` virtual warehouse

Once you complete the exercises in this section, remember to set the `USE_CACHED_RESULT` parameter back to its original value of `TRUE` in the current session:

```
alter session set use_cached_result = TRUE;
```

9.3 Optimizing performance with data caching

Data caching is a process that stores copies of data in a temporary storage location with fast access, also referred to as a cache, so that the data can be retrieved faster than from its original storage location. Caching allows users to reuse previously retrieved or computed data, which helps to improve query performance.

Snowflake maintains different types of data caches within the three layers of its architecture. As described in chapter 1, the three layers of the Snowflake architecture are cloud services, query processing, and database storage.

The cloud services layer contains the query results cache and the metadata cache with the following properties:

- The *query results cache* stores the results of executed queries, usually for 24 hours. This is helpful when the same query is executed multiple times because Snowflake doesn't have to execute the query a second time; it just retrieves the results from the cache. We saw this behavior earlier in this chapter when we described how Snowflake uses persisted query results.
- The *metadata cache* stores metadata, including information about databases, schemas, tables, views, and other database objects. It also stores information about these objects, such as the number of rows in a table and other properties. A query whose result can be retrieved from the metadata cache executes quickly without accessing table data or using a virtual warehouse.

The query processing layer contains a cache for each of the virtual warehouses. When queries are executed using a virtual warehouse, the data from the underlying tables is stored in the warehouse cache. Subsequent queries can read data from the cache in the query processing layer instead of accessing the table data in the database storage layer. The size of the cache is determined by the warehouse size, meaning larger warehouses have larger cache areas.

Figure 9.7 illustrates the three layers of the Snowflake architecture and the associated caches: the metadata cache, the query results cache, and the warehouse cache for each virtual warehouse.

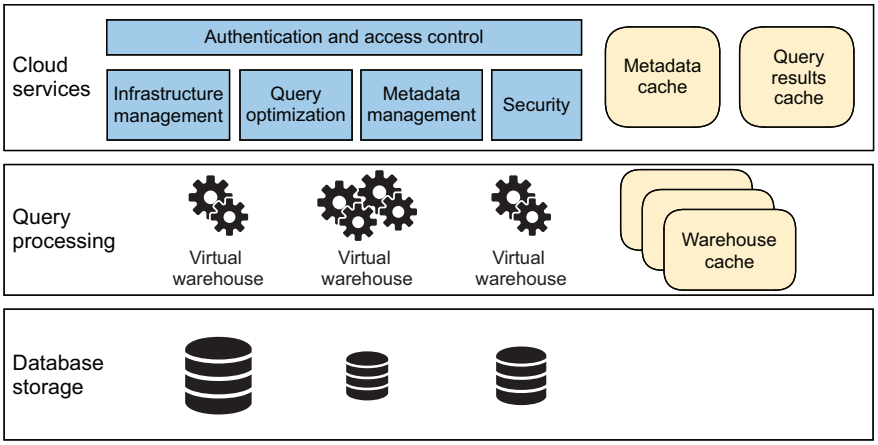


Figure 9.7 The three layers of the Snowflake architecture are database storage, query processing, and cloud services. The cloud services layer holds the metadata and the query results cache. The query processing layer holds the warehouse cache for each virtual warehouse.

9.3.1 *Illustrating the metadata cache*

Let's illustrate how the metadata cache works with an example. Data engineers often check the number of records in a table after ingesting data. To see how many records were inserted into the `RETAILER_SALES` table, they might execute a simple query like the following:

```
select count(*) from RETAILER_SALES;
```

After executing this query, let's open the query profile. In the graphical representation of the query, we should see a box indicating that the query was a metadata-based result, as shown in figure 9.8.

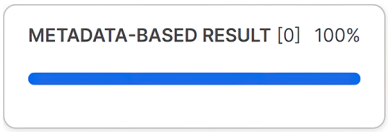


Figure 9.8 Query profile after executing a query that uses the metadata cache

When the metadata cache is used to retrieve a query's results, no virtual warehouse is used, and no credits are consumed.

9.3.2 Utilizing the warehouse cache efficiently

The virtual warehouse cache is available when the warehouse is running and is dropped when the warehouse is suspended. This explains why some queries execute slower when the warehouse is initially started. After the warehouse is active, query performance improves when queries are executed over time, and the data from the underlying tables is copied into the cache.

This is an important concept to understand when deciding how to configure the virtual warehouse `AUTO_SUSPEND` parameter, which specifies how long the warehouse can be inactive before Snowflake suspends it. To prevent the cache from disappearing once a warehouse is suspended, you might consider keeping the warehouse running, but this consumes credits.

> **NOTE** Remember that an active warehouse incurs costs even when it is not processing any queries.

Snowflake recommends the following general guidelines when specifying the `AUTO_SUSPEND` parameter for a warehouse:

- A warehouse dedicated only to tasks can be suspended immediately because another user is unlikely to execute queries on the same warehouse.
- A warehouse dedicated to ad hoc querying can be suspended after about 5 minutes because the cache is not very useful with non-repetitive queries.
- A warehouse dedicated to analytical queries performed by business analysts and reporting tools can be suspended after 10 minutes or more, allowing users to access the cached data.

We can set the `AUTO_SUSPEND` parameter when we initially create the warehouse or alter it later. For example, let's alter the `BAKERY_WH_XSMALL` warehouse to suspend after 5 minutes of inactivity. The `AUTO_SUSPEND` parameter expects the value in seconds, so we will set the value to 300 seconds by executing the following command:

```
alter warehouse BAKERY_WH_XSMALL set AUTO_SUSPEND = 300;
```

9.4 Reducing query queuing

When many users or applications use the same virtual warehouse to execute queries, the warehouse may not have sufficient resources to process all the queries simultaneously. In such cases, the warehouse queues the queries, waiting until resources become available. Query performance degrades when this occurs because the query execution time increases by the waiting time. Reducing queuing can improve performance because the query has less waiting time before execution.

9.4.1 Examining queuing

To learn whether a virtual warehouse is experiencing queuing or has experienced queuing in the past, we can use the Snowsight user interface by clicking the Warehouse option in the Admin menu and then exploring the chosen warehouse. We can also write a query that retrieves queuing data from the `WAREHOUSE_LOAD_HISTORY` table in the `SNOWFLAKE` database. This table contains data about the workload in each warehouse within 5-minute intervals. We will examine the contents of the following columns in this table:

- `AVG_RUNNING`—The ratio of the total running time of all queries during the interval and the total time of the interval
- `AVG_QUEUED_LOAD`—The ratio of the total queuing time of all queries during the interval and the total time of the interval

Because the `SYSADMIN` role that we are using doesn't have access to this table in the `SNOWFLAKE` database, we must use the `ACCOUNTADMIN` role to grant the `USAGE_VIEWER` database role to the `SYSADMIN` role using the following commands:

```
use role ACCOUNTADMIN;
grant database role SNOWFLAKE.USAGE_VIEWER to role SYSADMIN;
use role SYSADMIN;
```

Then, we can use the `SYSADMIN` role again to query the `WAREHOUSE_LOAD_HISTORY` table. To see how much time each warehouse spent queuing by day for the past seven days, we can summarize the `AVG_QUEUED_LOAD` column and group by the warehouse and the day. We will derive the day from the `START_TIME` column. We will also summarize the `AVG_RUNNING` column for comparison. The query is

```
select
  to_date(start_time) as start_date,
  warehouse_name,
  sum(avg_running) as total_running,
  sum(avg_queued_load) as total_queued
from SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_LOAD_HISTORY
where TO_DATE(start_time) > DATEADD(day,-7,TO_DATE(CURRENT_TIMESTAMP()))
group by all
order by 1, 2;
```

The output of this query is shown in table 9.4 (not all data is shown). If you are following along using your Snowflake account, your results will reflect the usage in your account and will differ from these.

Table 9.4 Output of the query that summarizes the queuing time and the total execution time by each warehouse by day for the past seven days

Start date	Warehouse name	Total running time	Total queuing time
2023-08-04	BAKERY_WH_LARGE	0.038386667	0.000000000

Table 9.4 Output of the query that summarizes the queuing time and the total execution time by each warehouse by day for the past seven days *(continued)*

Start date	Warehouse name	Total running time	Total queuing time
2023-08-04	BAKERY_WH_MEDIUM	0.000986667	0.000000000
2023-08-04	BAKERY_WH_SMALL	0.001086667	0.000000000
2023-08-04	BAKERY_WH_XSMALL	0.181326668	0.000000000

As shown in table 9.4, the total queuing time value is always 0. This is expected in the Snowflake trial account used for the exercises in this chapter because multiple users don't query the warehouses simultaneously, which would cause queuing. However, you might see different results when you examine queuing time in an actual Snowflake account where multiple users execute queries simultaneously.

To reduce query queuing, Snowflake recommends the following actions:

- When the virtual warehouse is a single-cluster warehouse (not multicluster), create additional warehouses and assign different warehouses to different groups of users or applications so that fewer queries are executed against each warehouse.
- Instead of creating additional warehouses, convert the single-cluster warehouse to a multicluster warehouse. In this case, Snowflake provisions additional clusters as needed by the workload and suspends them when no longer needed.
- Increase the maximum number of clusters when the virtual warehouse is already a multicluster warehouse.

TIP Only consider optimizing warehouses to reduce query queuing when users ask for it or applications require better performance. If queuing occurs during nightly batch loads and performance is not crucial, you can accept queuing because it indicates that the warehouse is fully utilized, and you are not spending credits for when the warehouse is active and idle.

9.4.2 Limiting concurrently running queries

Another option to improve query performance when many users use the same virtual warehouse to execute queries is to limit the number of queries running concurrently. With fewer queries running simultaneously, each gets more of the warehouse's resources. Large and complex queries benefit the most from having more resources for execution.

We can limit the number of queries running concurrently on a virtual warehouse by changing the value of the `MAX_CONCURRENCY_LEVEL` parameter from the default value, which is eight, to a lower value. For example, to limit the number of concurrently running queries on the `BAKERY_WH_LARGE` warehouse to at most six, we can execute the following command:

```
alter warehouse BAKERY_WH_LARGE set MAX_CONCURRENCY_LEVEL = 6;
```

TIP By limiting the number of queries running concurrently, you can inadvertently cause more queries to be queued because the warehouse will process fewer queries simultaneously. Be sure to understand query queuing as described in the previous section, and monitor warehouse execution after you make any changes to the warehouse parameters.

9.5 Monitoring compute consumption

Snowflake provides many options for viewing, monitoring, and controlling compute costs. The Snowsight user interface displays visual dashboards that show costs according to various parameters. Data engineers can also write queries that retrieve data from the views in the `ACCOUNT_USAGE` and `ORGANIZATION_USAGE` schemas in the `SNOWFLAKE` database. You can use these queries to create custom reports related to consumption or send alerts when the cost deviates from expected thresholds.

NOTE More information about monitoring compute costs can be found in the Snowflake documentation at https://mng.bz/0Mvl.

To view compute cost in the Snowsight user interface, click the Cost Management option under the Admin menu, choose a warehouse, and look at the Consumption tab. You will see a dashboard that resembles what is shown in figure 9.9. Your dashboard will be different, representing your consumption in your Snowflake account.

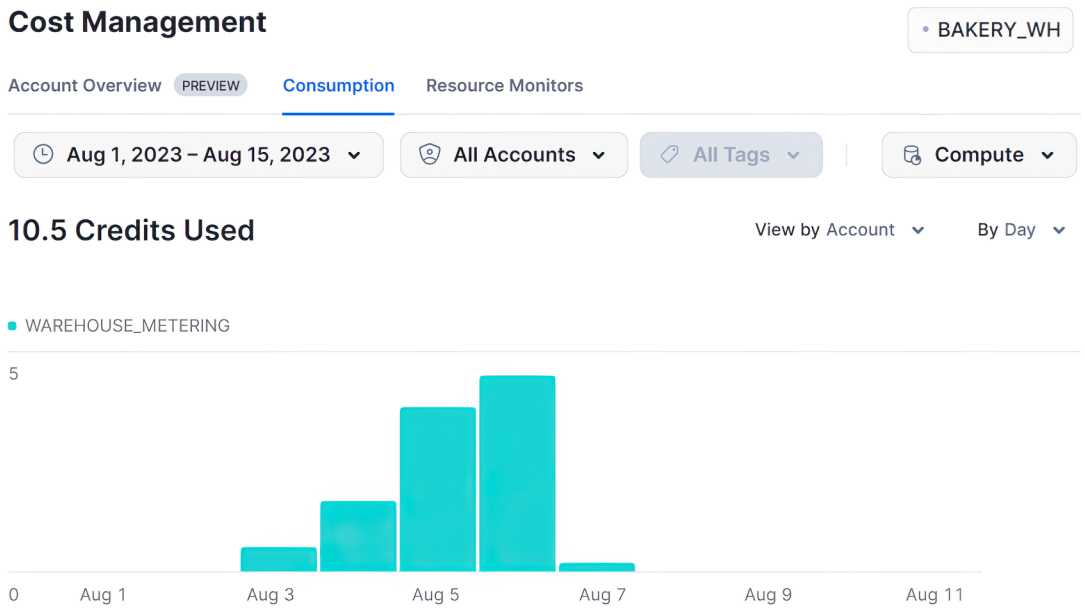


Figure 9.9 A sample cost management dashboard in the Snowsight user interface shows a warehouse's daily compute credit consumption at a chosen time interval.

Resource monitors

If you don't want to continuously monitor the cost or worry about overspending on your Snowflake account, set up one or more *resource monitors*. These Snowflake objects allow Snowflake administrators to limit credits consumed by a warehouse or the entire Snowflake account during a specific time interval. This can help prevent warehouses from consuming more credits than expected.

You can configure the resource monitor to send an alert, suspend the warehouse after the currently running queries are complete, or suspend the warehouse immediately when the consumption approaches the defined limit.

More information about resource monitors is available in the Snowflake documentation at https://mng.bz/KDVZ.

This chapter described some of the most important aspects of cost controlling in Snowflake. While data engineers are responsible for building cost-efficient data pipelines, they often work in teams with Snowflake administrators and system architects who share responsibilities related to controlling cost, such as defining and sizing virtual warehouses, monitoring consumption, or setting up resource monitors.

Summary

- Snowflake uses a consumption-based pricing model, where users pay for storage and compute resources. The total cost is calculated as the sum of the compute resources cost, the storage resources cost, and the data transfer resources cost.
- The total cost for using Snowflake's compute resources is calculated as the sum of the virtual warehouse compute cost, the serverless compute cost, and the cloud services compute cost.
- A Snowflake virtual warehouse is characterized by its size and additional properties that define and automate warehouse activity. The warehouse size determines the compute resources for executing queries and other processing operations.
- Snowflake persists the results of queries for a certain period after they are executed. This helps to optimize performance when the same query is executed many times because it retrieves the persisted results instead of re-executing the query.
- Increasing the warehouse size to the next-larger value consumes twice the number of credits. It provides roughly twice as many resources, which results in faster query execution time, particularly for large and complex queries.
- Spilling happens when a warehouse runs out of memory while executing a query, causing it to write data to local disk storage.
- To improve query performance due to spilling, try to rewrite the query to reduce the amount of data it must process. If this is not possible, increase the warehouse size.

- Data caching is a process that stores copies of data in a temporary storage location with fast access, also referred to as a cache, so that the data can be retrieved faster than from its original storage location.
- Snowflake maintains different types of data caches within its three layers of architecture. The cloud services layer contains the query results cache and the metadata cache, and the query processing layer contains a cache for each of the virtual warehouses.
- When many users or applications use the same virtual warehouse to execute queries, the warehouse may not have sufficient resources to process all the queries simultaneously. In such cases, the warehouse queues the queries, waiting until resources become available.
- To learn whether a virtual warehouse is experiencing queuing or has experienced queuing in the past, query the `WAREHOUSE_LOAD_HISTORY` table in the `SNOWFLAKE` database.
- To reduce query queuing, Snowflake recommends creating additional warehouses, converting a single-cluster warehouse to a multicluster warehouse, or increasing the maximum number of clusters in a multicluster warehouse.
- Snowflake provides many options for viewing, monitoring, and controlling compute costs. The Snowsight user interface displays visual dashboards that show costs according to various parameters. This information is stored in multiple views in the `ACCOUNT_USAGE` and `ORGANIZATION_USAGE` schemas in the `SNOWFLAKE` database, which can be queried.

10 Data governance and access control

This chapter covers

- Snowflake role-based access control
- Securing data with row access policies
- Protecting sensitive data with masking policies

Data stored in various repositories such as source systems, data lakes, data warehouses, reporting solutions, or data products can contain confidential or sensitive information only authorized users can access. Data engineers are responsible for including data governance and access control requirements when building data pipelines to ensure users see only the data they are authorized to see. Snowflake supports role-based access control and data governance features, such as row access policies and masking policies, that limit data access to authorized users.

In this chapter, we will describe the Snowflake *role-based access control* (RBAC) model, where access privileges are assigned to roles that are granted to users. We will demonstrate row access policies, typically used in multitenant solutions where many business units store data in the same database but can only see data from their business unit. We will also review masking policies that mask sensitive data, such as personal or financial information.

To illustrate the examples in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries and delivers these baked goods to small businesses, such as grocery stores, coffee shops, restaurants, and hotels in the neighborhood. The bakery wants to securely store the data it collects so only authorized users can access it.

The bakery will introduce roles, such as the data engineer role, which can read and write data, and the data analyst role, which can only read data. As the bakery is expanding, it wants to manage the bread and pastry business units separately, giving each business unit manager access to only the data related to their business unit. Additionally, the bakery wants to ensure that any personal data stored in tables appears masked when users query it.

NOTE All code and sample data files for this chapter are available in the accompanying GitHub repository in the Chapter_10 folder at https://mng.bz/9opo.

The SQL code is stored in multiple files whose names start with Chapter_10_Part*, where * indicates a sequence number and additional information refers to the file's content. Please follow the exercises by reading the files sequentially.

10.1 Role-based access control

Snowflake architects and administrators usually design and implement access control structures that allow or restrict access to database objects for specific users or roles. They can define access control in Snowflake in two ways:

- As a *discretionary access control* (DAC) model, where each object has an owner who grants or revokes access to the object to other users or roles
- As a *role-based access control* (RBAC) model, where each object is owned by a role, and access is granted to users by assigning the respective role

NOTE For more information about access control, refer to the Snowflake documentation at https://mng.bz/j0Ep.

The RBAC model is often preferred because it ensures centralized grant management rather than allowing individual object owners to grant or revoke privileges to their liking. In the RBAC model, privileges on securable objects are granted to and revoked from roles. Roles are then assigned to users to control what actions they can perform on the securable objects.

A hierarchy of roles can be created by granting roles to other roles. A role at a higher level of the hierarchy inherits all privileges granted to roles at a lower level. Snowflake provides a set of system-defined roles and functionality for defining a hierarchy of custom roles.

10.1.1 System-defined roles

Every Snowflake account includes system-defined roles with privileges required for account management. These system-defined roles cannot be dropped, and their privileges cannot be revoked.

We have been using the SYSADMIN role for the exercises in the previous chapters because we didn't have any custom roles created. Before defining custom roles, it's beneficial to understand the purpose of a few system-defined roles, including ACCOUNTADMIN, SYSADMIN, SECURITYADMIN, and USERADMIN. The purpose of these system-defined roles is as follows:

- ACCOUNTADMIN is the top-level role in each account and is used for account configuration, monitoring, and maintenance only.

WARNING Never create objects or issue grants using the ACCOUNTADMIN role. If you do, the ACCOUNTADMIN role will own the objects or grants, making it difficult to manage with a different role in the future.

- SECURITYADMIN is a powerful role with the MANAGE GRANTS privilege. It can create, monitor, and manage users and roles in the account. This role is usually not granted custom roles, so it doesn't have access to data.
- USERADMIN is the role usually used to create users and roles in an implementation environment and manage the users and roles it creates.
- SYSADMIN is the primary role that creates objects such as warehouses, databases, and other objects in the account. Custom roles are usually granted to this role, which allows it to access all data and assume all roles.

10.1.2 Custom roles

Snowflake recommends creating a hierarchy of custom roles, with the topmost custom role granted to the SYSADMIN role. This role structure allows system administrators to manage all objects in the account because each role in the hierarchy inherits the privileges of the roles in the lower levels.

NOTE For more information about Snowflake's role hierarchy and privilege inheritance, refer to the Snowflake documentation at https://mng.bz/WV4w.

The USERADMIN or any other role with the CREATE ROLE privilege can create custom roles. Once a role is defined, it must be granted to a user or another role to take effect.

When creating an object in Snowflake, the object ownership can be implemented in two different ways according to the schema type:

- When creating an object in a *regular schema*, the role that creates the object also owns it. This role has all privileges on the object, including the ability to grant or revoke privileges on the object to other users or roles and to transfer ownership.
- When creating an object in a *managed access schema*, the schema itself owns it. There are no individual object owners. Privileges on objects in a managed

access schema can be granted by the schema owner or by a role with the `MANAGE GRANTS` privilege.

10.1.3 Designing RBAC

Let's set up a simple RBAC environment for the examples in this chapter. To design RBAC for an environment, consider the following:

- Define a set of *functional roles* for users according to their use of the environment. For example, functional roles could be data engineer, data analyst, or data scientist. Some of these roles, such as data analyst, are too generic. We will further refine functional roles in the following sections according to the business functions within the organization, such as the bakery business unit data analyst, the pastry business unit data analyst, or the human resources data analyst. Functional roles often align to users based on their job functions.
- Define a set of *access roles* with a common set of privileges, such as a role with full access to objects in each schema, another role with read-only access, and additional roles as needed—for example, roles with read and write privileges.
- Remember that access roles are granted to functional roles according to the required privileges of each functional role, and functional roles are granted to users.

TIP Do not grant access roles to other access roles, and grant functional roles to other functional roles sparingly to ensure that the role hierarchy is not too complex to maintain.

- Use managed access schemas to prevent object owners from granting access to individual objects to other roles at their discretion. Define future grants on objects in the schemas so that newly created objects receive grants to access roles automatically.

To illustrate the RBAC concept for the bakery example, we will create the following objects:

- A schema named `RAW` that stores raw data
- A schema named `RPT` that stores data transformed for reporting
- An access role named `BAKERY_FULL` that has full privileges on all data in all schemas
- An access role named `BAKERY_READ` that can read all data in the `RPT` schema
- A data engineer functional role named `DATA_ENGINEER` that has full privileges to all objects in all schemas
- A data analyst functional role named `DATA_ANALYST` that can read data only in the `RPT` schema

The roles and objects used in the described example are shown in figure 10.1.

Let's continue by creating the environment. If you have followed the exercises in the previous chapters, you already have the `BAKERY_DB` database created using the

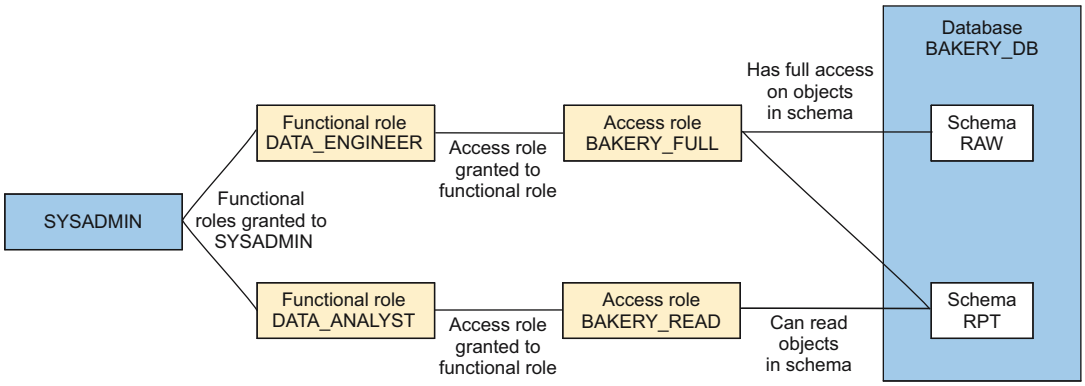


Figure 10.1 Role-based access control configuration of the environment used in the example in this chapter. The environment includes two functional roles, `DATA_ENGINEER` and `DATA ANALYST`, and two access roles, `BAKERY_FULL` and `BAKERY_READ`. Access roles are granted to functional roles, and functional roles are granted to the `SYSADMIN` role.

`SYSADMIN` role. You also have the `BAKERY_WH` virtual warehouse created. If not, you can create them now. We will create these objects if they don't already exist by executing the following commands using the `SYSADMIN` role:

```
use role SYSADMIN;
create warehouse if not exists BAKERY_WH with warehouse_size = 'XSMALL';
create database if not exists BAKERY_DB;
use database BAKERY_DB;
```

Then we will create the schemas using the managed access option:

```
create schema RAW with managed access;
create schema RPT with managed access;
```

We will switch to the `USERADMIN` role—because this role has the `CREATE ROLE` privilege—to create the access roles and the functional roles:

```
use role USERADMIN;
create role BAKERY_FULL;
create role BAKERY_READ;

create role DATA_ENGINEER;
create role DATA_ANALYST;
```

We will switch to the `SECURITYADMIN` role—because this role has the `MANAGE GRANTS` privilege—to grant privileges to each access role. To grant full access to the `BAKERY_FULL` role, we must grant usage on the `BAKERY_DB` database, usage on all schemas in the `BAKERY_DB` database, and all privileges in each of the schemas:

```
use role SECURITYADMIN;
grant usage on database BAKERY_DB to role BAKERY_FULL;
grant usage on all schemas in database BAKERY_DB to role BAKERY_FULL;
grant all on schema BAKERY_DB.RAW to role BAKERY_FULL;
grant all on schema BAKERY_DB.RPT to role BAKERY_FULL;
```

To grant read access in the `RPT` schema to the `BAKERY_READ` role, we must grant usage on the `BAKERY_DB` database, usage on the `RPT` schema in the `BAKERY_DB` database, and the select privilege on all tables and views in the `RPT` schema:

```
grant usage on database BAKERY_DB to role BAKERY_READ;
grant usage on schema BAKERY_DB.RPT to role BAKERY_READ;
grant select on all tables in schema BAKERY_DB.RPT to role BAKERY_READ;
grant select on all views in schema BAKERY_DB.RPT to role BAKERY_READ;
```

With these grants, the `BAKERY_READ` role receives read access to all existing tables and views in the `RPT` schema. Since we just created the `RPT` schema, it contains no tables or views. We want to ensure that the `BAKERY_READ` role has access to new tables and views created in this schema in the future. Therefore, we must also grant future access using the following commands:

```
grant select on future tables in schema BAKERY_DB.RPT to role BAKERY_READ;
grant select on future views in schema BAKERY_DB.RPT to role BAKERY_READ;
```

> **NOTE** In this example, we are granting read access only to tables and views in the schema. If we store other types of objects in the schema, like sequences, user-defined functions, and so on, we must issue additional grants for each object type.

Still using the `SECURITYADMIN` role, we will grant access roles to functional roles. The `BAKERY_FULL` access role is granted to the `DATA_ENGINEER` functional role because this role requires full access to all objects in all schemas:

```
grant role BAKERY_FULL to role DATA_ENGINEER;
```

The `BAKERY_READ` access role is granted to the `DATA_ANALYST` functional role because this role requires read access in the `RPT` schema for reporting:

```
grant role BAKERY_READ to role DATA_ANALYST;
```

As recommended by the RBAC concept, we will grant both functional roles to the `SYSADMIN` role so that it inherits all privileges from the functional roles and can access all objects:

```
grant role DATA_ENGINEER to role SYSADMIN;
grant role DATA_ANALYST to role SYSADMIN;
```

Once the functional roles are created, we can grant them to the users who perform the respective business functions. Because this is an exercise, we don't have any additional

users in our Snowflake account. But we still want to test using the functional roles. To do that, we will grant the functional roles to our current user. We can retrieve the name of the current user into a session variable named `my_current_user` using the `CURRENT_USER()` function:

```
set my_current_user = CURRENT_USER();
```

Then, we can grant the functional roles to the user represented in the `my_current_user` session variable. Because we are using a variable instead of a name, we must use the `IDENTIFIER()` function and prefix the variable name with a dollar sign:

```
grant role DATA_ENGINEER to user IDENTIFIER($my_current_user);
grant role DATA_ANALYST to user IDENTIFIER($my_current_user);
```

The functional role will also need privileges to use a virtual warehouse. Therefore, we will grant usage on the `BAKERY_WH` warehouse to the functional roles:

```
grant usage on warehouse BAKERY_WH to role DATA_ENGINEER;
grant usage on warehouse BAKERY_WH to role DATA_ANALYST;
```

> **TIP** In the exercises in this chapter, we are using just one warehouse named `BAKERY_WH`. In real scenarios, additional warehouses are created and granted to different functional roles for better performance and cost management.

Once the environment is created, let's do a test to verify that the access control is working as expected. First, we will use the `DATA_ENGINEER` role to create a table named `EMPLOYEE` in the `RAW` schema and insert a few sample values into this table. Since the `DATA_ENGINEER` role has full access, it should be allowed to create the table and insert data into it using the following commands:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema RAW;

create table EMPLOYEE (
  id integer,
  name varchar,
  home_address varchar,
  department varchar,
  hire_date date
);

insert into EMPLOYEE values
(1001, 'William Jones', '5170 Arcu St.', 'Bread', '2020-02-01'),
(1002, 'Alexander North', '261 Ipsum Rd.', 'Pastry', '2021-04-01'),
(1003, 'Jennifer Navarro', '880 Dictum Ave.', 'Pastry', '2019-08-01'),
(1004, 'Sandra Perkins', '55 Velo St.', 'Bread', '2022-05-01');
```

Then we will switch to the DATA_ANALYST role and select from the EMPLOYEE table. Since the DATA_ANALYST role is not authorized to see data in the RAW schema, the following query should not succeed:

```
use role DATA_ANALYST;
select * from RAW.EMPLOYEE;
```

The output of this query is an error message saying "Object 'BAKERY_DB.RAW.EMPLOYEE' does not exist or not authorized." The DATA_ANALYST role can view data only in the RPT schema. We can use the DATA_ENGINEER role again to create a view named EMPLOYEE in the RPT schema that selects the data from the EMPLOYEE table in the RAW schema as follows:

```
use role DATA_ENGINEER;
create view RPT.EMPLOYEE as
select id, name, home_address, department, hire_date
from RAW.EMPLOYEE;
```

Then we can switch to the DATA_ANALYST role and select data from the EMPLOYEE view in the RPT schema:

```
use role DATA_ANALYST;
select * from RPT.EMPLOYEE;
```

This command succeeds because the data analyst can read data from tables and views in the RPT schema. The output of the command is shown in table 10.1.

Table 10.1 Output when selecting from the EMPLOYEE view in the RPT schema

ID	Name	Home address	Department	Hire date
1001	William Jones	5170 Arcu St.	Bread	2020-02-01
1002	Alexander North	261 Ipsum Rd.	Pastry	2021-04-01
1003	Jennifer Navarro	880 Dictum Ave.	Pastry	2019-08-01
1004	Sandra Perkins	55 Velo St.	Bread	2022-05-01

In a real-world scenario, the role-based access control configuration is usually much more detailed than the simple example used in this exercise. The following are some additional considerations when designing role-based access control in larger environments:

- In our example, we used the SYSADMIN, USERADMIN, and SECURITYADMIN system-defined roles to create objects and roles and issue grant commands. Usually, we don't use system-defined roles to build environments. Instead, we create additional environment administration roles that perform these actions within each environment.

- As in our example, access roles are usually defined for each schema rather than the entire database. This ensures greater flexibility in granting access to functional roles on a schema level. Stored procedures or reusable scripts are usually created to automate the creation of access roles.
- Instead of using a single virtual warehouse like in our example, multiple virtual warehouses are created and granted to individual roles to meet their usage requirements better and monitor compute costs.

In our example, the data engineer created the view required by the data analyst. In some organizations, the business users create the reporting layer. In this case, they could not use the `DATA_ANALYST` role because according to our role-based access design, this role only has read access.

If the data analysts create their own reporting layer, we could grant the `DATA_ANALYST` role additional privileges, allowing it to create and maintain objects in designated schemas. Alternatively, we could create additional roles for the advanced data analysts who create the reporting layer. At the same time, the `DATA_ANALYST` role could still only have read privileges for users requiring read access.

10.2 Securing data with row access policies

Databases in real-world applications sometimes store data in a *multitenant* architecture that allows multiple user groups—also called tenants—to store data in the same database, sharing the infrastructure and running only one instance of the application. However, the data belonging to a particular user group should be accessed only by those in that group. Some examples of user groups that might share an application are business units within an organization, geographical regions, or production plants.

In the previous section, we created a `DATA_ANALYST` role that can view all data in tables and views in the `RPT` schema. We also created a sample `EMPLOYEE` table that stores information about employees in the bakery's bread and pastry business units. We will now create two additional functional roles, named `DATA_ANALYST_BREAD` and `DATA_ANALYST_PASTRY`. Each will have access to view data in the `RPT` schema but will see only the data from their respective business units. To accomplish that, we will apply Snowflake *row access policies*.

Snowflake row access policies determine which rows to return in the query result based on the role that executes the query and the conditions provided in the body of the row access policy. A row access policy is an object stored in a schema; therefore, we will create a schema named `DG`, where we will store the policy.

Row access policies can be managed in a centralized, decentralized, or hybrid governance approach as follows:

- In a *centralized* governance approach, the data governance officer in an organization defines the policies and applies them to objects.
- In a *decentralized* governance approach, the individual teams define the policies and apply them to objects.

- In a *hybrid* governance approach, the data governance officer in an organization defines the policies, and the individual teams apply them to objects.

To keep the exercises in this chapter simple, we will use the decentralized approach, in which the DATA_ENGINEER role defines and applies policies.

NOTE For more information about Snowflake row access policies, refer to the Snowflake documentation at https://mng.bz/86eB.

Figure 10.2 illustrates how the new DATA_ANALYST_BREAD and DATA_ANALYST_PASTRY functional roles fit into the RBAC model we created earlier. Each functional role is granted the BAKERY_READ access role, allowing them to view data in the RPT schema. Additionally, a row access policy is created in the DG schema and applied to the EMPLOYEE view in the RPT schema. The row access policy includes rules that filter rows based on the user's functional role.

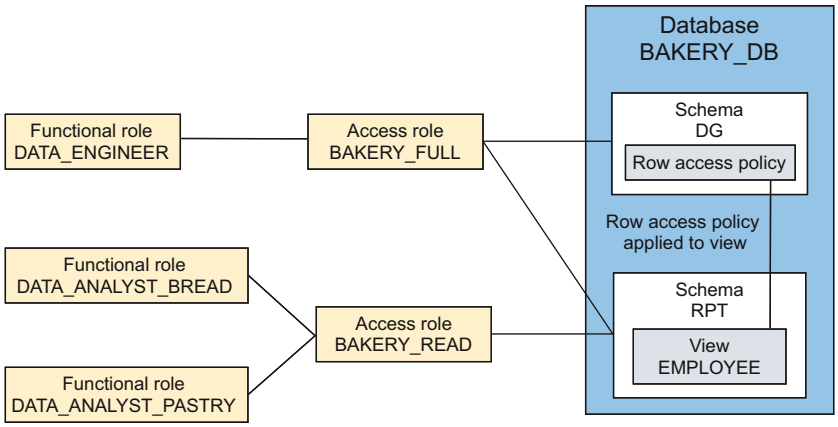


Figure 10.2 Functional roles DATA_ANALYST_BREAD and DATA_ANALYST_PASTRY are granted the BAKERY_READ access role, which allows them to read data in the RPT schema. A row access policy is applied to the EMPLOYEE view in the RPT schema, filtering rows based on the user's functional role.

To implement the row access policy, let's start by creating the DG schema with managed access using the SYSADMIN role and granting full access to the BAKERY_FULL access role:

```
use role SYSADMIN;
use database BAKERY_DB;

create schema DG with managed access;
grant all on schema DG to role BAKERY_FULL;
```

Then, we will use the USERADMIN role to create the new functional roles and grant the BAKERY_READ access role to the functional roles:

```
use role USERADMIN;
create role DATA_ANALYST_BREAD;
create role DATA_ANALYST_PASTRY;

grant role BAKERY_READ to role DATA_ANALYST_BREAD;
grant role BAKERY_READ to role DATA_ANALYST_PASTRY;
```

To be able to test these roles, we will grant them to our current user using a session variable as in the previous section:

```
set my_current_user = current_user();
grant role DATA_ANALYST_BREAD to user IDENTIFIER($my_current_user);
grant role DATA_ANALYST_PASTRY to user IDENTIFIER($my_current_user);
```

We must also grant virtual warehouse usage to the newly created functional roles so they will be able to select data:

```
use role SYSADMIN;
grant usage on warehouse BAKERY_WH to role DATA_ANALYST_BREAD;
grant usage on warehouse BAKERY_WH to role DATA_ANALYST_PASTRY;
```

Since the DATA_ENGINEER role will create and apply row access policies, we must grant the required privileges to it. It needs the privilege to create row access policies in the DG schema and the privilege to apply the policies on objects:

```
use role ACCOUNTADMIN;
grant create row access policy
  on schema BAKERY_DB.DG to role DATA_ENGINEER;
grant apply row access policy on account to role DATA_ENGINEER;
```

Now we can use the DATA_ENGINEER role to create a row access policy named RAP_BUSINESS_UNIT in the DG schema. This policy will include the following conditions:

- If the role that executes the query on the associated object is DATA_ENGINEER, the query returns data (it allows the DATA_ENGINEER role to view all data in the table).
- If the role that executes the query on the associated object is DATA_ANALYST_BREAD, and the value in the DEPARTMENT column is "Bread," the query returns data.
- If the role that executes the query on the associated object is DATA_ANALYST_PASTRY, and the value in the DEPARTMENT column is "Pastry," the query returns data.
- In all other cases, the query doesn't return the data.

The following is the code that creates the RAP_BUSINESS_UNIT row access policy in the DG schema:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema DG;
```

```
create row access policy DG.RAP_BUSINES_UNIT
as (DEPARTMENT varchar)
returns boolean ->
  case
-- return TRUE when the role is the creator of the row access policy
    when (is_role_in_session('DATA_ENGINEER'))
      then TRUE
-- grant access based on the mapping of role and department
    when (is_role_in_session('DATA_ANALYST_BREAD'))
      and DEPARTMENT = 'Bread'
      then TRUE
    when (is_role_in_session('DATA_ANALYST_PASTRY'))
      and DEPARTMENT = 'Pastry'
      then TRUE
-- otherwise return FALSE
    else FALSE
  end;
```

NOTE The row access policy in this example has the conditions defining which role can view which data coded in the body of the policy. Another way to implement the conditions in environments with many more roles and values is to create a mapping table between the roles and the filtering values and use it in the query in the body of the policy. For more information about using a mapping table, refer to the Snowflake documentation at https://mng.bz/EONj.

Once the row access policy is defined, it can be applied to the `EMPLOYEE` view in the `RPT` schema on the `DEPARTMENT` column:

```
alter view BAKERY_DB.RPT.EMPLOYEE
  add row access policy RAP_BUSINES_UNIT on (DEPARTMENT);
```

Now we can test to verify that the row access policy is working as expected. Let's start with the `DATA_ANALYST_BREAD` role and select data from the `EMPLOYEE` view in the `RPT` schema:

```
use role DATA_ANALYST_BREAD;
select * from BAKERY_DB.RPT.EMPLOYEE;
```

The output of this command should return only rows that belong to the Bread department, as shown in table 10.2.

Table 10.2 Output when selecting from the `EMPLOYEE` view in the `RPT` schema using the `DATA_ANALYST_BREAD` role

ID	Name	Home address	Department	Hire date
1001	William Jones	5170 Arcu St.	Bread	2020-02-01
1004	Sandra Perkins	55 Velo St.	Bread	2022-05-01

Then select the data from the same `EMPLOYEE` view in the `RPT` schema using the `DATA_ANALYST_PASTRY` role:

```
use role DATA_ANALYST_PASTRY;
select * from BAKERY_DB.RPT.EMPLOYEE;
```

The output of this command should return only rows that belong to the Pastry department, as shown in table 10.3.

Table 10.3 Output when selecting from the EMPLOYEE view in the RPT schema using the DATA_ANALYST_PASTRY role

ID	Name	Home address	Department	Hire date
1002	Alexander North	261 Ipsum Rd.	Pastry	2021-04-01
1003	Jennifer Navarro	880 Dictum Ave.	Pastry	2019-08-01

10.3 Protecting sensitive data with masking policies

Databases in real-world environments often store sensitive data, such as personal information like names, addresses, telephone numbers, social security numbers, and so on, or confidential information, like salary amounts. To avoid having to protect sensitive information, data warehouses sometimes don't store any such information. However, data analysts or data scientists might still need sensitive information. In such cases, the data warehousing solutions must ensure that such information is stored securely and only accessible to authorized users.

The Snowflake *masking policy* functionality protects sensitive data from unauthorized access while allowing authorized users to see it. Sensitive data is not stored as masked in the underlying tables. Instead, it is presented as masked, partially masked, or obfuscated at runtime when users execute a query. Masking policies are schema-level objects that define the conditions and masking functions to transform the data at query runtime to various roles that execute the query.

Like row access policies, masking policies can be implemented in a centralized, decentralized, or hybrid approach. In this exercise, the DATA_ENGINEER role will create and apply the masking policy using a decentralized approach.

The column HOME_ADDRESS in the EMPLOYEE table is considered personally identifiable information that must be masked when unauthorized users query it. To fulfill this requirement, we will create a masking policy that masks this column when queried by any role other than the DATA_ENGINEER role.

We start by granting the privileges to create and apply masking policies to the DATA_ENGINEER role:

```
use role ACCOUNTADMIN;
grant create masking policy on schema BAKERY_DB.DG to role DATA_ENGINEER;
grant apply masking policy on account to role DATA_ENGINEER;
```

Then we will use the DATA_ENGINEER role to create a masking policy named ADDRESS_MASK in the DG schema. The masking policy returns data when the current role is

DATA_ENGINEER and returns three asterisks otherwise. The code that creates this masking policy is

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema DG;

create masking policy ADDRESS_MASK
as (addr varchar)
returns varchar ->
  case
    when current_role() in ('DATA_ENGINEER') then addr
    else '***'
  end;
```

Once the masking policy is created, it can be applied to the HOME_ADDRESS column in the EMPLOYEE view in the RPT schema:

```
alter view BAKERY_DB.RPT.EMPLOYEE
modify column HOME_ADDRESS
set masking policy ADDRESS_MASK;
```

To test if the masking policy is working, we can use one of the functional roles we created in the previous section—for example, the DATA_ANALYST_BREAD role. This role should see masked data, as defined in the masking policy. Let's execute the query:

```
use role DATA_ANALYST_BREAD;
select * from BAKERY_DB.RPT.EMPLOYEE;
```

The output of this command should return only rows that belong to the Bread department with the HOME_ADDRESS column masked (displayed as three asterisks, as defined in the masking policy), as shown in table 10.4.

Table 10.4 Output when selecting from the EMPLOYEE view in the RPT schema using the DATA_ANALYST_BREAD role and after the masking policy on the HOME_ADDRESS column has been applied

ID	Name	Home address	Department	Hire date
1001	William Jones	***	Bread	2020-02-01
1004	Sandra Perkins	***	Bread	2022-05-01

We can select from the EMPLOYEE view in the RPT schema again using the DATA_ENGINEER role. This role should see unmasked data as defined in the masking policy. Let's execute the query:

```
use role DATA_ENGINEER;
select * from BAKERY_DB.RPT.EMPLOYEE;
```

The output from this query should be the complete unmasked data, as in table 10.1.

Masking policies can have more complex definitions than in our example, where we just replaced the value with asterisks. For example, when masking credit card information, we can define the masking policy to return only the last four digits. Masking policies can mask data at different levels depending on the role that executes the query. For example, they can show the full data to the authorized role, partially masked data to selected roles, and fully masked data to any other role.

In addition to row access policies and masking policies, Snowflake supports other data governance features, such as data classification, object tagging, tag-based masking policies, and more. For more information about the Snowflake data governance features, refer to the Snowflake documentation at https://mng.bz/NBZ7.

Summary

- Data engineers are responsible for including data governance and access control requirements when building data pipelines to ensure users see only the data they are authorized to see.
- Snowflake supports role-based access control and data governance features, such as row access policies and masking policies, that limit data access to authorized users.
- Snowflake architects and administrators design and implement access control structures that allow or restrict access to database objects for specific users or roles.
- In the RBAC model, privileges on securable objects are granted to and revoked from roles. Roles are then assigned to users to control what actions they are allowed to perform on the securable objects. A hierarchy of roles can be created by granting roles to other roles.
- Snowflake recommends creating a hierarchy of custom roles, with the topmost custom role granted to the `SYSADMIN` role. This role structure allows system administrators to manage all objects in the account because each role in the hierarchy inherits the privileges of the roles in the lower levels.
- To design RBAC for an environment, define a set of functional roles and a set of access roles with a common set of privileges, and then grant access roles to functional roles.
- Use managed access schemas to prevent object owners from granting access to individual objects to other roles at their discretion.
- Snowflake row access policies determine which rows to return in the query result based on the role that executes the query and the conditions provided in the body of the row access policy.

- Row access policies can be managed in a centralized, decentralized, or hybrid governance approach.
- The Snowflake masking policy functionality protects sensitive data from unauthorized access while allowing authorized users to see it. Sensitive data is not stored as masked in the underlying tables. Instead, it is presented as masked, partially masked, or obfuscated at runtime when users execute a query.

Part 3

Building data pipelines

This part of the book consolidates all your knowledge so far and demonstrates how to build a comprehensive data pipeline that executes on schedule. The chapters in this part progressively build on each other, resulting in a complete data pipeline.

Chapter 11 lays the groundwork by defining the data transformation layers, including extract, staging, data warehouse, and presentation.

Chapter 12 introduces incremental data ingestion, which is faster than full ingestion, as it involves moving and storing less data, resulting in reduced storage and compute costs.

Chapter 13 explains data pipeline orchestration as the process that involves scheduling, defining dependencies, error handling, and sending notifications to ensure efficient execution of data pipeline steps.

In chapter 14, we learn how to conduct data quality tests that validate data integrity and completeness and take remedial measures when test results don't meet the data quality standards.

Chapter 15 covers continuous integration, a software development practice in which data engineers frequently merge their code changes into the repository. After the merge, automated scripts execute the code, create database objects, perform integration tests, and carry out other necessary actions.

The examples used throughout the book to illustrate data engineering concepts and related Snowflake functionality should provide a solid foundation for continuing your journey in real-world data engineering.

11 Designing data pipelines

This chapter covers

- Designing data pipelines
- Comparing data pipeline patterns
- Choosing data transformation layers
- Defining role-based access control
- Building a sample data pipeline

A data pipeline is an automated sequence of steps designed to support the extraction, movement, ingestion, transformation, storage, and presentation of data from a source to a target platform. Data engineers must design data pipelines before writing code to implement them. To create a sound design, they must understand the purpose of the data pipeline, identify its data sources and targets, decide on a data pipeline pattern, choose the appropriate data transformation layers, and consider other user requirements, such as data governance and security.

In this chapter, we will design a data pipeline that ingests data from multiple sources. We will compare data pipeline patterns, including ETL (extract-transform-load), ELT (extract-load-transform), and ETLT (extract-transform-load-transform). We will choose the data transformation layers, such as extract, staging, data

warehouse, or presentation. We will set up role-based access control so that only authorized users can access data in the various layers of the pipeline.

To illustrate the examples in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries and delivers these baked goods to small businesses, such as grocery stores, coffee shops, restaurants, and hotels in the neighborhood. In the previous chapters, we developed different ways of ingesting data from various sources. We will now incorporate these data ingestion functionalities into a comprehensive data pipeline.

The data sources used in the previous chapters were file-based, such as CSV or JSON files. We also illustrated getting data from the Snowflake Marketplace in chapter 8. In real-world organizations, many data sources are relational databases used by operational systems that power the business. For the exercises in this chapter, we will simulate a few tables that might be sourced from the fictional bakery's information system, like details about business partners and the baked goods the bakery makes.

NOTE All code and sample data files for this chapter are available in the accompanying GitHub repository in the Chapter_11 folder at https://mng.bz/PN2g.

The SQL code is stored in multiple files whose names start with Chapter_11_Part*, where * indicates a sequence number and additional information means the file's content. Please follow the exercises by reading the files sequentially.

11.1 Designing data pipelines

Data pipelines should be designed to be robust, flexible, and scalable. They must accommodate increasing data volumes and additional data sources without breaking the design. Automation and maintainability are also vital aspects of effective data pipeline design.

The most common purpose of data pipelines is to present clean data to business users in a form they can readily consume. This can be accomplished with various applications, such as data warehousing, data marts, or data mesh, as described in chapter 1. In this chapter, we will design a data pipeline for a data warehousing application.

11.1.1 Extracting data

When designing data pipelines, we start by identifying the data sources, including their location and data format. Some examples of data sources we encountered in previous chapters are

- CSV files, uploaded to a Snowflake stage (chapters 2 and 6)
- CSV files, stored in external cloud storage (chapter 3)
- JSON files, stored in external cloud storage (chapters 4 and 5)
- Data available in the Snowflake Marketplace (chapter 8)

As described in the corresponding chapters, we took different approaches to ingesting data, depending on the format, the location, and the required ingestion frequency. We scheduled the pipeline if the data was required daily and used Snowpipe when the data was needed more frequently throughout the day.

Most data sources in organizations are operational systems that usually store data in databases. Because Snowflake is hosted in the cloud, it can't directly access data from such databases. A common approach to ingesting data from operational system databases into Snowflake is to use third-party data integration tools. Many data integration tools are available on the market, both open-source and commercial—for example, Fivetran, Informatica, Matillion, and Talend, to name a few.

An alternative to using third-party data integration tools to extract data from source system databases and copy it to Snowflake tables is to build a custom application using programming languages and development environments that the developers are familiar with. For example, data engineers can use Snowpark to create a custom data integration application.

Snowflake Ecosystem, Connectors, and Native App

Snowflake works with various tools, technologies, connectors, drivers, programming languages, and utilities. These solutions are grouped into categories, one of which is data integration.

For a list of data integration tools in the Snowflake Ecosystem, refer to the Snowflake documentation at https://mng.bz/JNrP. This is not an exhaustive list, as new tools are added or existing tools rebranded. Not all data integration tools are part of the Snowflake Ecosystem. Many additional tools, both commercial and open source, work with Snowflake as well.

Snowflake Connectors provide native integration of third-party applications and databases, such as Google Analytics or ServiceNow. Snowflake is expected to continue to support even more third-party applications in the future. Documentation about Snowflake Connectors is available at https://other-docs.snowflake.com/en/connectors.

The Snowflake Native App Framework can be used to build a custom data integration application and distribute it through a listing. More information about the Snowflake Native App Framework is available at https://mng.bz/w51W.

Each organization has its own criteria and requirements when deciding whether to use a third-party tool or custom-build a data integration application. The decision must be made before starting to build data pipelines.

To set up the examples in this chapter, we will assume that the bakery uses a data integration tool to extract data from its operational system and stores it in Snowflake daily. We will take a few tables from this fictional operational system, like the business partners and details about the baked goods they make.

11.1.2 Comparing data pipeline patterns

Once data has been ingested from a source database or file into Snowflake, it must be formatted, cleaned, and transformed to be valuable to downstream consumers. Various data pipeline design patterns, including ETL, ELT, or ETLT, can be used to implement this process, as described in the following sections.

ETL PATTERN

The ETL pattern originates from the earliest data warehousing applications and is still widely used today. With this pattern, data is extracted from the source systems, such as operational databases or files; formatted, cleaned, and transformed according to business requirements; and finally loaded into the target tables in Snowflake. The ETL pattern is illustrated in figure 11.1.

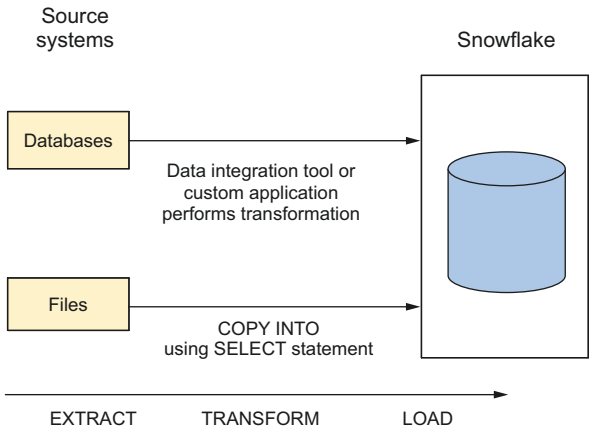


Figure 11.1 ETL pattern where data is first extracted from the source systems, then transformed according to business requirements, and finally loaded into Snowflake

As with any design pattern, the ETL pattern has advantages and disadvantages. Some of the advantages are

- It is suitable for traditional data warehousing applications where the data warehouse data model differs from the source system data model, requiring transformations.
- Complex data transformations are performed before the data is loaded into the data warehouse, eliminating the need to perform more complex transformations when preparing the data for downstream consumption.

The disadvantages of the ETL pattern are

- The complex transformations that are performed when extracting data from source systems can be time-consuming and require significant resources to process.
- The transformations are embedded in the tools or custom applications used for extracting data and may be cumbersome to maintain.
- Only the data resulting from the transformations is loaded into the data warehouse, while the source data before the transformation is unavailable.

ELT PATTERN

The ELT pattern is becoming popular as a replacement for the ETL pattern as data volumes increase. With this pattern, data is first extracted from source systems without any transformations and stored in a designated database in Snowflake, often referred to as an extraction layer or a landing zone. Then it is transformed inside Snowflake,

making it much more efficient than the ETL pattern because the full power of Snowflake compute resources can be applied to the transformation queries. The ELT pattern is illustrated in figure 11.2.

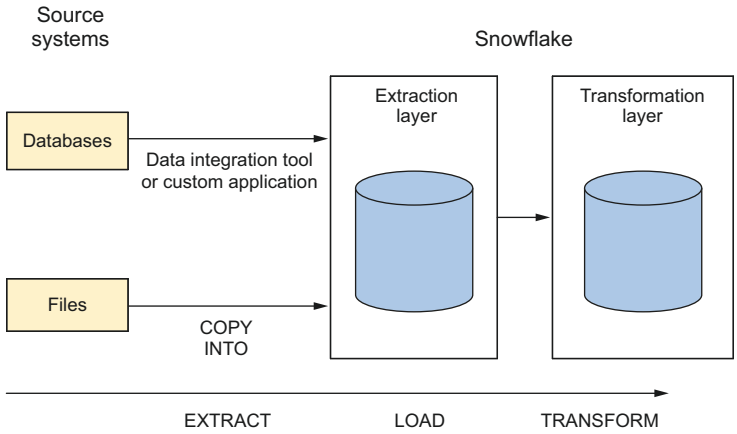


Figure 11.2 ELT pattern where data is first extracted without transformations, loaded into Snowflake, and then transformed using Snowflake compute resources

Some advantages of the ELT pattern are

- It is suitable for large data volumes that can be loaded into Snowflake efficiently without the added transformation queries that would consume time and resources.
- It can be used to ingest various data formats, both structured and semistructured, into Snowflake and transform them in a later step in the data pipeline.

Some disadvantages of the ELT pattern are

- Data is not formatted or cleaned before it is stored in Snowflake.
- More storage is consumed because the raw data is ingested, stored in Snowflake, and then transformed and stored again.

ETLT PATTERN

The ETLT pattern is a recently introduced pattern that combines the best of the ETL and ELT patterns. The first transformation in the ETLT pattern is used for simple transformations, such as data type conversions, that can be performed before loading the data without too much overhead and resource consumption. The second transformation is used for the more complex business transformations that benefit from using Snowflake compute resources.

> **TIP** When designing data pipelines, any data pipeline pattern may be used depending on the requirements and limited by the imposed constraints. For example, the ETL pattern is a good fit when the source system is a relational database with reasonably small amounts of data that require transformations.

The ELT pattern is a good fit when the source data contains large volumes of unstructured data. Sometimes, pre-existing architecture designs impose the pattern that must be used.

11.1.3 Choosing data transformation layers

As data flows through the pipeline, it produces intermediate outputs that can be stored physically in objects such as database tables, materialized views, or dynamic tables. Data can also be transformed virtually by saving the intermediate outputs as database views. It's a best practice to organize the outputs of the pipeline steps in layers.

The most common data transformation layers are the extraction layer, the staging layer, the data warehouse layer, and the presentation layer. Each layer follows the previous one, starting with ingesting data from source systems and resulting in clean transformed data suitable for downstream consumers. This layered approach to data warehousing transformations ensures better visibility and control over the data flows in the pipelines.

None of the layers are mandatory. For example, if you are building data marts in the presentation layer without a data warehouse, you can skip the data warehouse layer. You can skip the extraction layer if you use a data integration tool to extract data from the source system and perform transformations as in the ETL pattern. You can also add additional layers according to your needs.

The most common data transformation layers are described in more detail in the following sections.

EXTRACTION LAYER

The extraction layer is usually the starting point for the data pipeline. It can host the external or internal Snowflake stage, where the source data files are staged for ingestion. If used, it can host external tables. It can also store raw data from various sources in structured or semistructured format.

STAGING LAYER

The staging layer is often used to organize and consolidate data before it's stored in the data warehouse. Only light data transformations are implemented in this layer, like filtering, adding technical columns such as the ingestion timestamp, calculating surrogate keys if used, identifying changed data, etc. There is usually no business logic in the transformations in this layer.

DATA WAREHOUSE LAYER

The data warehouse layer is the common area where all the data for the entire business is collected from the various sources and modeled in a consistent data model that can take many forms, such as

- Custom-designed data model using a chosen data modeling approach—for example, normalized (Inmon approach), ensemble (anchor modeling or data vault), and others

- Industry-standard data model appropriate for the organization (banking, insurance, retail, manufacturing, etc.) and customized as needed

Ideally, the data stored in the data warehouse data model can be used for multiple use cases and serve multiple business units.

PRESENTATION LAYER

The presentation layer is where downstream reporting and analytics solutions access their data in a self-service way. The tables in this layer are structured to optimize query performance, often in a dimensional data model using fact and dimension tables (Kimball approach).

A graphical representation of the most common data transformation layers is shown in figure 11.3.

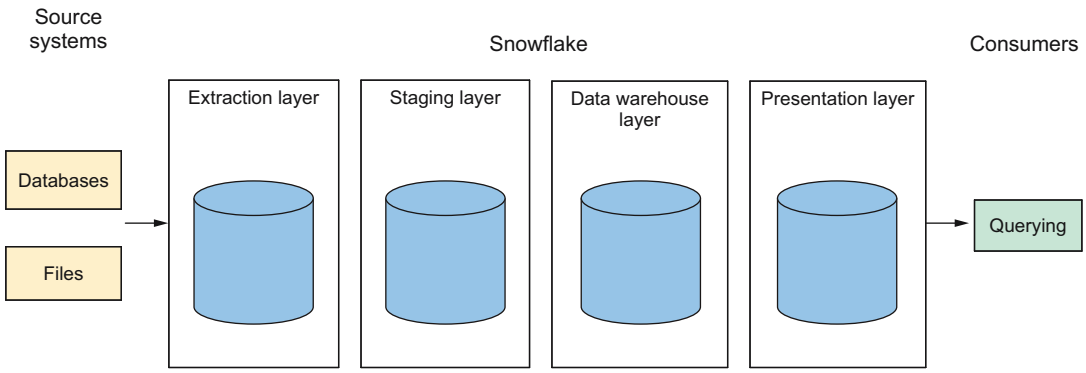


Figure 11.3 The most common data transformation layers in data pipelines are the extraction layer, the staging layer, the data warehouse layer, and the presentation layer.

11.1.4 Organizing data warehouse layers

The layers described in the previous section represent the logical layers in the data pipeline. Each layer contains data outputs that can be stored as objects, such as tables, views, materialized views, dynamic tables, or other types of objects in Snowflake.

The objects that represent outputs from the data pipeline layers can be organized hierarchically in containers in various ways—for example:

- The data pipeline can store all objects in a single database, and the layers are represented as schemas.
- Each of the layers of the data pipeline is a separate database.
- Some of the layers—for example, the extraction layer—are separate databases, while the remaining layers are schemas in another database.
- The presentation layer is a separate database with a schema for each business unit that is accessing the data, while the remaining layers are schemas in another database.

Since the data warehouse layers are logical units, they can be implemented as physical objects according to your unique requirements. When designing the physical structures for the layers, security is an essential consideration because we often want to limit consumers' access to only the objects for which they are authorized.

The following are some guidelines when deciding on the physical organization of the databases and schemas for the data transformation layers:

- When various tools or groups of developers access multiple data sources, you can organize the extraction layer as a separate database for each source system. This ensures the separation of the objects in each source system, especially when the source system contains many tables (examples are some popular enterprise resource planning systems) already organized in schemas. In this case, you can extract the data into a database organized by schemas according to the source system schemas.
- The staging layer can be implemented as additional schemas in the extraction layer database or as a schema in the data warehouse layer database. The warehouse layer usually combines data from all sources, and the staging layer is the intermediate step where the data comes together from the extraction layer.
- The presentation layer can be implemented as a schema in the database containing the data warehouse layer. However, if many business units have disparate reporting and analytics needs, you can organize the reporting layer in a separate database with individual schemas for each business unit. If even more separation is required—for example, for analytics sandboxes—each business unit can have its own database that serves its reporting needs.

For the examples in this chapter, we will use a single database named `BAKERY_DB` for all data transformation layers, with each layer represented as a schema. The extraction layer schema named `EXT` will host the staged data files. The `STG` staging layer schema will store the flattened semistructured data and data extracted from the bakery's operational IT system. The data warehouse layer schema named `DWH` will store combined data from the staging layer. The presentation layer schema named `MGMT` will store aggregated data for the bakery manager to use via a reporting tool or Snowflake dashboards.

The database and schemas used in the examples in this chapter are shown in figure 11.4.

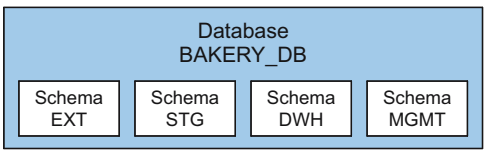


Figure 11.4 The data transformation layers used in the examples in this chapter are implemented as a single database named `BAKERY_DB`, with each layer represented as a schema named `EXT`, `STG`, `DWH`, and `MGMT`.

To start implementing the data pipeline, we will create the schemas with managed access and the corresponding access roles following the role-based access control (RBAC) approach as described in chapter 10.

> **NOTE** The exercises in this chapter assume that you completed the RBAC exercise in chapter 10. If you haven't done so already, please execute the `Chapter_10_Part1_role_based_access_control.sql` script in the Chapter_10 folder in the Git repository before continuing with the exercises in this chapter.

11.1.5 Creating schemas with access control

We will create the schemas in the `BAKERY_DB` database. The first step is to use the `SYSADMIN` role to create the new `EXT`, `STG`, `DWH`, and `MGMT` schemas with managed access (refer to chapter 10 for more information if needed):

```
use role SYSADMIN;
use database BAKERY_DB;
create schema EXT with managed access;
create schema STG with managed access;
create schema DWH with managed access;
create schema MGMT with managed access;
```

Then we will use the `SECURITYADMIN` role (because this role has the `MANAGE GRANTS` privilege) to grant full privileges on the `EXT`, `STG`, `DWH`, and `MGMT` schemas to the `BAKERY_FULL` access role:

```
use role SECURITYADMIN;
grant all on schema BAKERY_DB.EXT to role BAKERY_FULL;
grant all on schema BAKERY_DB.STG to role BAKERY_FULL;
grant all on schema BAKERY_DB.DWH to role BAKERY_FULL;
grant all on schema BAKERY_DB.MGMT to role BAKERY_FULL;
```

We will grant read-only privileges on current and future tables and views in the `MGMT` schema to the `BAKERY_READ` role:

```
grant select on all tables in schema BAKERY_DB.MGMT to role BAKERY_READ;
grant select on all views in schema BAKERY_DB.MGMT to role BAKERY_READ;

grant select on future tables in schema BAKERY_DB.MGMT to role BAKERY_READ;
grant select on future views in schema BAKERY_DB.MGMT to role BAKERY_READ;
```

We don't have to grant any additional privileges to the `DATA_ENGINEER` and `DATA_ANALYST` functional roles because these roles have already been granted the `BAKERY_FULL` and `BAKERY_READ` access roles, respectively, when we created them in chapter 10 and will inherit the new access privileges automatically.

11.2 Building a sample data pipeline

With the role-based access control configuration defined, we can now build a data pipeline for the bakery example. The pipeline ingests data from two sources:

- Order information from some of the bakery's customers, stored as JSON files in cloud storage. We already built this part of the pipeline in chapter 4, and we will reuse parts of the code.
- Details about the bakery's business partners and baked goods that reside in the bakery's operational IT system database. We assume that the data is extracted using a data integration tool and the tables created in the staging layer of the data pipeline.

We will build each data transformation layer in sequence, starting with the extraction layer, then the staging and data warehouse layers, and ending with the reporting layer. The purpose of the reporting layer is for the bakery manager to view summarized order information for the following days, grouped by the baked goods category.

We will use the `DATA_ENGINEER` functional role and the `BAKERY_WH` virtual warehouse to create all parts of the data pipeline.

11.2.1 Implementing the extraction layer

We will start with building the extraction layer. This layer contains the external stage that points to the cloud storage location where the JSON files are located. We will also create a table to store the data copied from the external files.

Using the `BAKERY_DB` database and the `EXT` schema representing the extraction layer, we will create an external stage named `JSON_ORDERS_STAGE`, just like in chapter 4. To do that, we must use the `ACCOUNTADMIN` role to create a storage integration object named `PARK_INN_INTEGRATION`. Please refer to chapter 4 for detailed instructions on creating this object if you haven't done so already. If you already created this object previously, you don't have to recreate it.

Grant usage privileges on the `PARK_INN_INTEGRATION` storage integration object to the `DATA_ENGINEER` role because we are using this role to build the data pipeline (in chapter 4, we granted usage to the `SYSADMIN` role because we didn't have the `DATA_ENGINEER` role at the time):

```
use role ACCOUNTADMIN;
grant usage on integration PARK_INN_INTEGRATION to role DATA_ENGINEER;
```

Then switch to the `DATA_ENGINEER` role, and use the `BAKERY_DB` database and the `EXT` schema:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema EXT;
```

Create an external stage named JSON_ORDERS_STAGE using the PARK_INN_INTEGRATION storage integration as described in chapter 4. Upload the sample JSON files Orders_2023-09-01.json and Orders_2023-09-04.json from the GitHub repository in the Chapter_11 folder to the object storage location used in the stage.

To verify that the storage integration and the stage are set up correctly, view the files in the stage by executing the LIST command:

```
list @JSON_ORDERS_STAGE;
```

In the output of this command, you should see the two files you uploaded. Create a table that will store the order information in raw (JSON) format (refer to chapter 4 for more details if needed):

```
create table JSON_ORDERS_EXT (
  customer_orders variant,
  source_file_name varchar,
  load_ts timestamp
);
```

Load the data from the files in the stage into the JSON_ORDERS_EXT table:

```
copy into JSON_ORDERS_EXT
from (
  select
    $1,
    metadata$filename,
    current_timestamp()
  from @JSON_ORDERS_STAGE
)
on_error = abort_statement
;
```

To verify that the data was loaded as expected, we can select from the JSON_ORDERS_EXT table:

```
select * from JSON_ORDERS_EXT;
```

The output of this command is shown in table 11.1.

Table 11.1 Output when selecting from the JSON_ORDERS_EXT table (the first column is abbreviated)

Customer orders	Source filename	Load timestamp
{"Customer":"Park Inn", "Order date":"2023-09-01","Orders": [{"Delivery date": "2023-09-04","Orders by day":…	Orders_2023-09-01.json	2023-08-29 00:11:45.501

Table 11.1 Output when selecting from the JSON_ORDERS_EXT table (the first column is abbreviated) *(continued)*

Customer orders	Source filename	Load timestamp
{"Customer":"Park Inn", "Order date":"2023-09-04","Orders": [{"Delivery date": "2023-09-05","Orders by day":…	Orders_2023-09-01.json	2023-08-29 00:11:45.501

The extraction layer in this example uses the ELT pattern because data is extracted from the JSON files and stored in Snowflake in its raw format. It will be transformed in the next layer.

11.2.2 Implementing the staging layer

The staging layer stores data from both data sources, the transformed data from the JSON files, and the tables from the bakery's operational system.

TRANSFORMED JSON DATA

Let's start with the JSON files. We will transform the data by flattening the JSON structure into a relational table format. Using the STG schema, we will create a view named JSON_ORDERS_STG with the flattened data (refer to chapter 4 for more information about the view logic if needed):

```
use database BAKERY_DB;
use schema STG;

create view JSON_ORDERS_STG as
select
  E.customer_orders:"Customer"::varchar as customer,
  E.customer_orders:"Order date"::date as order_date,
  CO.value:"Delivery date"::date as delivery_date,
  DO.value:"Baked good type":: varchar as baked_good_type,
  DO.value:"Quantity"::number as quantity,
  source_file_name,
  load_ts
from EXT.JSON_ORDERS_EXT E,
lateral flatten (input => customer_orders:"Orders") CO,
lateral flatten (input => CO.value:"Orders by day") DO;
```

To verify that the data in the view has the expected format, we can query the view:

```
select *
from JSON_ORDERS_STG;
```

The output from this view is shown in table 11.2 (not all data is shown).

Table 11.2 Output when selecting from the JSON_ORDERS_STG view

Customer	Order date	Delivery date	Baked good type	Quantity	Source filename
Park Inn	2023-09-01	2023-09-04	English muffin	30	Orders_2023-09-01.json
Park Inn	2023-09-01	2023-09-04	Whole wheat loaf	6	Orders_2023-09-01.json
Park Inn	2023-09-01	2023-09-04	White loaf	4	Orders_2023-09-01.json

TABLES FROM THE OPERATIONAL SYSTEM

Assuming that the bakery uses a data integration tool to extract data from the operational system, we will create two tables in the STG schema that simulate tables that would be created by the data integration tool. The two tables, named PARTNER and PRODUCT respectively, contain information about the bakery's business partners and details about the baked goods.

Let's create these tables and populate them with some sample data (not all sample data is shown; use the full code in the Chapter_11_Part4_STG_layer_from_database.sql file in the Chapter_11 folder in the GitHub repository when populating the tables):

```
create table PARTNER (
partner_id integer,
partner_name varchar,
address varchar,
rating varchar
);

insert into PARTNER values
(101, 'Coffee Pocket', '501 Courtney Wells', 'A'),
(102, 'Lily''s Coffee', '2825 Joshua Forest', 'A'),
...
(112, 'Murphy Mill', '700 Darren Centers', 'A');

create table PRODUCT (
product_id integer,
product_name varchar,
category varchar,
min_quantity integer,
price number(18,2),
valid_from date
);

insert into PRODUCT values
(1, 'Baguette', 'Bread', 2, 2.5, '2023-06-01'),
(2, 'Bagel', 'Bread', 6, 1.3, '2023-06-01'),
...
(12, 'Chocolate Muffin', 'Pastry', 12, 3.6, '2023-06-01');
```

The tables PARTNER and PRODUCT should now be created and populated with sample data. The sample data includes an ID column (PARTNER_ID in the PARTNER table and PRODUCT_ID in the PRODUCT table) representing the primary key in each table.

No further action is needed from the data engineer because the maintenance of these tables is done by the data integration tool.

11.2.3 Implementing the data warehouse layer

The data warehouse layer combines data from all sources in one unified data model. To build this layer, we will use a normalized relational data model and will include all data from the staging layer.

We can use the PARTNER and PRODUCT tables directly from the staging layer because they don't require any additional transformations. We will create views in the data warehouse layer that take the data from the staging layer:

```
use database BAKERY_DB;
use schema DWH;

create view PARTNER as
select partner_id, partner_name, address, rating
from STG.PARTNER;

create view PRODUCT as
select product_id, product_name, category, min_quantity, price, valid_from
from STG.PRODUCT;
```

In actual data warehousing implementations, we sometimes encounter situations where data is taken directly from the source without applying additional transformations. Although creating views that select data without transforming it may appear redundant, it's better to have a consistent data flow between the layers and all data available in one place in the data warehousing layer.

To bring the order information into the data warehousing layer and ensure the data is integrated, we will join the JSON_ORDERS_STG staging table with the PARTNER and PRODUCT tables in the staging layer to derive the primary keys from these tables. This is needed for a proper normalized relational data model so that the data can be joined by primary keys later in the reporting layer.

For example, when we examine the data in the JSON_ORDERS_STG table as shown in table 11.2., we see that the customer is indicated by its name, "Park Inn." To combine this data with the PARTNER table, we must match the customer name in the JSON_ORDERS_STG table with the partner name in the PARTNER table to get the PARTNER_ID primary key column.

Similarly, we must match the baked good type in the JSON_ORDERS_STG table with the product name in the PRODUCT table to get the PRODUCT_ID primary key column.

> **NOTE** In real scenarios, you might encounter customer or product names in the ORDERS table that don't match any names in the PARTNER and PRODUCT tables. To address this, you could add data quality tests into your pipeline as described in chapter 14.

We will create a view named ORDERS that joins the JSON_ORDERS_STG table with the PARTNER and PRODUCT tables, deriving the primary keys as described previously, by executing the following command:

```
create view ORDERS as
select PT.partner_id, PRD.product_id, ORD.delivery_date,
  ORD.order_date, ORD.quantity
from STG.JSON_ORDERS_STG ORD
inner join STG.PARTNER PT
  on PT.partner_name = ORD.customer
inner join STG.PRODUCT PRD
  on PRD.product_name = ORD.baked_good_type;
```

Let's examine the data in the view by selecting from it:

```
select *
from ORDERS;
```

The output of this command is shown in table 11.3 (not all data is shown).

Table 11.3 Output when selecting from the ORDERS view

Partner ID	Product ID	Delivery date	Order date	Quantity
109	2	2023-09-04	2023-09-01	25
109	2	2023-09-05	2023-09-04	22
109	3	2023-09-04	2023-09-01	30

We now have three views in the data warehouse layer—ORDERS, PARTNER, and PRODUCT—structured in a relational data model, as shown in figure 11.5.

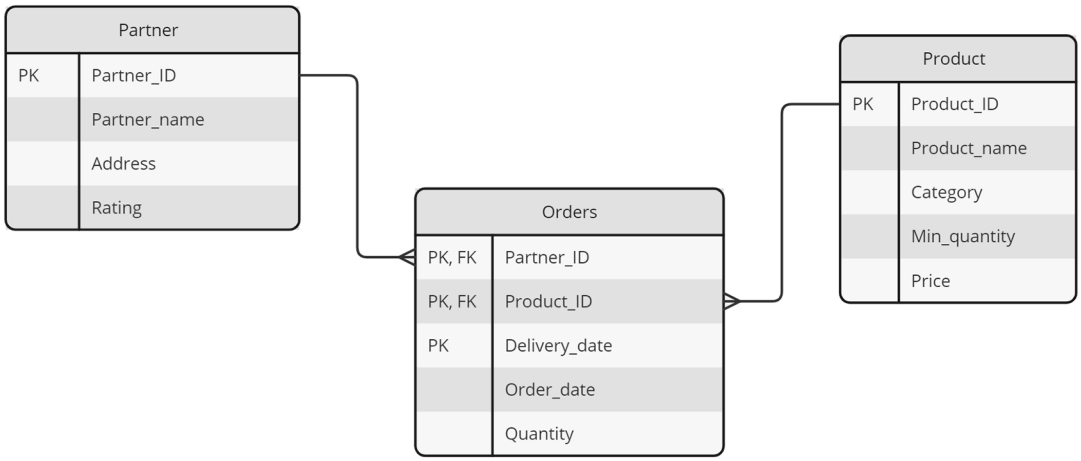


Figure 11.5 Normalized data warehouse data model with the ORDERS, PARTNER, and PRODUCT tables

NOTE The data model is implemented by views in this example. The primary and foreign keys are shown as a logical representation since views don't have physical constraints such as primary or foreign keys.

TIP When writing transformation queries, it's a best practice to select from tables or views in the same layer. For example, the `ORDERS` view joins tables from the `STG` layer. To keep the data flows neatly separated by layers, we don't mix and match by joining tables or views from different layers.

11.2.4 Implementing the reporting layer

The bakery manager uses the reporting layer to learn how many of each type of baked good are ordered in the following days. For this purpose, the data engineer will create a view in the MGMT schema that summarizes orders by the delivery date and baked goods type, adding the baked goods category:

```
use database BAKERY_DB;
use schema MGMT;

create view ORDERS_SUMMARY as
select ORD.delivery_date, PRD.product_name, PRD.category,
  sum(ORD.quantity) as total_quantity
from dwh.ORDERS ORD
left join dwh.PRODUCT PRD
on ORD.product_id = PRD.product_id
group by all;
```

Once the view is created, the bakery manager will use the `DATA_ANALYST` role to select data from the view:

```
use role DATA_ANALYST;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema MGMT;

select * from ORDERS_SUMMARY;
```

The output of this query is shown in table 11.4 (not all data is shown).

Table 11.4 Output when selecting from the `ORDERS_SUMMARY` view

Delivery date	Product name	Category	Total quantity
2023-09-04	Bagel	Bread	25
2023-09-05	Bagel	Bread	22
2023-09-04	English muffin	Bread	30
2023-09-04	Croissant	Pastry	36
2023-09-06	Croissant	Pastry	25

In the data pipeline illustrated in this chapter, we primarily use views because the amount of data is small, and we don't have to worry about query performance. However, in real data transformation pipelines, the data amounts can be massive, and we often can't get away with views. Instead, we must implement data pipelines in ways that don't process all data every time we execute the pipeline.

One way to limit the amount of data processed during pipeline execution is to ingest data incrementally. We will expand on the data pipeline from this chapter by adding incremental data ingestion in the next chapter.

Summary

- A data pipeline is an automated sequence of steps designed to support the extraction, movement, ingestion, transformation, storage, and presentation of data from a source to a target platform.
- Most data sources in organizations are operational systems that typically store data in databases. A common approach to ingesting data from operational system databases into Snowflake is to use third-party data integration tools or build a custom application.
- Once data has been ingested from a source database or file into Snowflake, it must be formatted, cleaned, and transformed to be valuable to downstream consumers.
- The ETL data pipeline pattern extracts data from source systems, such as operational databases or files; formats, cleans, and transforms it according to business requirements; and, finally, loads it into the target tables in Snowflake.
- The ELT data pipeline pattern extracts data from source systems without any transformations and stores it in a designated database in Snowflake. Then data is transformed inside Snowflake, using the full power of Snowflake compute resources.
- The ETLT data pipeline pattern combines the ETL and ELT patterns. The first transformation is for simple transformations that are performed before loading the data. The second transformation performs more complex business transformations that benefit from using Snowflake compute resources.
- As data flows through the pipeline, it produces intermediate outputs that can be stored physically in objects such as database tables, materialized views, or dynamic tables. Data can also be transformed virtually by saving the intermediate outputs as database views. It's a best practice to organize the outputs of the pipeline steps in layers.
- The most common data transformation layers are the extraction layer, the staging layer, the data warehouse layer, and the presentation layer. None of the layers are mandatory, and more layers can be added if needed.
- When implementing data pipelines, it's a best practice to keep the data flows neatly separated by layers, rather than joining tables or views from different layers.

12 Ingesting data incrementally

This chapter covers

- Comparing data ingestion approaches
- Preserving history with slowly changing dimensions
- Detecting changes with Snowflake streams
- Maintaining data with dynamic tables
- Querying historical data

In previous chapters, we built data pipelines that handled small amounts of data, and we didn't consider performance or regular pipeline execution. However, in real-world scenarios, data engineers usually deal with large data volumes that require additional considerations in pipeline design, such as avoiding processing all data every time the pipeline executes. One way to limit the processed data volume during pipeline execution is to ingest data incrementally.

Incremental data ingestion is faster than full ingestion, as it involves moving less data, resulting in lower storage and compute costs. Virtual warehouses require less time to process the data and consume fewer credits. Additionally, the intermediate data pipeline layers store less data.

In this chapter, we will design data pipelines using different methods of incremental data ingestion. We will describe slowly changing dimensions used to track historical data in data warehouses and how to simplify ingestion with append-only strategies. We will describe how to utilize Snowflake streams to detect changes in source data and incorporate them into data pipelines. We will use dynamic tables to maintain data in the data warehouse. Finally, we will explain how to query tables that store historical data.

To illustrate the examples in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries and delivers these baked goods to small businesses, such as grocery stores, coffee shops, restaurants, and hotels in the neighborhood. In chapter 11, we built a data pipeline that ingests data from two sources and transforms the data through the extraction, staging, data warehouse, and presentation layers.

One of the data sources is order information from the bakery's customers, stored as JSON files in cloud storage. The data from these files is copied into Snowflake tables in the extraction layer. Then the JSON data is flattened into a relational structure in the staging layer.

Another data source is the bakery's operational IT system database, which stores its core business data, including details about the business partners and baked goods. A data integration tool ingests this data into the staging layer.

The data from the staging layer is normalized in the data warehouse layer. Finally, the data is summarized and presented for reporting in the presentation layer. Figure 12.1 illustrates the data pipeline steps and the corresponding database objects in each layer.

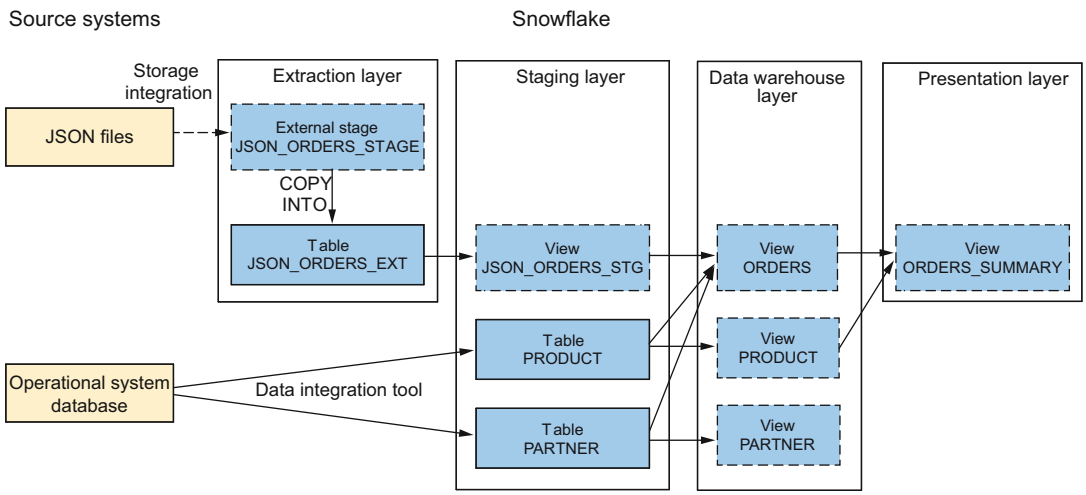


Figure 12.1 Data pipeline that ingests order data from JSON files stored in object storage and details about business partners and baked goods from the bakery's operational IT system database. The JSON data is flattened into a relational structure in the staging layer, normalized in the data warehouse layer, and summarized and presented for reporting in the presentation layer.

In this chapter, we will enhance this pipeline to ingest data incrementally.

> **NOTE** All code and sample data files for this chapter are available in the accompanying GitHub repository in the Chapter_12 folder at https://mng.bz/q0VE.
>
> The SQL code is stored in multiple files whose names start with Chapter_12_Part*, where * indicates a sequence number and additional information refers to the file's content. Please follow the exercises by reading the files sequentially.
>
> The exercises in this chapter use objects created in chapter 11, so you must perform all the exercises in chapter 11 before continuing with the exercises in this chapter.

12.1 Comparing data ingestion approaches

Incremental data ingestion refers to copying only new or changed data from the source systems to the data warehouse instead of full ingestion, which copies all data. In typical data warehouse scenarios, we perform an initial data ingestion, copying all data from the source systems the first time we execute the data pipeline. This ensures that all the historical data is available in the data warehouse, and full ingestion is required.

Data in the source systems usually changes over time—especially data that originates from operational IT systems used by the business to support its daily operations. New data is added, and existing data is updated or deleted. Data pipelines are executed regularly to keep up with the changes in the source systems and update the data in the data warehouse. There are different ways of ensuring that data is in sync, as described in the following sections.

12.1.1 Full ingestion

When data volumes are low and users don't require tracking historical data changes, data pipelines can perform full ingestion on each scheduled execution. In this scenario, the data pipeline replaces all data in the data warehouse with the data from the source systems. The process is simple and doesn't require any particular logic to implement, as it replaces all data either in the form of a view or by truncating the old data and inserting new data into a table.

However, full ingestion doesn't work well with large data volumes, as it performs slower and consumes more resources. Because historical changes in data are not preserved, there is a risk of losing data. If some data is deleted in the source system, it will also disappear in the data warehouse upon overwriting. In addition, full ingestion consumes needless resources because the same historical data is overwritten each time the pipeline is executed.

Most data warehousing solutions use incremental ingestion for regularly scheduled executions. This approach ensures better performance and reduces resource consumption while preserving historical data changes.

12.1.2 Incremental ingestion

With incremental ingestion, we identify data that has been added or changed in the source system since the last pipeline execution. Users sometimes delete data in the source system, but we typically don't delete data in the data warehouse because preserving history is usually preferred. Instead of deleting data in the data warehouse, we treat a deletion as a change and update the record with a status or flag indicating that the data is no longer valid.

Because only the new or modified data is copied, incremental ingestion is more efficient than completely reloading the entire dataset each time. However, incremental ingestion requires more logic and complexity in the pipeline design, which can make it challenging to maintain and guarantee data consistency, particularly if a pipeline execution fails.

Table 12.1 shows a comparison between full and incremental ingestion.

Table 12.1 Comparison between full and incremental ingestion

	Full ingestion	Incremental ingestion
Data scope	Ingests all data and inserts into the target tables	Ingests only new or changed data and appends or updates the target tables
Performance	Executes longer because all data is processed	Executes faster because fewer records are processed
Resource utilization	Consumes more resources, both in storage and compute	Consumes fewer resources
Code design	Simple because all data is ingested without additional logic	Complex because incremental changes are applied to the data in the target tables, and there is a greater chance of data inconsistencies if a pipeline fails
Storing historical changes in data	Historical changes are not preserved because data is overwritten.	Can preserve historical changes, depending on the design
Maintenance	Easy because data is overwritten every time	Can be challenging to troubleshoot, depending on the design
Ingestion frequency	Usually scheduled less frequently because it takes longer to execute	Can be scheduled more frequently because it takes less time to execute

With full ingestion, the data is overwritten, and historical changes are not preserved. In contrast, with incremental ingestion, historical data changes may or may not be preserved depending on the users' requirements. When users don't need historical data changes, the incremental data can be merged into the target tables, overwriting new values with previous values so that only the latest value is available. However, when users want to preserve historical changes, a commonly used design is slowly changing dimensions, as described in the next section.

12.2 Preserving history with slowly changing dimensions

Slowly changing dimensions (SCD) is a widely used design pattern for maintaining historical changes in data. It is most useful when the source system provides information about the last time the data changed. Slowly changing dimensions work with different patterns, such as SCD1, SCD2, and others.

Data warehouses usually work with the SCD2 pattern. With this pattern, each record in a table contains two timestamp columns that indicate the time interval during which the record is valid. If the record is currently valid and we don't know when the validity expires, we use a date value far in the future to denote the end of the validity interval.

12.2.1 SCD type 2

Let's illustrate the SCD2 pattern with an example. Say we initially ingested data from the `PARTNER` table into the data warehouse on June 1, 2023. Each record in this table has a validity interval from June 1, 2023 (when the data warehouse first saw the record) until December 31, 9999 (a date far in the future). One sample record from this table is shown in table 12.2.

Table 12.2 Sample data in the `PARTNER` table after initial ingestion

Partner ID	Partner name	Rating	Valid from	Valid to
103	Crave Coffee	B	1.6.2023	31.12.9999

As shown in table 12.2, the Crave Coffee business partner has the rating value B, which has been valid since June 1, 2023, and is still valid.

Then the business partner's rating changed to A on August 8, 2023. According to the SCD2 pattern, we insert a new row into the `PARTNER` table with the new rating value. This new row is valid from August 8, 2023, until a date far into the future. In addition to inserting this new row, which is now valid, we must update the validity of the previous row to indicate that the B rating was valid until August 8, 2023.

Table 12.3 shows the two rows in the `PARTNER` table. The first row is the same as the initial row in table 12.2; only the end date of the validity period is updated. The second row is the newly inserted row after the rating was changed from B to A in the source system.

Table 12.3 Sample data in the `PARTNER` table according to the SCD2 pattern

Partner ID	Partner name	Rating	Valid from	Valid to
103	Crave Coffee	B	1.6.2023	8.8.2023
103	Crave Coffee	A	8.8.2023	31.12.9999

Since there are now two rows in the table for the same business partner, the `Partner ID` column is not unique and cannot be defined as the primary key in the table. A

composite key containing the `Partner ID` column and the `Valid from` column is unique in the table because each row always includes the date when the data changed. This logic applies generally across all SCD2 implementations: the primary key in an SCD2 table is a composite key made up of the unique identifier of the entity whose history we are tracking and the date (or timestamp) of the beginning of the validity period.

Implementing the SCD2 pattern requires inserting new rows and updating previous rows to end their validity. This algorithm can be cumbersome to maintain, particularly when troubleshooting pipeline failures to reconstruct the correct timelines. Additionally, when updating data in Snowflake, the micro-partition containing the changed data must be rewritten, which can impact performance.

An *append-only* strategy that preserves historical data changes but avoids updating data is an alternative to the classic SCD2 pattern.

12.2.2 Append-only strategy

With the append-only strategy, rows are inserted into the table when data changes. Instead of a validity interval indicated by the start and end timestamp, only the start timestamp is stored in each row in the table. There is no need to store the end timestamp of the validity interval because it can easily be calculated. The end timestamp of each row is equal to the start timestamp of the following row, except for the last row, which has no following row. This last row represents the record currently valid; therefore, it has a date far in the future as the end timestamp.

For example, after initially ingesting data from the `PARTNER` table into the data warehouse on June 1, 2023, each row has the validity start timestamp set to June 1, 2023, and no column indicates the end timestamp.

When a business partner's rating changed from B to A on August 8, 2023, we inserted a new row into the `PARTNER` table with the new rating value, and the start timestamp is August 8, 2023. Since there is no end timestamp for the validity interval, we don't have to update anything. Sample data after the initial ingestion and the rating change according to the append-only strategy is shown in table 12.4.

Table 12.4 Sample data in the `PARTNER` table with the append-only strategy

Partner ID	Partner name	Rating	Valid from
103	Crave Coffee	B	1.6.2023
103	Crave Coffee	A	8.8.2023

We can calculate the end timestamp of the validity interval of each row using the SQL `LEAD()` window function. For each row within each Partner ID group, we calculate the end timestamp by taking the start timestamp from the following row. We also apply the `NVL()` function, which replaces the `NULL` value of the end timestamp of the last row

(which has no following row) with a date far in the future. The query that calculates the end timestamp is

```
select
  partner_id,
  partner_name,
  rating,
  valid_from,
  NVL(
    LEAD(valid_from) over (partition by partner_id order by valid_from),
    '9999-12-31'
  ) as valid_to
from PARTNER;
```

> **NOTE** This query is shown for illustration purposes and is not part of the exercises in this chapter.

The output of this query should be the same as the data shown in table 12.3. The query that calculates the end timestamp can be implemented as a view presented to the downstream consumers or materialized for better performance.

The append-only strategy preserves history, like in the SCD2 pattern, but doesn't require updates, making the logic simpler and easier to maintain.

12.2.3 Designing idempotent data pipelines

When designing data pipelines that ingest data incrementally, it's important to consider how the pipeline behaves when it fails and must be restarted. Ideally, the pipeline should be robust and designed to produce consistent data or not require too much effort to troubleshoot.

Data pipelines should be designed to be idempotent, meaning they result in the same data without duplicates, gaps, or inconsistencies, even when the pipeline is executed multiple times.

To prevent data inconsistencies and ensure that pipelines perform idempotently, follow these design guidelines:

- Merge data into target tables based on unique identifiers.
- Avoid duplication by calculating hash values of rows in tables and checking that rows with the same hash value are not ingested again.
- Ingest data that has been created or changed after the latest ingestion timestamp.

12.3 Detecting changes with Snowflake streams

To implement incremental ingestion, we must be able to identify new or changed data in the source systems. Snowflake provides *streams* that track changes resulting from data manipulation language (DML) operations, such as inserting, updating, or deleting data. Streams also keep metadata about the operations.

> **NOTE** More information about Snowflake streams is available in the documentation at https://mng.bz/75gg.

Streams can be created on standard tables, external tables, underlying tables for a view, or directory tables. When a stream is created, it takes a snapshot of the object on which the stream was created at that point in time. Each subsequent SQL statement that inserts, updates, or deletes data in the underlying object is reflected in the stream.

When a stream is created, hidden columns are added to the source table or the underlying tables of a view that store change tracking metadata. These columns are

- `METADATA$ACTION`—Contains the name of the DML operation executed. This column contains either the value `INSERT` or `DELETE`. When an `UPDATE` operation is performed on the table, the stream records the change as a `DELETE` of the previous version of the data before it was updated and an `INSERT` with the new version of the data.
- `METADATA$ISUPDATE`—Contains a Boolean `TRUE` or `FALSE` flag indicating whether the operation was part of an UPDATE statement.
- `METADATA$ROW_ID`—Contains the unique row ID.

Streams record only the difference between two versions of a table. If more than one DML operation is executed on a row—for example, if a row is added and updated—the stream records only the difference between the previous and current versions but not any changes in between.

NOTE Streams don't contain any table data. They only keep track of changes in the underlying objects.

We can view the contents of a stream by querying it with a SQL `SELECT` statement to see the changes in underlying data since the stream was created or last consumed. A stream is *consumed* when data is written to one or more target tables within a transaction. When the transaction is committed, the stream becomes empty and starts accumulating new changes from that point in time.

12.3.1 Ingesting files from cloud storage incrementally

Let's illustrate how we can use a stream to detect changes in our data pipeline from chapter 11. Order information is stored in JSON files in cloud storage and copied into the `JSON_ORDERS_EXT` table in the extraction layer. A view named `JSON_ORDERS_STG` flattens the data in the staging layer.

To implement incremental ingestion, we will no longer use the `JSON_ORDERS_STG` view (because a view processes all data) but will create a table named `JSON_ORDERS_TBL_STG` instead. We will insert flattened JSON data incrementally into this table by taking data only from the new files that arrived in cloud storage since the previous pipeline execution. We will create a stream named `JSON_ORDERS_STREAM` on top of the `JSON_ORDERS_EXT` table in the extraction layer to detect the newly arrived data. The objects used in this pipeline are shown in figure 12.2.

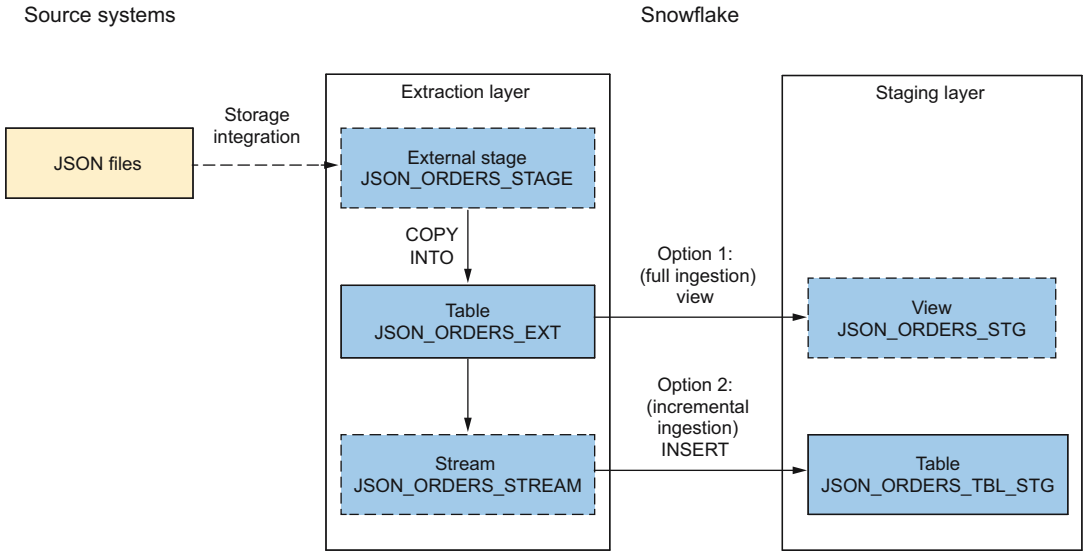


Figure 12.2 Two ways to ingest data into the staging layer. The first option represents full ingestion implemented as a view in the staging layer; the second option uses a Snowflake stream to detect changes and inserts only changed data into the staging table.

We will use the `DATA_ENGINEER` role, the `BAKERY_WH` warehouse, and the `BAKERY_DB` database for this pipeline, as in chapter 11. We will execute the first step in the `EXT` schema:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema EXT;
```

To make it easier to follow along with the exercise and see the incremental changes in the data, we will remove some of the data we used in the exercises in previous chapters and upload new data. To clean the environment, perform the following steps:

- Remove all files from the external storage location where the `JSON_ORDERS_STAGE` stage is pointing using your preferred method of working with files in your object storage provider.
- Upload a new file named Orders_2023-09-05.json from the Chapter_12 folder in the GitHub repository to the external storage location.

WARNING Upload just the one JSON file for now, even though there are additional files in the repository. You will upload more files later to test incremental ingestion.

We will need a table in the extraction layer to store the data from the files in object storage and a stream on this table to help us detect changes. We can reuse the `JSON_ORDERS_EXT` table from the previous exercises, but we will recreate the table to remove any data

already stored in the table (alternatively, we could truncate the table). To recreate the JSON_ORDERS_EXT table, we can execute the following command:

```
create or replace table JSON_ORDERS_EXT (
  customer_orders variant,
  source_file_name varchar,
  load_ts timestamp
);
```

Then we will create a stream named JSON_ORDERS_STREAM on the JSON_ORDERS_EXT table using the following command:

```
create stream JSON_ORDERS_STREAM
on table JSON_ORDERS_EXT;
```

We can view the data in the stream by executing the SQL SELECT command as follows:

```
select * from JSON_ORDERS_STREAM;
```

Because the stream was just created, the output of this command should show that the stream is empty.

We will copy the data from the file in the object storage location we uploaded to the JSON_ORDERS_EXT table. But first let's view the files in the stage to verify that the file is there:

```
list @JSON_ORDERS_STAGE;
```

The LIST command should show that the file Orders_2023-09-05.json is in the stage.

To copy the data from the stage into the JSON_ORDERS_EXT table, we can execute the COPY command:

```
copy into JSON_ORDERS_EXT
from (
  select
    $1,
    metadata$filename,
    current_timestamp()
  from @JSON_ORDERS_STAGE
)
on_error = abort_statement;
```

The output from the COPY command should indicate that data from the Orders_2023-09-05.json file was copied into the table.

Let's recheck the data in the stream by querying it:

```
select * from JSON_ORDERS_STREAM;
```

The stream should now contain the newly uploaded file, indicating that this change occurred in the underlying table. The output of this command is shown in table 12.5

(the `Customer orders` column is abbreviated, and the `METADATA$ROW_ID` column is omitted).

Table 12.5 Contents of the `JSON_ORDERS_STREAM` after copying data into the `JSON_ORDERS_EXT` table

Customer orders	Source filename	Load timestamp	METADATA$ ACTION	METADATA$ ISUPDATE
{"Customer": "Park Inn", "Order date": "2023-09-05", "Orders": […	Orders_2023-09-05.json	2023-09-10 12:26:17	INSERT	FALSE

To consume the stream, we will insert flattened data from the stream's underlying table into the `JSON_ORDERS_TBL_STG` staging table.

We will create the `JSON_ORDERS_TBL_STG` staging table in the `STG` schema by executing the following command:

```
create table STG.JSON_ORDERS_TBL_STG (
  customer varchar,
  order_date date,
  delivery_date date,
  baked_good_type varchar,
  quantity number,
  source_file_name varchar,
  load_ts timestamp
);
```

We will insert the flattened data from the stream into the staging table by executing the following command (refer to chapter 11 for an explanation of the query logic if needed):

```
insert into STG.JSON_ORDERS_TBL_STG
select
  customer_orders:"Customer"::varchar as customer,
  customer_orders:"Order date"::date as order_date,
  CO.value:"Delivery date"::date as delivery_date,
  DO.value:"Baked good type":: varchar as baked_good_type,
  DO.value:"Quantity"::number as quantity,
  source_file_name,
  load_ts
from EXT.JSON_ORDERS_STREAM,
lateral flatten (input => customer_orders:"Orders") CO,
lateral flatten (input => CO.value:"Orders by day") DO;
```

After inserting, we can view the data in the table by querying it:

```
select * from STG.JSON_ORDERS_TBL_STG;
```

The query should return data as shown in table 12.6 (the first few rows are shown).

Table 12.6 Contents of the JSON_ORDERS_TBL_STG staging table

Customer	Order date	Delivery date	Baked good type	Quantity	Source filename	Load timestamp
Park Inn	2023-09-05	2023-09-06	English muffin	12	Orders_2023-09-05.json	2023-09-10 12:26:17
Park Inn	2023-09-05	2023-09-06	Whole wheat loaf	6	Orders_2023-09-05.json	2023-09-10 12:26:17
Park Inn	2023-09-05	2023-09-06	Blueberry muffin	12	Orders_2023-09-05.json	2023-09-10 12:26:17

We can check the data in the stream again:

```
select * from JSON_ORDERS_STREAM;
```

The stream should now be empty because the INSERT statement consumed it.

Let's repeat with another incremental ingestion. Upload the file named Orders_2023-09-06.json from the Chapter_12 folder in the GitHub repository to the object storage location.

Execute the COPY command again to copy the data from the object storage location to the JSON_ORDERS_EXT table in the extraction layer:

```
copy into JSON_ORDERS_EXT
from (
  select
    $1,
    metadata$filename,
    current_timestamp()
  from @JSON_ORDERS_STAGE
)
on_error = abort_statement;
```

The output from the COPY command should indicate that data from the Orders_2023-09-06.json file was copied into the table. Although we have two files in the object storage location, the first one ingested in the previous step is not ingested again; only the newly uploaded file is ingested.

> **NOTE** We learned in chapter 3 that the COPY command tracks metadata for each table into which data is loaded from staged files. This guarantees that subsequent or simultaneous executions of the COPY command don't load data from the same file again so that data is not duplicated. Therefore, the COPY command behaves like incremental ingestion, and we don't have to do anything special in the pipeline to make it incremental.

After copying the data from the object storage location into the JSON_ORDERS_EXT table in the extraction layer, we can recheck the contents of the stream:

```
select * from JSON_ORDERS_STREAM;
```

The output of this command shows the contents of the last file that was ingested.

We will consume the stream by inserting data from the stream's underlying table into the JSON_ORDERS_TBL_STG table in the staging layer:

```
insert into STG.JSON_ORDERS_TBL_STG
select
  customer_orders:"Customer"::varchar as customer,
  customer_orders:"Order date"::date as order_date,
  CO.value:"Delivery date"::date as delivery_date,
  DO.value:"Baked good type":: varchar as baked_good_type,
  DO.value:"Quantity"::number as quantity,
  source_file_name,
  load_ts
from EXT.JSON_ORDERS_STREAM,
lateral flatten (input => customer_orders:"Orders") CO,
lateral flatten (input => CO.value:"Orders by day") DO;
```

After inserting data into the underlying table, we can recheck the contents of the stream to see if it is empty after it is consumed.

12.3.2 Preserving history when ingesting data incrementally

In addition to files with information about customer orders in cloud storage that we ingested incrementally in the previous section, we will look at ingesting data from the bakery's operational system incrementally.

We assume the operational system data is ingested via a data integration tool into the staging layer. This tool executes on schedule and updates the data in the staging layer as it is updated in the source system. For this exercise, we will simulate the output of the data integration tool by updating and inserting data in the staging layer via DML commands.

To implement incremental ingestion, we will no longer use the PRODUCT view in the data warehouse layer (because a view processes all data) but will create a table named PRODUCT_TBL instead. Using the append-only strategy described earlier in this chapter, we will insert data from the PRODUCT table in the staging layer into the PRODUCT_TBL table. We will create a view named PRODUCT_VALID_TS in the data warehouse layer that calculates the end timestamp of each row's validity interval. The objects used in this pipeline are shown in figure 12.3.

Let's start this exercise in the data warehouse layer:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema DWH;
```

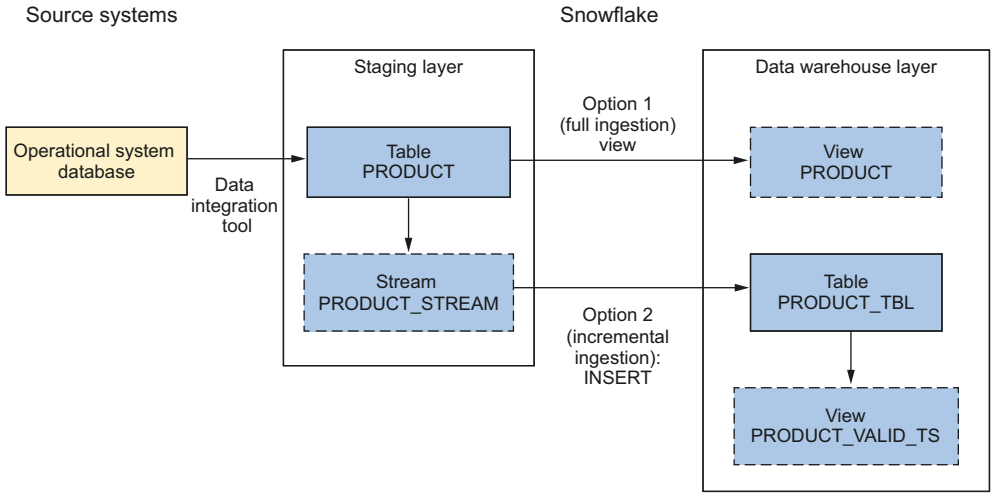


Figure 12.3 Two ways to ingest data into the data warehouse layer. The first option represents full ingestion implemented as a view in the data warehouse layer; the second option uses a table into which data is inserted using the append-only strategy and a view that calculates the end timestamp of each row's validity interval.

We will create a table named `PRODUCT_TBL` and populate it initially with the data from the `PRODUCT` table in the staging layer:

```
create table PRODUCT_TBL as
select * from STG.PRODUCT;
```

To view the data in the newly created table, we can query it with the `SELECT` command:

```
select * from PRODUCT_TBL;
```

The output of this command is shown in table 12.7 (the first few rows are shown).

Table 12.7 Contents of the `PRODUCT_TBL` table

Product ID	Product name	Category	Minimum quantity	Price	Valid from
1	Baguette	Bread	2	2.5	2023-06-01
2	Bagel	Bread	6	1.3	2023-06-01
3	English muffin	Bread	6	1.2	2023-06-01

To implement incremental ingestion, we will create a stream on the `PRODUCT` table in the staging layer:

```
use schema STG;
create stream PRODUCT_STREAM on table PRODUCT;
```

Since the stream was just created, it should be empty (you can check by selecting from it).

Then we will simulate a few changes that might be propagated from the source system into the staging layer using the data ingestion tool. We will update one row in the PRODUCT staging table and insert one new row. For this exercise, we will assume that both actions happened on August 8, 2023.

First, let's update the English Muffin product so that it is no longer in the Bread category but in the Pastry category:

```
update PRODUCT
  set category = 'Pastry', valid_from = '2023-08-08'
  where product_id = 3;
```

Then we will add Sourdough Bread as a new product that the bakery makes:

```
insert into PRODUCT values
  (13, 'Sourdough Bread', 'Bread', 1, 3.6, '2023-08-08');
```

After making these changes in the staging table, we can check the contents of the stream by querying it:

```
select * from PRODUCT_STREAM;
```

As expected, the stream reflects these changes by showing the update and the insert operation as shown in table 12.8 (the METADATA$ROW_ID column is omitted).

Table 12.8 Contents of the PRODUCT_STREAM stream after updating and inserting data in the underlying PRODUCT table

Product ID	Product name	Category	Minimum quantity	Price	Valid from	METADATA$ ACTION	METADATA$ ISUPDATE
3	English muffin	Bread	6	1.2	2023-06-01	DELETE	TRUE
3	English muffin	Pastry	6	1.2	2023-06-01	INSERT	TRUE
13	Sourdough bread	Bread	1	3.6	2023-06-01	INSERT	FALSE

As we can see in table 12.8., two rows in the stream capture the UPDATE operation. The stream shows this operation as a pair of DELETE and INSERT rows, as indicated in the METADATA$ACTION column and with a value of TRUE in the METADATA$ISUPDATE column. One row also represents the INSERT operation, as seen in the METADATA$ACTION column, and it has a value of FALSE in the METADATA$ISUPDATE column.

We can use the information from the stream to make the corresponding changes in the PRODUCT_TBL table in the data warehouse layer. We could perform

the actions exactly as shown in the stream by deleting the old rows and then inserting the new rows. But because we want to preserve history in the target table, we will not delete any rows, and therefore, we will not use the rows with a value of DELETE in the METADATA$ACTION column.

Because we want to use the append-only strategy, we will insert all new rows by filtering for columns with a value of INSERT in the METADATA$ACTION column. To make these changes in the PRODUCT_TBL target table, we will execute the following command:

```
insert into DWH.PRODUCT_TBL
select product_id, product_name, category, min_quantity, price, valid_from
from PRODUCT_STREAM
where METADATA$ACTION = 'INSERT';
```

This command consumes the stream, which means that the stream should now be empty. We can verify this by querying it:

```
select * from PRODUCT_STREAM;
```

The command should return no results because the stream is empty.

Let's see the data in the PRODUCT_TBL target table after inserting the data from the stream:

```
select * from DWH.PRODUCT_TBL;
```

The output of this command is shown in table 12.9 (not all rows are shown).

Table 12.9 Contents of the PRODUCT_TBL target table after inserting from the stream

Product ID	Product name	Category	Minimum quantity	Price	Valid from
1	Baguette	Bread	2	2.5	2023-06-01
2	Bagel	Bread	6	1.3	2023-06-01
3	English muffin	Bread	6	1.2	2023-06-01
3	English muffin	Pastry	6	1.2	2023-08-08
13	Sourdough bread	Bread	1	3.6	2023-08-08

As we can see in table 12.9, the PRODUCT_TBL table now has two rows for English Muffins, one row for Bread, valid from June 1, 2023, and a second row for Pastry, valid from the date of the change, August 8, 2023. There is also one new row for Sourdough Bread, valid from August 8, 2023.

Finally, we can create a view named PRODUCT_VALID_TS in the data warehouse layer that calculates the end timestamp of the validity interval of each row in the PRODUCT_TBL

table, as explained earlier in this chapter. To create the view, execute the following command:

```
create view DWH.PRODUCT_VALID_TS as
select
  product_id,
  product_name,
  category,
  min_quantity,
  price,
  valid_from,
  NVL(
    LEAD(valid_from) over (partition by product_id order by valid_from),
    '9999-12-31'
  ) as valid_to
from DWH.PRODUCT_TBL
order by product_id;
```

To check the data in the view, we can query it:

```
select * from DWH.PRODUCT_VALID_TS;
```

The output of this command is shown in table 12.10 (not all rows are shown).

Table 12.10 Contents of the `PRODUCT_VALID_TS` view

Product ID	Product name	Category	Minimum quantity	Price	Valid from	Valid to
1	Baguette	Bread	2	2.5	2023-06-01	9999-12-31
2	Bagel	Bread	6	1.3	2023-06-01	9999-12-31
3	English muffin	Bread	6	1.2	2023-06-01	2023-08-08
3	English muffin	Pastry	6	1.2	2023-08-08	9999-12-31
13	Sourdough bread	Bread	1	3.6	2023-08-08	9999-12-31

As we can see in table 12.10, each row has two timestamps, indicating the start and end timestamps of the validity period.

In addition to the `PRODUCT` table, we have the `PARTNER` table in the staging layer, which is populated by the bakery's operational system. If we want to ingest data incrementally and preserve history, we can implement the same incremental ingestion pipeline for the `PARTNER` table as for the `PRODUCT` table.

> **NOTE** Since the steps for creating the incremental pipeline that populates the partner data are the same as those for populating the product data, we will not explain them again in this section. The code for the incremental pipeline that populates the partner data is available in the `Chapter_12_Part3_stream_PARTNER.sql` script in the Chapter_12 folder in the GitHub repository, so you can execute the steps on your own.

Figure 12.4 shows the objects in the extraction, staging, and warehouse layers used to build the pipeline with incremental ingestion that we have created so far.

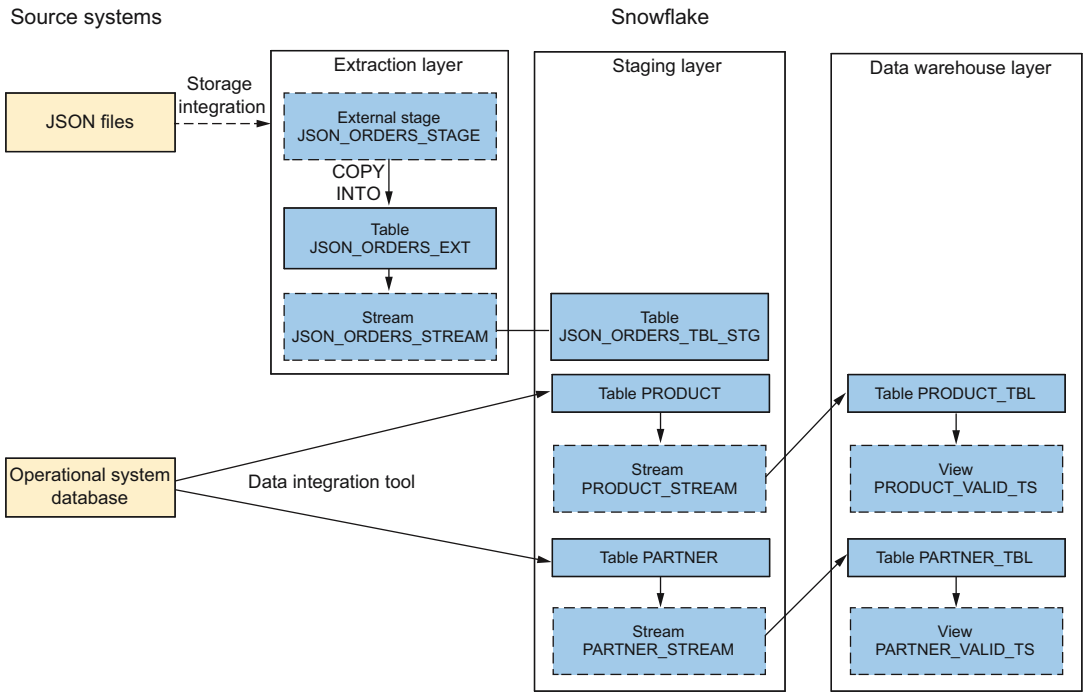


Figure 12.4 Objects in the extraction, staging, and warehouse layers used to build the pipeline with incremental ingestion

To complete the incremental data pipeline to the reporting layer, like in chapter 11, we will combine the order data with the partners and product data in a normalized data model. In chapter 11, we created a view named `ORDERS` in the data warehouse layer by joining the `JSON_ORDERS_STG`, `PRODUCT`, and `PARTNER` tables from the staging layer.

Because we are building an incremental data pipeline, we want to avoid using views that process all data. One way to incrementally ingest the normalized orders data in the data warehouse layer would be to follow an approach similar to the one we used when populating the products and partners data: instead of a view, create a target table that stores the data; then create a stream on the source table to track changes and consume the stream by writing data to the target table.

Although using a stream would be a perfectly valid approach, we also want to demonstrate how Snowflake *dynamic tables* can implement incremental ingestion, as described in the next section.

12.4 Maintaining data with dynamic tables

We introduced Snowflake dynamic tables in chapter 5 as part of continuous ingestion pipelines with Snowpipe. We can also use dynamic tables for incremental data ingestion because they work on the principle that they process only new or changed data behind the scenes rather than populating the full data.

Snowflake defines dynamic tables as the building blocks of declarative data transformation pipelines. Instead of creating a target table and inserting or merging data into it, we define a dynamic table as a query for the results we want to see. We don't have to create a stream to track changes because Snowflake does that in the background.

Let's implement the normalized order data in the data warehouse layer as a dynamic table named `ORDERS_TBL`. First, we will construct a query that joins the `JSON_ORDERS_TBL_STG`, `PRODUCT`, and `PARTNER` tables from the staging layer by joining on the product and partner name columns to derive the corresponding unique identifiers. This is the query that selects the normalized data.

Listing 12.1 Query that selects normalized data from the staging layer

```
use schema DWH;
select PT.partner_id, PRD.product_id, ORD.delivery_date,
  ORD.order_date, ORD.quantity
from STG.JSON_ORDERS_TBL_STG ORD
inner join STG.PARTNER PT
  on PT.partner_name = ORD.customer
inner join STG.PRODUCT PRD
  on PRD.product_name = ORD.baked_good_type;
```

The output of this query is shown in table 12.11 (not all data is shown).

Table 12.11 Output from the query in listing 12.1

Partner ID	Product ID	Delivery date	Order date	Quantity
109	3	2023-09-06	2023-09-05	12
109	8	2023-09-06	2023-09-05	6
109	11	2023-09-06	2023-09-05	12

Now that we have the query constructed, let's create a dynamic table named `ORDERS_TBL` using this query. We will specify 1 minute as the `TARGET_LAG` parameter, meaning that the table should lag by no more than 1 minute whenever the data in any of the tables used in the query changes. We must also specify the virtual warehouse that the dynamic table will use, in our case `BAKERY_WH`. Here is the command that creates this dynamic table using the query from listing 12.1:

```
create dynamic table ORDERS_TBL
  target_lag = '1 minute'
  warehouse = BAKERY_WH
  as
```

```
select PT.partner_id, PRD.product_id, ORD.delivery_date,
  ORD.order_date, ORD.quantity
from STG.JSON_ORDERS_TBL_STG ORD
inner join STG.PARTNER PT
  on PT.partner_name = ORD.customer
inner join STG.PRODUCT PRD
  on PRD.product_name = ORD.baked_good_type;
```

We can select from the dynamic table to see the data:

```
select *
from ORDERS_TBL;
```

Because the dynamic table was just created, it contains the same data returned from the query in listing 12.1, as shown in table 12.11. Whenever data in any of the tables used in the query to create the dynamic table changes, the data in the dynamic table updates accordingly.

12.4.1 Deciding when to use dynamic tables

Dynamic tables are an efficient way to create declarative incremental data pipelines. When deciding whether to use streams for incremental ingestion, as described in the previous section, or dynamic tables, as described in this section, consider the following guidelines. You can use dynamic tables when

- You don't want the complexity of creating, consuming, and scheduling streams.
- You don't need query constructs unsupported in dynamic tables, such as nondeterministic functions, external functions, or queries that read from external tables or materialized views.
- You don't need schedule control of when your data refreshes, and you accept the target data lag in your pipeline.
- You want to materialize the results of a query that joins multiple tables.

TIP Remember that you can't create a materialized view based on a query that joins multiple tables, but you can create a dynamic table instead.

- You want to perform multiple data transformation steps that follow sequentially in a data pipeline.

You shouldn't use dynamic tables in data pipelines where you want to control the schedule when the data refreshes. For example, in many data warehouses, the data might be refreshed on schedule every evening or hour. Users expect that data in all tables reflects a consistent state at a chosen point in time. With dynamic tables, you don't have this control, and one table may already be refreshed with new data while another table is still lagging.

When you have the requirement that all data must be refreshed and consistent at a chosen point in time, you should use streams (or any other means of incremental ingestion) and schedule them according to the requirements.

NOTE For more information about dynamic tables, refer to the Snowflake documentation at https://mng.bz/mRlM.

The final step in our data pipeline is to present summarized data to the bakery manager. The manager wants to see how many of each type of baked goods are ordered by the bakery's customers in the coming days. In chapter 11, we created a view named `ORDERS_SUMMARY` that performs this summarization.

12.4.2 Querying historical data

Now that we are building an incremental data pipeline, we can use a dynamic table again to perform this summarization. But before we create the dynamic table, let's construct the query that we will use in the dynamic table. This query joins the `ORDERS_TBL` and the `PRODUCT_TBL` tables from the data warehouse layer to derive the delivery date, the product name, the product category, and the summarized quantity, as shown in the following listing.

Listing 12.2 A query that summarizes data from the data warehouse layer

```
select ORD.delivery_date, PRD.product_name, PRD.category,
  sum(ORD.quantity) as total_quantity
from dwh.ORDERS_TBL ORD
left join dwh.PRODUCT_TBL PRD
on ORD.product_id = PRD.product_id
group by all;
```

The results of this query are shown in table 12.12 (not all data is shown).

Table 12.12 Output from the query in listing 12.2

Delivery date	Product name	Category	Total quantity
2023-09-06	English muffin	Bread	12
2023-09-07	English muffin	Bread	15
2023-09-08	English muffin	Bread	15
2023-09-06	English muffin	Pastry	12
2023-09-07	English muffin	Pastry	15
2023-09-08	English muffin	Pastry	15

When we look closely at the data returned from the query in listing 12.2 and shown in table 12.12, we notice that the ordered quantities for English muffins are repeated twice for the same delivery dates: once in the bread category and once in the pastry category. The reason for data duplication is that the `PRODUCT_TBL` table used in the query tracks historical changes in data. In the previous section, we saw that the category of the English muffin changed from Bread to Pastry, and as a result, we have two

rows for English muffins in the PRODUCT_TBL table, each with a different date of validity, as seen in table 12.9.

> **WARNING** When querying tables that store historical data, always pay attention to the timeline when selecting data, and provide the necessary filters that return only the data at a given point in time.

When constructing the query that summarizes ordered quantities of baked products for the bakery manager, we must understand the requirements. For example, does the bakery manager want to see the product category that was valid when a customer placed an order for the baked goods? Or does the manager want to see the current category of the baked goods so that the bakery department that currently makes English muffins receives the summarized data?

To query the historical product data at a given point in time, we can use the PRODUCT_VALID_TS view, which contains the start timestamp and the end timestamp of each row in the PRODUCT_TBL table. To select all data that is currently valid, we must add a WHERE clause to the query to filter for rows where the end timestamp is the date far in the future:

```
select * from DWH.PRODUCT_VALID_TS
where valid_to = '9999-12-31';
```

The results of this query should show that the English muffin is in the pastry category because that's currently valid.

If we wanted to see the data that was valid on August 1, 2023 (remember that the English muffin category changed on August 8, 2023, so on August 1, 2023, it should still show the old category before the change), we must add a filtering condition that returns data where the start timestamp is before August 1, 2023 and the end timestamp is after August 1, 2023:

```
select * from DWH.PRODUCT_VALID_TS
where valid_from <= '2023-08-01' and valid_to > '2023-08-01';
```

The results of this query should show that the English muffin is in the bread category, which was valid then.

Assuming the bakery manager wants to see the current baked goods category in the summarized report, we can rewrite the query as in the following listing.

Listing 12.3 Query from listing 12.2 that filters for the current values

```
select ORD.delivery_date, PRD.product_name, PRD.category,
  sum(ORD.quantity) as total_quantity
from dwh.ORDERS_TBL ORD
left join (
  select * from dwh.PRODUCT_VALID_TS where valid_to = '9999-12-31'
) PRD
on ORD.product_id = PRD.product_id
group by all;
```

The output from this query should be similar to the results shown in table 12.12 but without the duplicated records for the English muffins. Only records with the currently valid product category are included.

We can now use the query from listing 12.3 to create a dynamic table named `ORDERS_SUMMARY_TBL` in the `MGMT` schema that summarizes data for the bakery manager with a target lag of 1 minute:

```
use schema MGMT;
create dynamic table ORDERS_SUMMARY_TBL
  target_lag = '1 minute'
  warehouse = BAKERY_WH
  as
select ORD.delivery_date, PRD.product_name, PRD.category,
  sum(ORD.quantity) as total_quantity
from dwh.ORDERS_TBL ORD
left join (select * from dwh.PRODUCT_VALID_TS where valid_to = '9999-12-31') PRD
on ORD.product_id = PRD.product_id
group by all;
```

We can query the dynamic table to verify that it contains the data as expected:

```
select * from ORDERS_SUMMARY_TBL
```

The output of this query should be the same as the output from the query in listing 12.3.

With this step, we completed the implementation of a data pipeline that ingests data from the sources, transforms it, and presents it to the users for reporting. All steps are implemented incrementally, and only new or changed data is processed.

In this chapter, we executed all steps manually by executing commands such as `COPY INTO` or `INSERT`. In the next chapter, we will learn how to orchestrate the steps so the data pipeline executes on schedule without manual intervention.

Summary

- Incremental data ingestion refers to copying only new or changed data from the source systems to the data warehouse instead of full ingestion, which copies all data.
- When data volumes are low and users don't require tracking historical data changes, data pipelines can perform full ingestion on each scheduled execution. However, full ingestion doesn't work well with large data volumes, as it performs more slowly and consumes more resources.
- With incremental ingestion, we identify data that has been added or changed in the source system since the last pipeline execution. Because only the new or modified data is copied, incremental ingestion is more efficient than completely reloading the entire dataset each time.
- SCD is a widely used design pattern for maintaining historical changes in data. With the SCD2 type, each record in a table contains two timestamp columns that indicate the time interval during which the record is valid.

- With the append-only strategy, rows are inserted into the table when data changes. Instead of a validity interval indicated by the start and end timestamps, only the start timestamp is stored in each row. The end timestamp of the validity interval is calculated by taking the start timestamp of the following row.
- Whenever possible, data pipelines should be designed to be idempotent, meaning that they result in the same data without duplicates, gaps, or inconsistencies, even when the pipeline is executed multiple times.
- To implement incremental ingestion, we must be able to identify new or changed data in the source systems. Snowflake provides streams that track changes resulting from DML operations, such as inserting, updating, or deleting data. Streams also keep metadata about the operations.
- Streams don't contain any table data. They only keep track of changes in the underlying objects.
- Snowflake defines dynamic tables as the building blocks of declarative data transformation pipelines. Instead of creating a target table and inserting or merging data into it, we define a dynamic table as a query for the results we want to see. We don't have to create a stream to track changes because Snowflake does that in the background.
- When querying tables that store historical data, always pay attention to the timeline when selecting data, and provide the necessary filters that return only the data at a given point in time.

13 Orchestrating data pipelines

This chapter covers

- Orchestrating data pipelines with Snowflake tasks
- Sending notifications from tasks
- Orchestrating with task graphs
- Monitoring data pipeline execution
- Troubleshooting data pipeline failures

Data pipelines are a series of steps that perform data ingestion and transformation. They are usually scheduled to run at predefined times, often at night, to ensure that business users have fresh data every morning. If users need more recent data, data engineers can schedule the pipelines to run more frequently, such as every hour or every few minutes.

Since data pipelines involve many steps, data engineers ensure the steps are executed in the correct sequence. Data engineers must also have visibility into the data pipeline execution, including how long it took, how much data it ingested, and whether it finished successfully. The process that involves scheduling, defining dependencies, error handling, and sending notifications to ensure efficient execution of data pipeline steps is called *data pipeline orchestration*.

In this chapter, we will learn how to schedule data pipelines so that data engineers don't have to run them manually. We will describe the Snowflake Tasks feature, used to execute commands on schedule. We will learn how to create task dependencies, allowing the entire pipeline to be scheduled at a specific time, and how to execute all steps required to transform the data in the pipeline in order, one after another. We will also learn how to send email notifications from tasks to inform the data engineer if the pipeline execution is completed successfully. Finally, we will describe how to monitor and troubleshoot pipeline execution.

To illustrate the examples in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries and delivers these baked goods to small businesses, such as grocery stores, coffee shops, restaurants, and hotels in the neighborhood. In chapter 11, we built a data pipeline that ingests data from two sources and transforms the data through the extraction, staging, data warehouse, and presentation layers.

One of the data sources is order information from the bakery's customers, stored as JSON files in cloud storage. The data from these files is copied into Snowflake tables in the extraction layer. Then the JSON data is flattened into a relational structure in the staging layer.

Another data source is the bakery's operational IT system database, which stores its core business data, including details about the business partners and baked goods. A data integration tool ingests this data into the staging layer.

The data from the staging layer is normalized in the data warehouse layer. Finally, the data is summarized and presented for reporting in the presentation layer. In chapter 12, we designed the data pipeline to ingest data incrementally to avoid processing all data every time the pipeline is executed. Figure 13.1 illustrates the data pipeline steps and the corresponding database objects in each layer.

In this chapter, we will add orchestration to the data pipeline so that all steps are executed in sequence on schedule.

NOTE All code and sample data files for this chapter are available in the accompanying GitHub repository in the Chapter_13 folder at https://mng.bz/5OMD.

The SQL code is stored in multiple files whose names start with Chapter_13_Part*, where * indicates a sequence number and additional information refers to the file's content. Please read the files sequentially to follow the exercises.

The exercises in this chapter use objects created in chapters 11 and 12, so you must perform all the exercises in chapters 11 and 12 before continuing with the exercises in this chapter.

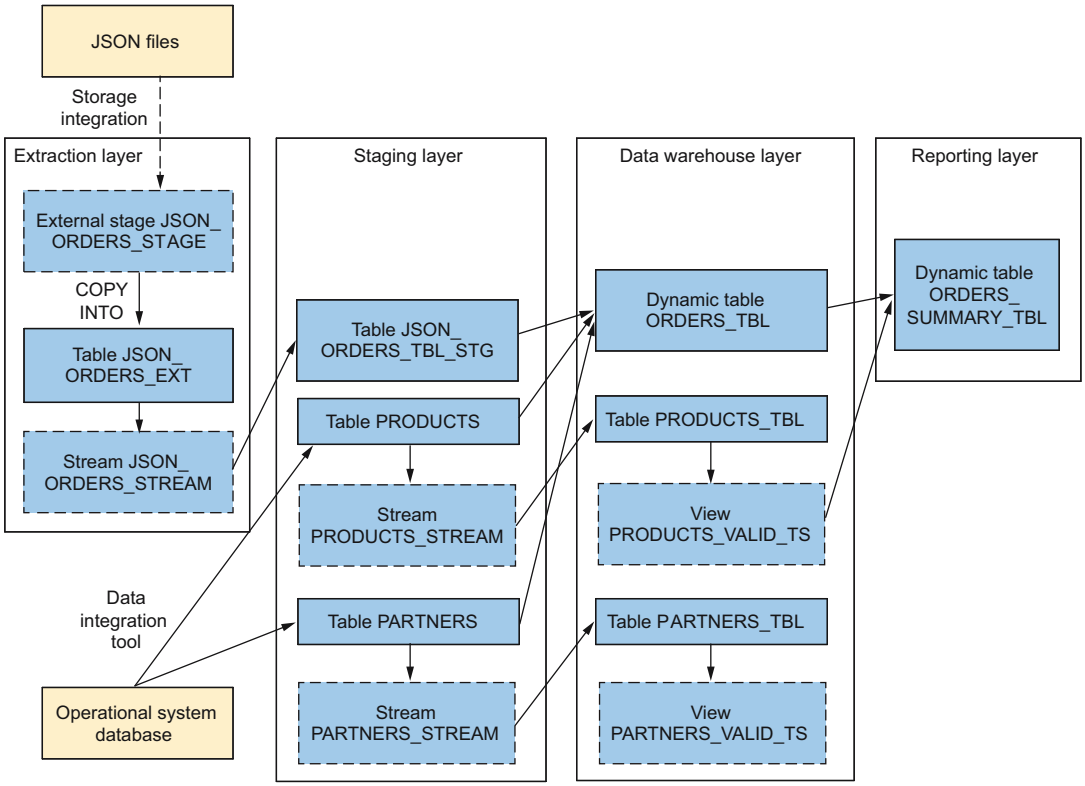


Figure 13.1 Objects in the extraction, staging, warehouse, and reporting layers used to build the bakery's data pipeline that ingests data incrementally

13.1 Orchestrating with Snowflake tasks

We introduced Snowflake tasks in chapter 2, where we created a single task executed on schedule. In this chapter, we will build more complex task graphs with dependencies to orchestrate the data pipeline.

Snowflake tasks are user-defined objects that enable the scheduling and execution of SQL commands, stored procedures, or Snowflake scripting code blocks. They can be scheduled to start executing at a specific time, repeat based on defined time intervals, or begin executing after a predecessor task is completed. Tasks can be combined with streams to support incremental processing.

Other data orchestration tools

There are different ways to orchestrate data pipelines besides using Snowflake tasks. One option is to use data integration tools that offer orchestration functionality and support the execution of Snowflake commands and calling stored procedures.

Another option is to use third-party orchestration tools, such as Airflow, Dagster, Prefect, or other open source or proprietary tools available on the market.

Snowflake tasks are usually used for orchestration when you require native functionality and prefer not to add third-party tools.

When you create a task, you must specify the compute resources it will use to execute the commands. You can specify a virtual warehouse or choose the serverless compute model, in which Snowflake manages the compute resources by automatically scaling them up or down as needed for each task.

NOTE For more information about Snowflake tasks, including recommendations on when to use serverless or user-managed tasks, refer to the Snowflake documentation at https://mng.bz/6Ya6.

13.1.1 Creating a schema to store the orchestration objects

Snowflake tasks are objects that are stored in a schema. In addition to tasks, we often have other objects, such as tables that store logging information, error handling information, or other configuration information related to the pipeline orchestration.

To store the orchestration objects, we don't want to use one of the existing schemas that represent the layers in the data pipeline, such as the extraction layer, the staging layer, the data warehouse layer, or the presentation layer. We prefer to keep the objects related to data pipeline orchestration separate from objects related to data ingestion and transformation, so we will create a new schema named `ORCHESTRATION` to store them.

We will create the new schema in the `BAKERY_DB` database. As described in chapter 10, we are using role-based access control (RBAC) in our Snowflake environment. This means that we must create new schemas with managed access together with the access roles and grant them to the functional roles.

First, we will use the `SYSADMIN` role to create the new `ORCHESTRATION` schema with managed access (refer to chapter 10 for more information if needed):

```
use role SYSADMIN;
use database BAKERY_DB;
create schema ORCHESTRATION with managed access;
```

Then we will use the `SECURITYADMIN` role (because this role has the `MANAGE GRANTS` privilege) to grant full privileges on the `ORCHESTRATION` schema to the `BAKERY_FULL` access role:

```
use role SECURITYADMIN;
grant all on schema BAKERY_DB.ORCHESTRATION to role BAKERY_FULL;
```

We don't have to grant any additional privileges to the `DATA_ENGINEER` functional role because this role has already been granted the `BAKERY_FULL` access role in chapter 10 and will inherit the new access privileges automatically.

13.1.2 Designing the orchestration tasks

To design the orchestration, let's review the data pipeline, as shown in figure 13.1, and consider which steps we must include and in what order.

First, we will review the ingestion and transformation of the order information JSON files stored in cloud storage. Table 13.1 summarizes the steps performed when ingesting these files incrementally and includes considerations for orchestration design.

Table 13.1 Steps performed when ingesting JSON files from cloud storage and transforming the data until the presentation layer

Step number	Step description	Consideration for orchestration design
1	New files arrive in the object storage location.	This happens externally by the bakery's customer, and we have no control over it in the data pipeline (we simulate this step by uploading files manually).
2	The `COPY INTO` command is executed to copy the contents of the newly arrived files into the `JSON_ORDERS_EXT` table in the extraction layer.	This step must be automated with a task.
3	A stream named `JSON_ORDERS_STREAM` keeps track of new data since the last pipeline execution.	The stream detects and tracks changes automatically, so we don't have to do anything specific in the data pipeline.
4	Insert data from the stream into the `JSON_ORDERS_TBL_STG` staging table.	This step must be automated with a task. It must execute after step 2.
5	The order data from the staging layer is propagated to the `ORDERS_TBL` dynamic table in the data warehouse layer, where the data is normalized.	The dynamic table automatically updates the latest changes according to the specified schedule. We don't have to do anything specific in the data pipeline.
6	The data from the data warehouse layer is summarized in the reporting layer in the `ORDERS_SUMMARY_TBL` dynamic table.	The dynamic table automatically updates the latest changes according to the specified schedule. We don't have to do anything specific in the data pipeline.

After examining the steps and considering what must be done when designing the pipeline orchestration, we see that we must create two tasks: one that performs the `COPY INTO` command (step 2 in table 13.1) and a second one that performs the `INSERT` command (step 4 in table 13.1). By defining a dependency, we must also ensure that the second task runs after the first one.

To allow the `DATA_ENGINEER` role to execute the tasks it creates, we must grant it the `EXECUTE TASK` privilege using the `ACCOUNTADMIN` role by executing the following commands:

```
use role ACCOUNTADMIN;
grant execute task on account to role DATA_ENGINEER;
```

Then, we can continue to use the DATA_ENGINEER role to create the tasks in the ORCHESTRATION schema in the BAKERY_DB database using the BAKERY_WH warehouse. We will execute the following commands:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema ORCHESTRATION;
```

13.1.3 Creating tasks with dependencies

The first task we will create will be to perform the COPY INTO command to copy data from the JSON_ORDERS_STAGE stage into the JSON_ORDERS_EXT table in the extraction layer. We will name the task COPY_ORDERS_TASK and schedule it to run every 10 minutes for now so that we can test it, but we will change the schedule later. We will provide the BAKERY_WH warehouse as the parameter to make it a user-managed task. The command that creates this task is

```
create task COPY_ORDERS_TASK
  warehouse = BAKERY_WH
  schedule = '10 M'
as
  copy into EXT.JSON_ORDERS_EXT
  from (
    select
      $1,
      metadata$filename,
      current_timestamp()
    from @EXT.JSON_ORDERS_STAGE
  )
  on_error = abort_statement;
```

Qualifies the table name with the schema name

Qualifies the stage name with the schema name

TIP We are creating the task in the ORCHESTRATION schema, while the stage and the target table are in the EXT schema. Therefore, we must qualify the object names by prefixing them with the schema name—for example, EXT.JSON_ORDERS_STAGE and EXT.JSON_ORDERS_EXT.

Remember that when a task is created, it is suspended and must be resumed to start executing. But we don't want to resume it yet because we must add another task that will be executed after this task is completed.

When creating tasks, it's best to test them to ensure they work correctly before allowing them to execute on schedule. It's more convenient to troubleshoot individual SQL commands while developing the task rather than trying to figure out what went wrong when a pipeline fails due to an error in a task. To test a task, we can run it manually with the EXECUTE TASK command:

```
execute task COPY_ORDERS_TASK;
```

The output from this command is a message saying "Task COPY_ORDERS_TASK is scheduled to run immediately." To view the progress and completion state of the task, we

can call the TASK_HISTORY() table function with the command shown in the following listing.

Listing 13.1 Calling the TASK_HISTORY() function

```
select *
  from table(information_schema.task_history())
  order by scheduled_time desc;
```

TIP When selecting from the TASK_HISTORY() table function, it is convenient to sort the output by the scheduled time in descending order so the most recent task execution is shown first in the output.

Supposing we called the TASK_HISTORY() table function on September 2, 2023, at 9:44 a.m., and the task was already completed, the output of this command might look like table 13.2 (a few selected columns are shown).

Table 13.2 Output of the TASK_HISTORY() table function when executing the task manually (a few selected columns are shown)

Name	State	Error message	Scheduled time	Scheduled from
COPY_ORDERS_TASK	SUCCEEDED		2023-09-02 09:44:52	EXECUTE TASK

We should see that the task is in a SUCCEEDED state, and the scheduled time is slightly later than the timestamp when the EXECUTE TASK command was issued. The output also shows that the task was scheduled using the EXECUTE TASK command.

Before the task was completed, it progressed through the SCHEDULED and EXECUTING states, but these take a very short time for our task, and you probably didn't see them because the task was already completed by the time you called the TASK_HISTORY() function.

If the task was not completed successfully—for example, if there was an error in the SQL command—you might see that the state is FAILED, and an error message is shown in the Error message column. In this case, you must investigate the cause of the error and fix the task as needed.

NOTE For more information about the parameters and output columns of the TASK_HISTORY() table function, refer to the Snowflake documentation at https://mng.bz/o0dv.

The next task we will create is the INSERT_ORDERS_STG_TASK that inserts data from the JSON_ORDERS_STREAM stream into the JSON_ORDERS_TBL_STG staging table. Again, like in the previous task, we will provide the BAKERY_WH warehouse as a parameter to make it a user-managed task. We must also remember to qualify the objects with the schema name—for example, EXT.JSON_ORDERS_STREAM and STG.JSON_ORDERS_TBL_STG—because the task is stored in a different schema.

This task has no schedule because it is executed after the previous task. Instead of adding a SCHEDULE keyword, we will specify the AFTER keyword and provide the COPY_ORDERS_TASK as the task after which the current task must run.

Another option we can specify when creating a task that selects data from a stream is the SYSTEM$STREAM_HAS_DATA keyword. As the name suggests, the task will execute only if the stream has data indicating that new or changed data has arrived. Without data in the stream, the task will not execute because there will be no data to process. The command that creates this task is

```
create task INSERT_ORDERS_STG_TASK
  warehouse = 'BAKERY_WH'
  after COPY_ORDERS_TASK
when
  system$stream_has_data('EXT.JSON_ORDERS_STREAM')
as
  insert into STG.JSON_ORDERS_TBL_STG
  select
    customer_orders:"Customer"::varchar as customer,
    customer_orders:"Order date"::date as order_date,
    CO.value:"Delivery date"::date as delivery_date,
    DO.value:"Baked good type":: varchar as baked_good_type,
    DO.value:"Quantity"::number as quantity,
    source_file_name,
    load_ts
  from EXT.JSON_ORDERS_STREAM,
  lateral flatten (input => customer_orders:"Orders") CO,
  lateral flatten (input => CO.value:"Orders by day") DO;
```

TIP When you create a task with a dependency using the AFTER keyword, you won't be able to test it manually with the EXECUTE TASK command. If you wish to test the task, you must remove the dependency, test it by executing it manually, and then alter the task to add the dependency again. The code to perform these steps is in the Chapter_13_Part2_create_tasks_orders.sql script in the Chapter_13 folder in the GitHub repository.

We now have two tasks, the COPY_ORDERS_TASK as the parent task and the INSERT_ORDERS_STG_TASK as the child task. Once both tasks are created and tested, we can resume them so they will start executing, one after the other, every 10 minutes as defined in the schedule:

```
alter task INSERT_ORDERS_STG_TASK resume;
alter task COPY_ORDERS_TASK resume;
```

Now let's see if the tasks are working correctly. We will simulate new data arriving in cloud storage by manually uploading a JSON file. You can upload the file named Orders_2023-09-07.json from the Chapter_13 folder in the GitHub repository to the cloud storage location.

WARNING Upload just the one JSON file for now, even though there are additional files in the repository. You will upload more files later to test subsequent pipeline executions.

After uploading the file, wait for the subsequent scheduled pipeline execution. Because the pipeline was scheduled to run every 10 minutes, it should start executing 10 minutes after the scheduled task was resumed.

Supposing we resumed the task on September 2, 2023, at 9:55 a.m., it should be scheduled to start executing 10 minutes later, at approximately 10:05. We can check the output of the `TASK_HISTORY()` table function by executing the command in listing 13.1. If we execute the command before 10:05, we should see that the task is scheduled to run, and the output of the command looks like table 13.3 (a few selected columns are shown).

Table 13.3 Output of the `TASK_HISTORY()` table function before the first pipeline execution

Name	State	Scheduled time	Scheduled from
COPY_ORDERS_TASK	SCHEDULED	2023-09-02 10:05:24	SCHEDULE

About 10 minutes after the task was resumed, the pipeline should start executing and finish soon after that since it has a small amount of data to process. Check the output of the `TASK_HISTORY()` table function again; it should look like table 13.4 (a few selected columns are shown).

Table 13.4 Output of the `TASK_HISTORY()` table function after the first pipeline execution

Name	State	Scheduled time	Scheduled from
COPY_ORDERS_TASK	SCHEDULED	2023-09-02 10:15:24	SCHEDULE
INSERT_ORDERS_STG_TASK	SUCCEEDED	2023-09-02 10:05:49	SCHEDULE
COPY_ORDERS_TASK	SUCCEEDED	2023-09-02 10:05:24	SCHEDULE

As we can see in the output of the `TASK_HISTORY()` table function in table 13.4, the `COPY_ORDERS_TASK` parent task was executed on schedule 10 minutes after it was resumed and succeeded. Immediately after that, the `INSERT_ORDERS_STG_TASK` ran, and this task also succeeded. The next scheduled execution of the parent task is now included in the output, indicating it will start at 10:15.

If we wait another 10 minutes and check the output of the `TASK_HISTORY()` table function again, we might see an output like table 13.5 (a few selected columns are shown).

Table 13.5 Output of the TASK_HISTORY() table function after the second pipeline execution

Name	State	Scheduled time	Scheduled from
COPY_ORDERS_TASK	SCHEDULED	2023-09-02 10:25:24	SCHEDULE
INSERT_ORDERS_STG_TASK	SKIPPED	2023-09-02 10:15:38	SCHEDULE
COPY_ORDERS_TASK	SUCCEEDED	2023-09-02 10:15:24	SCHEDULE
INSERT_ORDERS_STG_TASK	SUCCEEDED	2023-09-02 10:05:49	SCHEDULE
COPY_ORDERS_TASK	SUCCEEDED	2023-09-02 10:05:24	SCHEDULE

The state of the INSERT_ORDERS_STG_TASK after the second pipeline execution is SKIPPED because the task was defined to run only when the JSON_ORDERS_STREAM stream has data. When the stream has no data, the task is skipped but still shown in the output.

Another scenario where a task execution is skipped is if a task is still running when the next scheduled execution time of the same task occurs. This is because Snowflake ensures that only one instance of a task with a schedule is executed at a given time.

Before we continue building the data pipeline, let's take a brief detour to learn how to send email notifications from Snowflake. We will use this functionality later in the pipeline.

To prevent the scheduled tasks from needlessly executing while we prepare the next steps, let's suspend the pipeline for now by executing the following command:

```
alter task COPY_ORDERS_TASK suspend;
```

We can suspend just the parent task, but not necessarily the child task, because the child task will not execute unless the parent task executes.

13.2 Sending email notifications

Snowflake supports sending emails to recipients who verified their email addresses in the same Snowflake account from where the email originates. If you are using a free trial account, you already verified your email address when you registered, so you can test this functionality by emailing yourself.

When you want to send emails to other email addresses, they must exist as users in the same Snowflake account and verify their email addresses the first time they log in to Snowflake.

> **NOTE** More information about sending email notifications from Snowflake is available in the documentation at https://mng.bz/n0B4.

To send emails from Snowflake, you must create a notification integration. We will use the ACCOUNTADMIN role—because it has the required privileges—to create a notification integration named PIPELINE_EMAIL_INT by executing the following commands:

```
use role ACCOUNTADMIN;
create notification integration PIPELINE_EMAIL_INT
  type = email
  enabled = true;
```

Then, we will grant usage on the integration to the `DATA_ENGINEER` role, which will be sending emails:

```
grant usage on integration PIPELINE_EMAIL_INT to role DATA_ENGINEER;
```

Now we will switch to the `DATA_ENGINEER` role and send an email using the `PIPELINE_EMAIL_INT` notification integration. To send an email, we must call the `SYSTEM$SEND_EMAIL()` stored procedure and provide the following parameters:

- The name of the notification integration
- The email recipient, which can be one or more comma-separated email addresses
- The subject of the email
- The body of the email

The following is a command you can use to send an email to the email address of your Snowflake user to test the functionality:

```
call SYSTEM$SEND_EMAIL(
    'PIPELINE_EMAIL_INT',
    'firstname.lastname@youremail.com',
    'The subject of the email from Snowflake',
    'This is the body of the email.'
);
```

Substitute with the email address of your Snowflake user.

After executing this command, the output should be `TRUE`, and you should receive an email soon after that. Your email might look like the following listing (your output will vary depending on the email client you use).

Listing 13.2 Sample email notification

```
From: Snowflake Computing <no-reply@snowflake.net>
To: firstname.lastname@youremail.com
Subject: The subject of the email from Snowflake
--
This is the body of the email.
```

We will call the `SYSTEM$SEND_EMAIL` function to send email notifications from tasks when we continue to build the pipeline in the next section.

13.3 Orchestrating with task graphs

Tasks used to schedule the steps in data pipelines often have dependencies, as we saw earlier when creating the orchestration for the orders data pipeline where we created the `COPY_ORDERS_TASK` parent task and the `INSERT_ORDERS_STG_TASK` child task.

In typical data pipelines, we usually have a task that represents the beginning of the pipeline execution, called the *root task*. This task can perform actions such as writing to a log table to track when it started executing, sending email notifications, or performing other setup activities for the pipeline. The data ingestion and transformation steps then follow. The data pipeline usually has a task at the very end called the *finalizer task*. This task can perform actions such as writing to a log table that the pipeline execution is completed, recording status or error information, sending email notifications, or performing other activities to finalize the data pipeline execution.

In Snowflake, we build such pipeline orchestrations by creating *task graphs*, also known as DAGS (directed acyclic graphs). A task graph is a series of tasks composed of a root task and additional tasks, organized by their dependencies. The task flow is in a single direction, meaning that each task is a descendant of a previous task without circular dependencies. Tasks can have more than one child task and can be descendants of more than one parent task. A task runs only when all its predecessor tasks are completed successfully.

The root task is scheduled, but the dependent tasks are not because they run after the preceding tasks are completed successfully. The finalizer task always runs at the end of the pipeline, even if some of the preceding tasks fail.

13.3.1 Designing the task graph

In the data pipeline we are building, we currently have two tasks, `COPY_ORDERS_TASK` and `INSERT_ORDERS_STG_TASK`, that must run in sequence. We will now add a root task named `PIPELINE_START_TASK` that will send an email informing the recipient that the pipeline execution has started. We will add a finalizer task named `PIPELINE_END_TASK` that will also send an email informing the recipient that the pipeline execution is completed and providing some information about the processing steps.

In addition to ingesting order information from cloud storage, we must also ingest the partner and product data populated by a data integration tool from the bakery's operational IT system. Returning to figure 13.1, let's review the ingestion and transformation steps for the partner and product data. The steps performed and considerations for orchestration design are summarized in table 13.6.

Table 13.6 Steps performed when ingesting and transforming the partners and products data incrementally

Step number	Step description	Consideration for orchestration design
1	The data integration tool updates the `PARTNER` and `PRODUCT` tables in the staging layer.	This is done externally by the data integration tool, and we have no control over it in the data pipeline (we simulate this step by inserting or updating data manually).
2	Two streams named `PARTNER_STREAM` and `PRODUCT_STREAM` keep track of new data since the last pipeline execution.	The streams detect and track changes automatically. We don't have to do anything specific in the data pipeline.

Table 13.6 Steps performed when ingesting and transforming the partners and products data incrementally *(continued)*

Step number	Step description	Consideration for orchestration design
3	Inserts data from the `PARTNER_STREAM` stream into the `PARTNER_TBL` table in the warehouse layer	This step must be automated with a task.
4	Inserts data from the `PRODUCT_STREAM` stream into the `PRODUCT_TBL` table in the warehouse layer	This step must be automated with a task.

After examining the steps, we see that we must create two tasks: one that inserts data from the `PARTNER_STREAM` stream into the `PARTNER_TBL` table (step 3 in table 13.6) and a second one that inserts data from the `PRODUCT_STREAM` stream into the `PRODUCT_TBL` table (step 4 in table 13.6). The tasks we will create, named `INSERT_PARTNER_TASK` and `INSERT_PRODUCT_TASK` respectively, are independent and can run in parallel. When designing the task graph, we must ensure they run after the root task and before the finalizer task.

The complete task graph we will build is shown in figure 13.2.

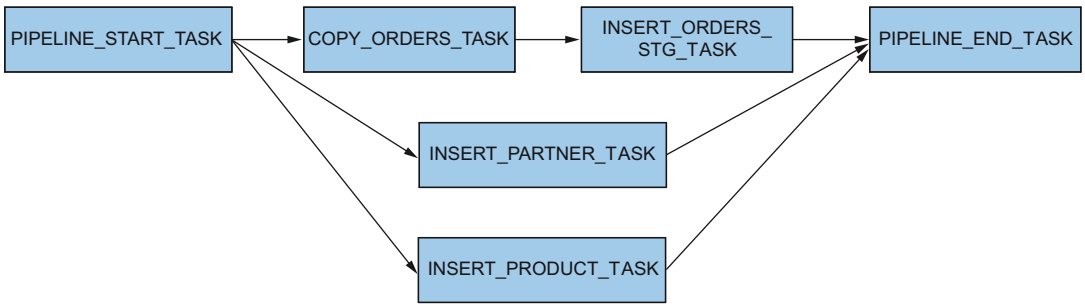


Figure 13.2 Tasks and their dependencies in the task graph. The pipeline starts with a root task named `PIPELINE_START_TASK`, executes the data ingestion and transformation steps in sequence, and ends with a finalizer task named `PIPELINE_END_TASK`.

Let's build this task graph for the data pipeline, still using the `DATA_ENGINEER` role, the `BAKERY_WH` warehouse, the `BAKERY_DB` database, and the `ORCHESTRATION` schema:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema ORCHESTRATION;
```

13.3.2 Creating the root task

The root task is the PIPELINE_START_TASK. We will schedule it to run every 10 minutes for testing for now, but we can change this schedule later. The task will send an email to the recipient informing them that the pipeline execution started along with the timestamp when it started. The code that creates this task is

```
create or replace task PIPELINE_START_TASK
  warehouse = BAKERY_WH
  schedule = '10 M'
as
  call SYSTEM$SEND_EMAIL(
    'PIPELINE_EMAIL_INT',
    'firstname.lastname@youremail.com',
    'Daily pipeline start',
    'The daily pipeline started at ' || current_timestamp || '.'
);
```

Substitute your email address.

Next, we must create the two tasks that insert data from the partners and products staging tables into the data warehouse layer. Since both tasks select data from their respective streams that detect changes in the staging data, we will add the WHEN SYSTEM$STREAM_HAS_DATA conditions. Each task must run after the PIPELINE_START_TASK root task. Therefore, we will add an AFTER keyword specifying the dependency. The code to create the INSERT_PRODUCT_TASK and the INSERT_PARTNER_TASK tasks is

```
create or replace task INSERT_PRODUCT_TASK
  warehouse = BAKERY_WH
  after PIPELINE_START_TASK
when
  system$stream_has_data('STG.PRODUCT_STREAM')
as
  insert into DWH.PRODUCT_TBL
  select product_id, product_name, category,
    min_quantity, price, valid_from
  from STG.PRODUCT_STREAM
  where METADATA$ACTION = 'INSERT';

create or replace task INSERT_PARTNER_TASK
  warehouse = BAKERY_WH
  after PIPELINE_START_TASK
when
  system$stream_has_data('STG.PARTNER_STREAM')
as
  insert into DWH.PARTNER_TBL
  select partner_id, partner_name, address, rating, valid_from
  from PARTNER_STREAM
  where METADATA$ACTION = 'INSERT';
```

13.3.3 Creating the finalizer task

The last task we need for the pipeline is the finalizer task named `PIPELINE_END_TASK`. When creating this task, we must specify that it is finalizing the `PIPELINE_START_TASK` root task by providing this value to the `FINALIZE` keyword:

```
create task PIPELINE_END_TASK
  warehouse = BAKERY_WH
  finalize = PIPELINE_START_TASK
as
  call SYSTEM$SEND_EMAIL(
    'PIPELINE_EMAIL_INT',
    'firstname.lastname@youremail.com',
    'Daily pipeline end',
    'The daily pipeline finished at ' || current_timestamp || '.'
);
```

Before we can resume the pipeline, we must make one more change: altering the `COPY_ORDERS_TASK`. This task is currently set to run on schedule. To make it part of the task graph we are building, we must remove the schedule and add the `AFTER` keyword to make it run after the root task. The task should be suspended before we can modify it. The commands that perform these changes are

```
alter task COPY_ORDERS_TASK suspend;
alter task COPY_ORDERS_TASK unset schedule;
alter task COPY_ORDERS_TASK
  add after PIPELINE_START_TASK;
```

Now we should have everything ready to resume the tasks in the pipeline and execute them. To resume all tasks, execute the following commands:

```
alter task PIPELINE_END_TASK resume;
alter task INSERT_PRODUCT_TASK resume;
alter task INSERT_PARTNER_TASK resume;
alter task INSERT_ORDERS_STG_TASK resume;
alter task COPY_ORDERS_TASK resume;
alter task PIPELINE_START_TASK resume;
```

Now that the task graph is resumed, we can wait for it to execute on schedule for 10 minutes (alternatively, we can issue the `EXECUTE TASK` command on the root task to start executing immediately). Once the pipeline starts executing, you should receive two emails: one informing you that the pipeline execution has started and another that it has been completed.

You can also call the `TASK_HISTORY()` table function using the command in listing 13.1 to see if the pipeline execution was successful. The output of this command looks like table 13.7 (a few selected columns are shown).

Table 13.7 Output of the TASK_HISTORY() table function after a scheduled execution of the task graph

Name	State	Scheduled time	Scheduled from
PIPELINE_START_TASK	SCHEDULED	2023-09-02 11:14:46	SCHEDULE
PIPELINE_END_TASK	SUCCEEDED	2023-09-02 11:05:23	SCHEDULE
INSERT_ORDERS_STG_TASK	SKIPPED	2023-09-02 11:05:13	SCHEDULE
COPY_ORDERS_TASK	SUCCEEDED	2023-09-02 11:04:56	SCHEDULE
INSERT_PARTNER_TASK	SKIPPED	2023-09-02 11:04:56	SCHEDULE
INSERT_PRODUCT_TASK	SKIPPED	2023-09-02 11:04:56	SCHEDULE
PIPELINE_START_TASK	SUCCEEDED	2023-09-02 11:04:46	SCHEDULE

As we can see in the output, all steps were executed successfully, and some were skipped because there was no data in the streams. The root task is scheduled to run again for the next scheduled execution.

13.3.4 Viewing the task graph

The data pipeline in this exercise is small and has a limited number of steps, so it executes quickly and is easy to manage. Data pipelines in the real world are much larger and have more dependencies. You can view your task graphs in the Snowsight user interface to help you understand their scope and dependencies.

To view the task graph we created earlier in the Snowsight user interface, follow these steps:

1. Navigate to the Data option in the left navigation menu.
2. Choose Databases.
3. Expand the BAKERY_DB database.
4. Expand the ORCHESTRATION schema.
5. Expand the Tasks group, where you should see all the tasks you created.
6. Choose the PIPELINE_START_TASK root task, which should open the Properties window of this task in the right pane.
7. Select the Graph tab in the right pane.

You should see a graphical representation of your task graph as in figure 13.3.

We can improve the orchestration of the data pipeline using the task graph with more informative notification messages and logging information. In the next section, we will enhance this pipeline and add more data to ingest.

In the meantime, to prevent the pipeline from executing on schedule every 10 minutes and needlessly consuming resources, as well as to allow making changes to the tasks, we will suspend it with the following command:

```
alter task PIPELINE_START_TASK suspend;
```

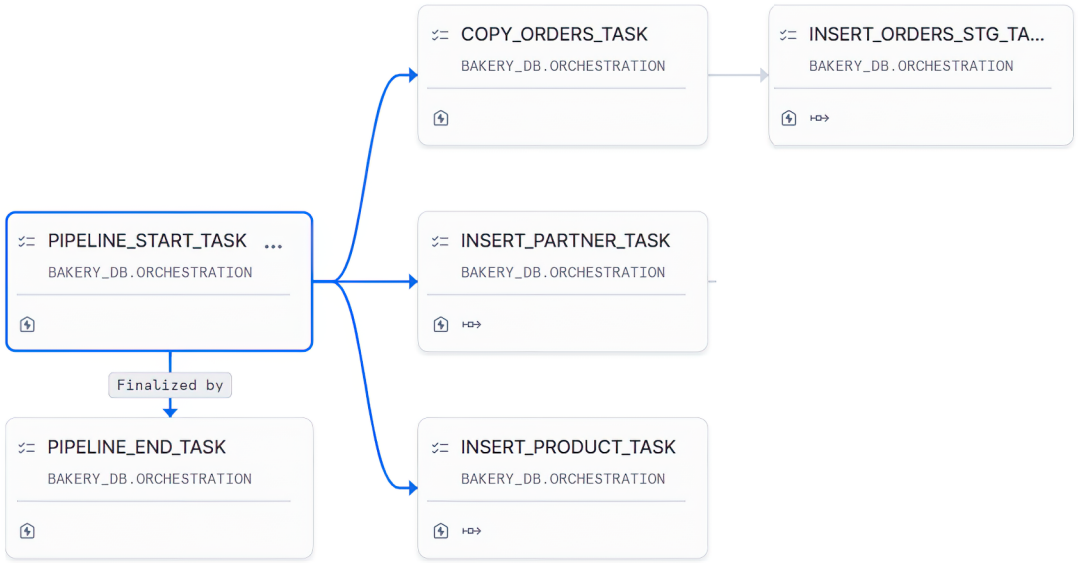


Figure 13.3 The task graph of the data pipeline originates from the `PIPELINE_START_TASK` root task in the Snowsight user interface.

Like previously, suspending only the root task is sufficient—but not any dependent tasks, because they will not run unless their predecessor completes execution.

13.4 Monitoring data pipeline execution

To allow data engineers to monitor the data pipeline execution, we will add logging to the tasks and enhance the email notification from the finalizer task to include the logging information in the body of the email.

13.4.1 Adding logging functionality to tasks

We will create a logging table named `PIPELINE_LOG` in the `ORCHESTRATION` schema to capture logging information, including the unique identifier of each pipeline execution, also called the run group ID, the root task name, the task name, the execution timestamp, and the number of processed rows:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema ORCHESTRATION;

create or replace table PIPELINE_LOG (
  run_group_id varchar,
  root_task_name varchar,
  task_name varchar,
  log_ts timestamp,
```

```
  rows_processed number
);
```

We will modify the COPY_ORDERS_TASK task to add functionality that inserts information into the logging table. To do this, we must make the following changes:

- Set a schedule for the task so we can test it (we will modify the task to unset the schedule and add a dependency on the root task later).
- Enclose the body of the task with the BEGIN and END keywords because we will add a command that inserts data into the logging table.
- Add an INSERT statement to insert data into the logging table.
- Retrieve the run group ID from the SYSTEM$TASK_RUNTIME_INFO system function by providing CURRENT_TASK_GRAPH_RUN_GROUP_ID as the parameter—this value represents a universally unique identifier (UUID) of the task graph execution.
- Retrieve the root task name from the SYSTEM$TASK_RUNTIME_INFO system function by providing CURRENT_ROOT_TASK_NAME as the parameter.
- Retrieve the current task name from the SYSTEM$TASK_RUNTIME_INFO system function by providing CURRENT_TASK_NAME as the parameter.
- Use the SQLROWCOUNT built-in variable that returns the number of affected rows in the last command.

The code that recreates the task is

```
create or replace task COPY_ORDERS_TASK
  warehouse = BAKERY_WH
  schedule = '10 M'
as
  begin
    copy into EXT.JSON_ORDERS_EXT
    from (
      select
        $1,
        metadata$filename,
        current_timestamp()
      from @EXT.JSON_ORDERS_STAGE
    )
    on_error = abort_statement;

    insert into PIPELINE_LOG
    select
      SYSTEM$TASK_RUNTIME_INFO('CURRENT_TASK_GRAPH_RUN_GROUP_ID'),
      SYSTEM$TASK_RUNTIME_INFO('CURRENT_ROOT_TASK_NAME'),
      SYSTEM$TASK_RUNTIME_INFO('CURRENT_TASK_NAME'),
      current_timestamp(),
      :SQLROWCOUNT;
  end;
```

To test, we can execute the task manually:

```
execute task COPY_ORDERS_TASK;
```

Then check the output of the `TASK_HISTORY()` function from listing 13.1 to verify that the task was executed successfully. We can also check the contents of the `PIPELINE_LOG` table to see if the logging data was inserted:

```
select * from PIPELINE_LOG;
```

The output of this command should contain one row of data, as in table 13.8 (some values are abbreviated).

Table 13.8 Contents of the `PIPELINE_LOG` table after executing the `COPY_ORDERS_TASK` manually

Run group ID	Root task name	Task name	Log timestamp	Rows processed
900b70c2...	PIPELINE_START_TASK	COPY_ORDERS_TASK	2023-09-02 12:17:24	0

When we have verified that the task is working correctly, we can alter the task to unset the schedule and add a dependency on the root task again:

```
alter task PIPELINE_START_TASK suspend;
alter task COPY_ORDERS_TASK unset schedule;
alter task COPY_ORDERS_TASK
  add after PIPELINE_START_TASK;
```

We can then add logging to the `INSERT_ORDERS_STG_TASK`, `INSERT_PRODUCT_TASK`, and `INSERT_PARTNER_TASK` tasks in the same manner. The code for recreating the tasks is available in the `Chapter_13_Part5_monitoring.sql` script in the Chapter_13 folder in the GitHub repository to execute it on your own.

13.4.2 Summarizing logging information in an email notification

We will also recreate the finalizer task by constructing a string with a return message summarizing the logging information from all tasks in the current run by reading from the `PIPELINE_LOG` table using a cursor. In the code, we will declare a variable named `RETURN_MESSAGE` to store the return message. We will define a cursor named `LOG_CUR` that reads from the `PIPELINE_LOG` table by filtering for the current run group ID, then loops through the records in the cursor and appends the task name and the number of processed rows to the return message. The return message is then appended to the body of the email sent to the recipient.

The code that recreates this task (remember to substitute the email address of your Snowflake user in the email recipient) is

```
create or replace task PIPELINE_END_TASK
  warehouse = BAKERY_WH
  finalize = PIPELINE_START_TASK
as
  declare
    return_message varchar := '';
```

Declares the return message variable

```
begin
  let log_cur cursor for
    select task_name, rows_processed
    from PIPELINE_LOG
    where run_group_id =
      SYSTEM$TASK_RUNTIME_INFO('CURRENT_TASK_GRAPH_RUN_GROUP_ID');

  for log_rec in log_cur loop
    return_message := return_message ||
      'Task: '|| log_rec.task_name ||
      ' Rows processed: ' || log_rec.rows_processed ||  '\n';
  end loop;

  call SYSTEM$SEND_EMAIL(
    'PIPELINE_EMAIL_INT',
    'firstname.lastname@youremail.com',
    'Daily pipeline end',
    'The daily pipeline finished at ' || current_timestamp || '.' ||
      '\n\n' || :return_message
  );
end;
```

Declares a cursor that reads from the PIPELINE_LOG table

Filters for the current run group ID

Loops through the records in the cursor

Concatenates the values to the return message

Includes the return message in the email body

After making all the changes to the tasks and before resuming them, we will add data in the sources so the pipeline will have something to process. Let's upload another file to the cloud storage location. This time, we will upload the file Orders_2023-09-08.json from the Chapter_13 folder in the GitHub repository to the cloud storage location.

We will also make a few changes to the partner and product tables in the staging area. Let's add a new partner in the `PARTNER` table and update the minimum quantity of a product in the `PRODUCT` table:

```
insert into STG.PARTNER values(
  113, 'Lazy Brunch', '1012 Astoria Avenue', 'A', '2023-09-01'
);
update STG.PRODUCT set min_quantity = 5 where product_id = 5;
```

With the preparation done, we can now resume all tasks:

```
alter task PIPELINE_END_TASK resume;
alter task INSERT_PRODUCT_TASK resume;
alter task INSERT_PARTNER_TASK resume;
alter task INSERT_ORDERS_STG_TASK resume;
alter task COPY_ORDERS_TASK resume;
alter task PIPELINE_START_TASK resume;
```

Then we can wait until the next scheduled pipeline execution or run the pipeline manually:

```
execute task PIPELINE_START_TASK;
```

As usual, we will check the output of the `TASK_HISTORY()` table function by executing the command from listing 13.1 and examine the output to verify that the execution was successful.

New data should also be inserted into the PIPELINE_LOG logging table. We can query this table by executing the following command and sorting the results by the log timestamp in descending order to see the latest rows at the top:

```
select * from PIPELINE_LOG order by log_ts desc;
```

The output of this command should look like table 13.9.

Table 13.9 Contents of the PIPELINE_LOG table after uploading a file to the cloud storage and updating the partners and products staging tables (some values are abbreviated)

Run group ID	Root task name	Task name	Log timestamp	Rows processed
f66163ce...	PIPELINE_START_TASK	INSERT_PARTNER_TASK	2023-09-02 12:42:31	1
f66163ce...	PIPELINE_START_TASK	INSERT_PRODUCT_TASK	2023-09-02 12:42:31	1
f66163ce...	PIPELINE_START_TASK	INSERT_ORDERS_STG_TASK	2023-09-02 12:42:31	4
f66163ce...	PIPELINE_START_TASK	COPY_ORDERS_TASK	2023-09-02 12:42:22	1

You should also have received emails notifying you about the pipeline start execution and the pipeline end execution. The email about the end execution should contain information about the executed tasks and the number of processed rows, like in the following listing.

Listing 13.3 Email notification from the pipeline execution

```
From: Snowflake Computing <no-reply@snowflake.net>
To: firstname.lastname@youremail.com
Subject: Daily pipeline end
--
The daily pipeline finished at 2023-09-02 12:42:42.

Task: BAKERY_DB.ORCHESTRATION.INSERT_PARTNER_TASK Rows processed: 1
Task: BAKERY_DB.ORCHESTRATION.INSERT_PRODUCT_TASK Rows processed: 1
Task: BAKERY_DB.ORCHESTRATION.INSERT_ORDERS_STG_TASK Rows processed: 4
Task: BAKERY_DB.ORCHESTRATION.COPY_ORDERS_TASK Rows processed: 1
```

When building data pipelines in the real world, you will probably enhance the logging and monitoring capabilities with additional steps, such as adding more columns to the logging table, like the user and role that executed the task, the virtual warehouse used (when using user-managed tasks), the total execution time, and so on.

We have been working with data pipelines that have been executed successfully until now. In the next section, we will discuss what to do when one or more tasks in a data pipeline fail.

Again, let's suspend the pipeline so it doesn't needlessly consume resources and send emails every 10 minutes:

```
alter task PIPELINE_START_TASK suspend;
```

13.5 Troubleshooting data pipeline failures

When one or more tasks in a data pipeline fail, you should see the tasks in a `FAILED` status when you examine the output of the `TASK_HISTORY()` table function. There are many reasons why tasks may fail—for example, errors in the code, errors in the data, errors when defining task dependencies, forgetting to resume a task after making changes, and so on.

When troubleshooting failures, consider these general guidelines, which can help you investigate the reason for failure and find a solution:

- Examine the error message from the failed task, which should give you an indication of where to look next.
- Ensure that the code in the task definition's body executes successfully and fix any errors. Consider adding exception handling functionality in the code, as described in chapter 4.
- Execute the task manually using the `EXECUTE_TASK` command, and fix any errors.

If a task that should execute doesn't execute, there could be various explanations—for example:

- The task is suspended. Check the task status in the Snowsight user interface or by executing the `SHOW TASKS` command, and resume the task if it was suspended. Remember that when you create or recreate a task, it becomes suspended, so you must resume it before executing it in the pipeline.
- The task dependency hasn't been set up correctly. Check the task graph in the Snowsight user interface to view the dependencies and make necessary changes.
- The predecessor task didn't run. Check the `TASK_HISTORY()` output and look for the execution status of the preceding task. Then troubleshoot the preceding task.
- The task schedule is set up so that it hasn't been scheduled to run yet. Check the `TASK_HISTORY()` output and look for the next scheduled execution.
- The task was skipped because the pipeline was still running from the previously scheduled execution.
- The task was skipped because there was no data to process—for example, when it reads from a stream and was defined with the `SYSTEM$STREAM_HAS_DATA` condition. Check if there was any data available to process.

Configuring error notifications

When using Snowflake tasks in a production environment, you can send error notifications from failed task executions to a cloud messaging service. You can configure a notification integration with your chosen cloud provider and specify it when creating a task. More information about configuring error notifications is available in the Snowflake documentation at https://mng.bz/vJOr.

Summary

- Data pipeline orchestration is the process that involves scheduling, defining dependencies, error handling, and sending notifications to ensure the efficient execution of data pipeline steps.
- Snowflake tasks are user-defined objects that enable the scheduling and execution of SQL commands, stored procedures, or Snowflake scripting code blocks. They can be scheduled to start executing at a specific time, repeat based on defined time intervals, or begin executing after a predecessor task is completed.
- When a task is created, it is suspended and must be resumed to start executing.
- When creating tasks, it's best to test them to ensure they work correctly before allowing them to execute on schedule. It's more convenient to troubleshoot individual SQL commands while developing the task than to fix them when a pipeline fails. To test a task, we can run it manually with the `EXECUTE TASK` command.
- To view a task's progress and completion state, we can call the `TASK_HISTORY()` table function and examine the output.
- Tasks can be combined with streams to support incremental processing. When creating a task, we can specify the `SYSTEM$STREAM_HAS_DATA` keyword. The task will execute only if the stream has data indicating that new or changed data arrived.
- Snowflake ensures that only one instance of a task with a schedule is executed at a given time. If a task is still running when the next scheduled execution time of the same task occurs, the task is skipped.
- Snowflake supports sending emails to recipients who verified their email addresses in the same Snowflake account from where the email originates. To send emails from Snowflake, you must create a notification integration.
- A task graph, also called a DAG, is a series of tasks composed of a root task and additional tasks organized by their dependencies. The task flow is in a single direction, meaning that each task is a descendant of a previous task without circular dependencies. Tasks can have more than one child task and can be descendants of more than one parent task.
- The root task in a task graph is scheduled, but the dependent tasks are not because they run after the preceding tasks are completed successfully. The

finalizer task always runs at the end of the pipeline, even if some of the preceding tasks fail.

- You can view your task graphs in the Snowsight user interface to help you understand their scope and dependencies.
- When one or more tasks in a data pipeline fail, you should see the tasks in a `FAILED` status when you examine the output of the `TASK_HISTORY()` table function.

14 Testing for data integrity and completeness

This chapter covers

- Data testing methods
- Incorporating data testing into data pipelines
- Applying the Snowflake data metric functions
- Alerting users when data metrics exceed thresholds
- Detecting data volume anomalies

Trustworthy data is the cornerstone of successful business intelligence and analytics solutions. To ensure that business users have high-quality data they can consume confidently, data engineers include data testing functionality when building data pipelines. They perform data quality tests that check for data integrity and completeness and take action when test results don't comply with the data quality standards.

In this chapter, we will learn how to incorporate data testing into data pipelines. First, we will describe and compare various data testing methods. We will add data testing steps to the data pipeline. We will introduce the Snowflake data metric functions to monitor data quality. We will describe how to add user-defined data metric functions. Then we will design alerts that notify data engineers and business users

when data quality metrics exceed the defined thresholds. Finally, we will learn how to use the Snowflake ML anomaly detection functionality to monitor data ingestion volumes and flag when data volumes deviate from the expected values.

To illustrate the examples in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries and delivers these baked goods to small businesses, such as grocery stores, coffee shops, restaurants, and hotels in the neighborhood. Previously, in chapter 8, the bakery looked to expand its production and sell the baked goods in retail stores nearby. The bakery now has a new customer, a supermarket chain that places large daily orders for baked goods.

In chapter 11, we built a data pipeline that ingests data from two sources: the bakery's operational IT system and order information from the bakery's customers, stored as JSON files in cloud storage. The pipeline transforms the data through the extraction, staging, data warehouse, and presentation layers. In chapter 12, we modified the data pipeline to ingest data incrementally to avoid processing all data every time the pipeline executes. In chapter 13, we added orchestration to the data pipeline so that all steps execute in sequence on schedule. In this chapter, we will enhance the pipeline with data testing functionality that checks for data integrity and completeness.

> **NOTE** All code and sample data files for this chapter are available in the accompanying GitHub repository in the Chapter_14 folder at https://mng.bz/4prv. The SQL code is stored in multiple files whose names start with Chapter_14_Part*, where * indicates a sequence number and additional information refers to the file's content. Please read the files sequentially to follow the exercises. The exercises in this chapter use objects created in chapters 11, 12, and 13, so you must perform all the exercises in chapters 11, 12, and 13 before continuing with the exercises in this chapter.

14.1 Data testing methods

Data testing involves adding steps to data pipelines that validate data quality and identify data quality violations. These steps can be executed *synchronously* by including them in the pipeline and executing them in sequence. Data testing can also be implemented *asynchronously* by separating the data ingestion functionality from the data testing functionality. Each approach has its merits, as described in the following sections.

14.1.1 Performing data testing as steps in the pipeline

One way to implement data testing is to add data testing steps in the pipeline. For example, we can add a testing step after each data ingestion or transformation step that tests for various data quality metrics, such as

- *Uniqueness*—Does the data comply with primary key constraints and have no duplicates?
- *Consistency*—Does the data comply with foreign key constraints?

- *Data type alignment*—Is the data the correct data type?
- *Data distribution*—Are the data values within the expected range?
- *Missing values*—Does the data contain missing values (null) that should be filled?
- *Completeness*—Did the data arrive in full, or is anything missing?
- *Recency*—When was the last time the data was ingested, or did it fail to arrive?

Based on the outcome of the data testing steps, the data engineer can implement various actions in the pipeline, such as stopping the pipeline execution, notifying designated recipients of the test results, or logging the test results for later inspection.

STOPPING PIPELINE EXECUTION

Stopping pipeline execution when data tests fail is usually not desired in order to avoid unnecessary delays in data ingestion. Pipeline execution should only stop when significant failures occur, such as when no data arrives from the source or when the data from the source is corrupted and can't be inserted into target tables—for example, when all date formats don't match the correct format.

NOTIFYING RECIPIENTS ABOUT THE TEST RESULTS

Ideally, data should be tested at the source and arrive at the data warehouse in a consistent state and in full. However, this is not always practical or possible. Many legacy source systems contain historical data that can't be changed, even if it contains errors or inconsistencies. In such cases, the data pipeline must ingest the data because it reflects what is stored in the source system, regardless of its quality.

Rather than stopping the data pipeline execution when data tests fail, the pipeline can send notifications to designated recipients about the state of the data. This allows the recipients to take actions, such as correcting the data in the source systems and ingesting the updated data in a subsequent pipeline execution. Another course of action is to let the downstream consumers handle the data that failed data quality testing in their solutions by excluding it or transforming it according to their needs.

FLAGGING DATA THAT FAILED DATA TESTS

The data pipeline can only flag data that fails data tests, without sending notifications, giving downstream consumers visibility into the data quality and allowing them to choose how to use it.

AVOIDING DELAYS IN PIPELINE EXECUTION DUE TO TESTING

In the early days of data warehousing, the *ETL (extract-transform-load)* approach was the most common. With this approach, the data is transformed, tested, and cleaned during the ingestion step. This often resulted in delayed data warehouse delivery because data engineers had to apply business rules to test data during ingestion, which increased the complexity of the pipeline steps. Data pipelines often rejected data that violated data quality rules, resulting in incomplete data in the data warehouse until the data was corrected at the source, causing even more delays.

The *ELT (extract-load-transform)* approach has gained traction recently. Because it ingests the data first, regardless of its quality, it enables fast and efficient pipeline

execution without unnecessary delays due to data quality violations. Depending on business requirements, the ingested data can be tested later in the pipeline or not tested at all.

14.1.2 Performing data testing independently of the pipeline

Another option to avoid delays during pipeline execution is to implement the data testing functionality independently of the data ingestion functionality. For example, we can perform data quality testing after each execution of the data ingestion pipeline.

When the ingestion frequency is high or when data is ingested continuously with Snowpipe, it may not be practical or time-efficient to perform data testing after each pipeline execution. In such cases, data engineers can implement the data testing process to execute on schedule as required by the downstream consumers—for example, every hour, every evening, or once per week.

Snowflake provides data metric functions that perform data testing on schedule, independently of any data pipelines. You can also add your own custom data metric functions. We will describe data metric functions in more detail later in this chapter.

14.2 Incorporating data testing steps in the pipeline

Let's first illustrate how we might add data testing steps to the data pipeline we built in previous chapters. In chapter 13, we created a task graph starting with a root task named `PIPELINE_START_TASK`, followed by the data ingestion and transformation steps, and ending with a finalizer task named `PIPELINE_END_TASK`.

After data ingestion, we want to test for data quality in the `PARTNER_TBL` and `PRODUCT_TBL` tables. To do that, we will add a `PARTNER_DQ_TASK` task after the `INSERT_PARTNER_TASK` task and a `PRODUCT_DQ_TASK` task after the `INSERT_PRODUCT_TASK` task, as shown in figure 14.1.

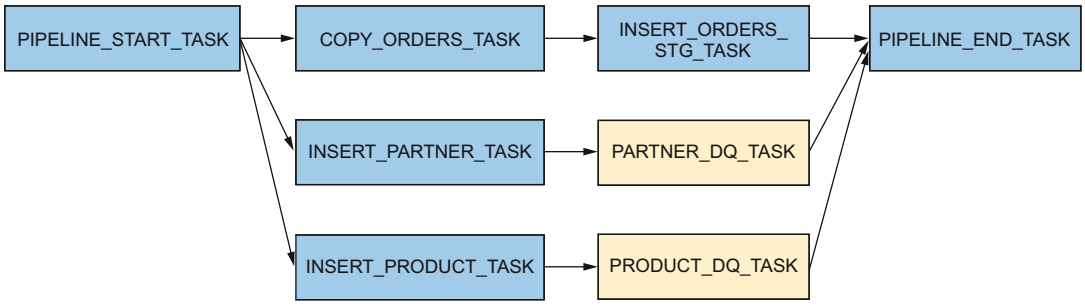


Figure 14.1 Tasks and their dependencies in the task graph from chapter 13. The pipeline starts with a root task named `PIPELINE_START_TASK`, executes the data ingestion and transformation steps in sequence, and ends with a finalizer task named `PIPELINE_END_TASK`. In this chapter, we will add the `PARTNER_DQ_TASK` and `PRODUCT_DQ_TASK` tasks that check data quality after ingesting data into the `PARTNER_TBL` and `PRODUCT_TBL` tables.

To begin, let's ingest some data that violates data quality rules. Supposing the bakery's source system, which is the source of the partners and products data, provides some messy data. As in previous chapters, we assume the bakery has a data integration tool that gets data from the operational IT system and stores it in the staging layer. In this exercise, we will simulate this process by manually inserting data into the staging tables.

We will add a row to the PARTNER table in the staging layer that has a missing rating (null value in the RATING column) and a row to the PRODUCT table in the staging layer that has an invalid product category (the value "Cake" in the CATEGORY column, which should contain only "Bread" or "Pastry"). We will use the DATA_ENGINEER role, the BAKERY_WH virtual warehouse, the BAKERY_DB database, and the STG schema to execute the following commands:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema STG;

insert into STG.PARTNER values
(114, 'Country Market', '12 Meadow Lane', null, '2023-10-10');

insert into STG.PRODUCT values
(14, 'Banana Muffin', 'Cake', 12, 3.20, '2023-10-10');
```

The data pipeline we created in the previous chapter should be suspended because that's how we left it to prevent it from executing every 10 minutes and needlessly consuming resources. We now want to execute the pipeline so that it ingests the data we just inserted in the staging tables. To do that, we could alter the pipeline to a resumed state and wait 10 minutes until it executes per schedule, but to avoid having to wait, we can execute it manually so it starts immediately:

```
execute task ORCHESTRATION.PIPELINE_START_TASK;
```

It should finish soon after it starts executing because it has very little data to process. You can check the status of the pipeline execution with the TASK_HISTORY() table function:

```
select *
from table(information_schema.task_history())
order by scheduled_time desc;
```

Since the pipeline includes notifications, you should also have received two emails—one informing you that the pipeline execution started and a second one informing you that the data pipeline execution completed—along with information that it processed one row in the INSERT_PARTNER_TASK task and one row in the INSERT_PRODUCT_TASK task.

Now that we have data that violates data quality rules in the data warehouse, let's illustrate how we could test for data quality as steps in the data pipeline.

Let's create a new schema named DQ where we will store the data quality testing functionality, following the role-based access control (RBAC) concept as described in chapter 10:

```
use role SYSADMIN;
use database BAKERY_DB;
create schema DQ with managed access;
use role SECURITYADMIN;
grant all on schema BAKERY_DB.DQ to role BAKERY_FULL;
```

We will create a table named DQ_LOG that will store the results of the data tests. The table contains columns similar to those in the PIPELINE_LOG table in the ORCHESTRATION schema we created in chapter 13 to store the task log information. The columns include the run group ID, the root task name, the task name, the log timestamp like in the PIPELINE_LOG table, and DQ_LOG specific columns indicating the database name, the schema name, and the table name of the table on which the data quality test was performed, the description of the data quality rule that was applied, the number of rows that violate the rule, and a variant object containing the offending row identifiers. The code that creates this table in the DQ schema using the DATA_ENGINEER role is

```
use role DATA_ENGINEER;
use schema DQ;

create or replace table DQ_LOG (
  run_group_id varchar, --CURRENT_TASK_GRAPH_RUN_GROUP_ID
  root_task_name varchar, --CURRENT_ROOT_TASK_NAME
  task_name varchar, --CURRENT_TASK_NAME
  log_ts timestamp,
  database_name varchar,
  table_name varchar,
  schema_name varchar,
  dq_rule_name varchar,
  error_cnt number,
  error_info variant
);
```

We will go to the ORCHESTRATION schema to work on the data quality tasks we will add to the pipeline:

```
use schema ORCHESTRATION;
```

14.2.1 Constructing the partner data quality task

Before we create the data quality tasks, let's construct the queries that perform data testing, starting with the PARTNER_TBL table. We want to identify rows that contain a null value in the RATING column by executing a query:

```
select * from DWH.PARTNER_TBL where rating is null;
```

The output of this command should return the one row we inserted earlier, as shown in table 14.1.

Table 14.1 Output of the command that selects partners whose rating contains a null value

Partner ID	Partner name	Address	Rating	Valid from
114	Country Market	12 Meadow Lane	null	2023-10-10

We want to format this result in a structure that can be stored in the `DQ_LOG` table, considering that in the future, more than one row could violate the data quality rule.

One way to format the result is to select only the `PARTNER_ID` unique identifier column, which provides sufficient information for the business user who will investigate the data quality violation after receiving the notification from the data testing task. In case there is more than one such row, we can combine all `PARTNER_IDs` into an array using the `ARRAY_AGG()` function by executing the command in the following listing.

Listing 14.1 Using the `ARRAY_AGG()` function to select all unique identifiers

```
select array_agg(PARTNER_ID) from DWH.PARTNER_TBL where rating is null;
```

This command returns an array of `PARTNER_IDs`, represented in our example as `[114]`.

We can now use the query from listing 14.1. to construct the `PRODUCT_DQ_TASK` task. We will schedule the task initially with a schedule so that we can test it by executing it manually. Later, once we are sure the task works as expected, we will remove the schedule and make it dependent on the preceding `INSERT_PARTNER_TASK` task. The command that creates the task is

```
create or replace task PARTNER_DQ_TASK
  warehouse = BAKERY_WH
  schedule = '10 M'
as
  declare
    error_info variant;
    error_cnt integer;
  begin
    select array_agg(PARTNER_ID) into error_info
    from DWH.PARTNER_TBL
    where rating is null;

    error_cnt := array_size(error_info);

    if (error_cnt > 0) then
      insert into DQ.DQ_LOG
      select
        SYSTEM$TASK_RUNTIME_INFO('CURRENT_TASK_GRAPH_RUN_GROUP_ID'),
        SYSTEM$TASK_RUNTIME_INFO('CURRENT_ROOT_TASK_NAME'),
        SYSTEM$TASK_RUNTIME_INFO('CURRENT_TASK_NAME'),
        current_timestamp(),
```

The query from listing 14.1

Derives the number of errors from the array size

Inserts into the DQ_LOG table if errors were found

```
        'BAKERY_DB',
        'DWH',
        'PARTNER_TBL',
        'Null values in the RATING column',
        :error_cnt,
        :error_info;
    end if;
  end;
```

After creating the task, we can execute it manually to test:

```
execute task PARTNER_DQ_TASK;
```

When the task execution completes, which should take very little time due to there being little data to process, we can check the DQ_LOG table:

```
select * from DQ.DQ_LOG;
```

We should see one row in this table, indicating that one row in the PARTNER_TBL table contains null values in the RATING column. The values in the DQ_LOG table should be as shown in table 14.2 (not all columns are shown).

Table 14.2 Data in the DQ_LOG table after executing the PARTNER_DQ_TASK task

Task name	Log timestamp	Table name	DQ rule name	Error count	Error info
PARTNER_DQ_TASK	2023-09-03 16:39:04	PARTNER_TBL	Null values in the RATING column	1	[114]

Finally, we will unset the schedule from the PARTNER_DQ_TASK task we just created and make it dependent on the INSERT_PARTNER_TASK:

```
alter task PARTNER_DQ_TASK unset schedule;
alter task PARTNER_DQ_TASK
  add after INSERT_PARTNER_TASK;
```

We must also remember to resume the task so it will run in the pipeline:

```
alter task PARTNER_DQ_TASK resume;
```

TIP In the PARTNER_DQ_TASK task, we insert data into the DQ_LOG table only when rows that violate the data quality rules exist. Depending on your user requirements, you might insert rows into the DQ_LOG table even when no records violate the data quality rules by providing a zero value in the ERROR_CNT column. This is to reassure the business users that a data testing step was performed, but no errors were found.

14.2.2 Constructing the product data quality task

Like the PARTNER_DQ_TASK task, we can construct the PRODUCT_DQ_TASK task, but instead of checking for null values, this task will check for allowed values in the CATEGORY column, which are "Bread" and "Pastry." The following listing shows the query that tests for this condition and returns rows that violate it.

Listing 14.2 Query that returns rows that are not in the allowed values list

```
select *
from DWH.PRODUCT_TBL
where category not in ('Bread', 'Pastry');
```

The output of this command should return the row we inserted into the PRODUCT staging table earlier, as shown in table 14.3.

Table 14.3 Output of the command that selects products whose category contains an invalid value in the CATEGORY column

Product ID	Product name	Category	Min quantity	Price	Valid from
14	Banana muffin	Cake	12	3.20	2023-10-10

The steps to create the PRODUCT_DQ_TASK task are similar to those for the PARTNER_DQ_TASK task, so we will not describe them again. You can create the PRODUCT_DQ_TASK task by executing the code in the Chapter_14_Part2_data_quality_task_PRODUCT.sql file in the Chapter_14 folder in the GitHub repository.

> **NOTE** In the query in listing 14.2, we are hard coding the allowed values "Bread" and "Pastry." In real-world scenarios, we usually have reference tables that contain valid values. In such cases, the query selects valid values from the reference table instead of hard coding them.

Once both data testing tasks are created, you can confirm that they appear correctly in the task graph by looking at the Snowsight user interface described in chapter 13. You should see a task graph like in figure 14.2.

14.2.3 Executing the pipeline with the data testing tasks

We can now execute the entire pipeline by resuming the PIPELINE_START_TASK or executing it manually. However, before we do that, note that the PARTNER_DQ_TASK and PRODUCT_DQ_TASK data testing steps depend on a preceding task that executes only when the corresponding stream has data. Therefore, if we execute the pipeline without making any changes in the staging tables, the INSERT_PARTNER_TASK and

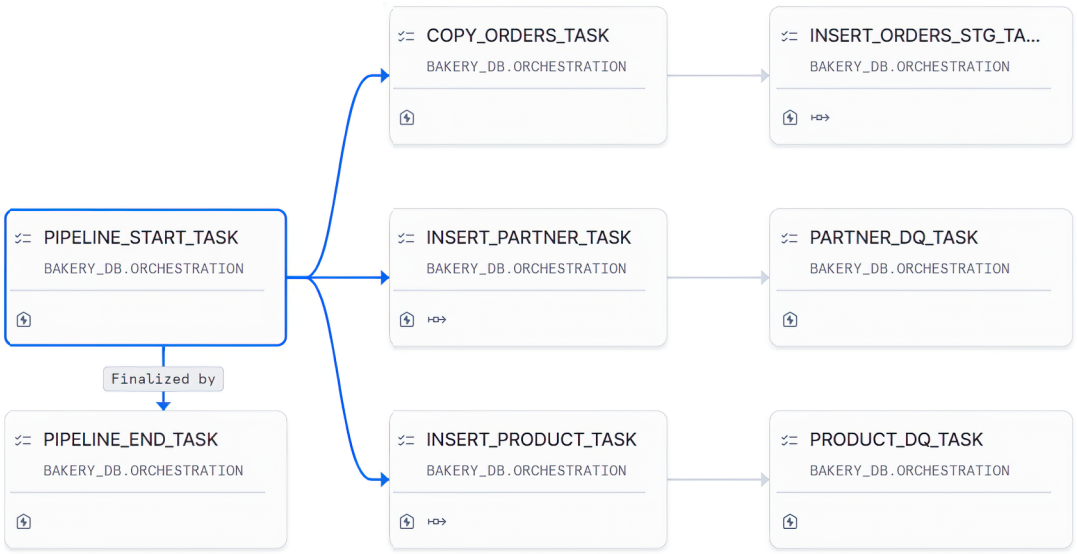


Figure 14.2 Task graph of the pipeline after adding the data testing tasks

`INSERT_PRODUCT_TASK` tasks will not execute because there is no data in the stream. Consequently, the dependent data testing tasks will not execute.

If we want to see the data testing steps in action, we must make changes in the staging tables, which we can do by simply updating the validity date of the records we inserted earlier:

```
update STG.PARTNER set valid_from = '2023-10-11' where partner_id = 114;
update STG.PRODUCT set valid_from = '2023-10-11' where product_id = 14;
```

Then we execute the pipeline manually:

```
execute task ORCHESTRATION.PIPELINE_START_TASK;
```

As previously mentioned, we can check the output of the `TASK_HISTORY()` table function to verify which pipeline steps were executed and their status. You should also receive two emails: one when the pipeline started and one when the pipeline finished. Finally, we can check the `DQ_LOG` table to see if the data testing steps detected any data quality violations:

```
select * from DQ.DQ_LOG order by log_ts desc;
```

The output from the previous command should return data, as shown in table 14.4.

Table 14.4 Data in the DQ_LOG table after executing the pipeline for the latest execution (not all columns are shown)

Task name	Log timestamp	Table name	DQ rule name	Error count	Error info
PRODUCT_DQ_TASK	2023-09-03 17:04:17	PRODUCT_TBL	Invalid values in the CATEGORY column	1	[14]
PARTNER_DQ_TASK	2023-09-03 17:04:16	PARTNER_TBL	Null values in the RATING column	1	[114]

The two data testing tasks we created exemplify adding data testing to the pipeline. To ensure thorough data testing, the pipeline should include more tests on multiple columns in the tables. Some of the tests—especially those performed on the same table—can be bundled together in a single task rather than creating an individual task for each data test on each table.

Although manually writing data testing functionality can be flexible, as demonstrated in this section, it can also involve substantial work and repetition. Data testing solutions, such as the Snowflake native data quality monitoring functions described in the next section, can help with repetitive work and simplify maintenance.

14.3 Applying the Snowflake data metric functions

Snowflake provides built-in system data metric functions in the SNOWFLAKE.CORE schema that measure standard data quality metrics. You can also define custom data metric functions according to your specific requirements, which you store in a designated schema.

Data metric functions are assigned to tables or views together with a schedule for how frequently they are evaluated. The results from the data metric function execution are stored in the event table in the Snowflake account. Since the data metric functions execute according to their defined schedule, we don't include them as steps in data pipelines.

> **NOTE** For more information about the Snowflake data quality functionality and data metric functions, refer to the Snowflake documentation at https://mng.bz/QV9Q. The Snowflake native data quality and data metric functions are available in the Snowflake Enterprise edition or higher and are currently not supported for Snowflake trial accounts. To use this feature, you must have access to a paid Snowflake account.

Let's add some data metric functions to the PARTNER_TBL and PRODUCT_TBL tables. We will use the DATA_ENGINEER role to work with the data metric functions, but first, we must grant it the following privileges:

- The database role SNOWFLAKE.DATA_METRIC_USER to access system-defined data metric functions in the SNOWFLAKE.CORE schema

- The EXECUTE DATA METRIC FUNCTION privilege to execute data metric functions
- The SNOWFLAKE.DATA_QUALITY_MONITORING_VIEWER application role to view the output of data metric functions

We will execute the following commands to grant these privileges:

```
use role ACCOUNTADMIN;
grant database role SNOWFLAKE.DATA_METRIC_USER to role DATA_ENGINEER;
grant EXECUTE DATA METRIC FUNCTION on account to role DATA_ENGINEER;
grant application role SNOWFLAKE.DATA_QUALITY_MONITORING_VIEWER
  to role DATA_ENGINEER;
```

14.3.1 System-defined data metric functions

We will start by applying a Snowflake system-defined data metric function to the PARTNER_TBL table. Snowflake provides the following system data metric functions:

- FRESHNESS—Returns the freshness of the data measured as the number of seconds between the last time the data was updated in the table and the data metric function scheduled time
- DATA_METRIC_SCHEDULED_TIME—Returns the timestamp of when the data metric function is scheduled
- NULL_COUNT—Returns the number of null values in a column
- DUPLICATE_COUNT—Returns the number of duplicate values in a column, including null values
- UNIQUE_COUNT—Returns the number of unique non-null values in the column or combination of columns
- ROW_COUNT—Returns the number of rows in a table

In this example, we will use the NULL_COUNT function on the RATING column in the PARTNER_TBL table to return the number of rows where this column contains a null value.

Before adding a data metric function to a column on a table, we must define the schedule based on which the data metric functions evaluate. We can schedule data metric functions on a table in different ways:

- By specifying a time interval in minutes
- By specifying a CRON expression to define the schedule
- By defining a trigger event that executes the data metric functions when the data in the table changes due to data manipulation language (DML) operations

In our example, we will initially schedule the data metric function to execute every 5 minutes so we can test it, but we will change the schedule later. Using the DATA_ENGINEER role, we can add the schedule to the DWH.PARTNER_TBL table by executing the following command:

```
use role DATA_ENGINEER;
alter table DWH.PARTNER_TBL set DATA_METRIC_SCHEDULE = '5 MINUTE';
```

Next, we will add the SNOWFLAKE.CORE.NULL_COUNT data metric function to the RATING column:

```
alter table DWH.PARTNER_TBL
  add data metric function SNOWFLAKE.CORE.NULL_COUNT
  on (rating);
```

After applying the schedule to the table and at least one data metric function to a column in the table, Snowflake takes care of the rest. After 5 minutes, we can examine the DATA_QUALITY_MONITORING_RESULTS table in the SNOWFLAKE database and LOCAL schema by executing the following command:

```
select measurement_time, table_name, metric_name, argument_names, value
from SNOWFLAKE.LOCAL.DATA_QUALITY_MONITORING_RESULTS
order by measurement_time desc;
```

It's often convenient to sort the results in descending order of the measurement time, with the latest results at the top. The output of this command is shown in table 14.5.

Table 14.5 Output when selecting from the DATA_QUALITY_MONITORING_RESULTS table after the data metric function was executed

Measurement time	Table name	Metric name	Argument names	Value
2023-09-03 18:23:16	PARTNER_TBL	NULL_COUNT	["RATING"]	1

As we can see in the output, the data metric function returns only the number of detected values in the VALUE column. It doesn't provide any other information about the offending rows. The business users must investigate on their own to identify which rows contain null values.

14.3.2 User-defined data metric functions

The Snowflake system-defined data metric functions are designed as generic functions that can be widely used for many data quality tests. However, they don't cater to our specific needs. We can create user-defined data metric functions to perform data testing according to our requirements.

A user-defined data metric function follows the same syntax as a standard UDF, except we provide the parameters as a table name and column names on which the function operates.

Let's create a user-defined data metric function that counts the number of values in a column that are not in the allowed values list. In our example, we want to check if any rows in the PRODUCT_TBL table contain values other than "Bread" or "Pastry" in the CATEGORY column.

We will create a data metric function named INVALID_CATEGORY with a table named T and a column named CAT as parameters. These parameters serve as placeholders to define the body of the function. We will later apply the function to a column in an

existing table, which will be executed on that table and column. The code that creates the user-defined data metric function is

```
create data metric function DQ.INVALID_CATEGORY(
  T table(CAT varchar))
  returns integer
as
$$
  select count(*)
  from T
  where CAT not in ('Bread', 'Pastry')
$$;
```

Before we can add the custom data metric function to the PRODUCT_TBL table, we must set the schedule on the table, again setting it to every 5 minutes so we can test it:

```
alter table DWH.PRODUCT_TBL set DATA_METRIC_SCHEDULE = '5 MINUTE';
```

Now we can add the INVALID_CATEGORY custom data metric function to the CATEGORY column on the PRODUCT_TBL table:

```
alter table DWH.PRODUCT_TBL
  add data metric function DQ.INVALID_CATEGORY
  on (category);
```

After about 5 minutes, we can check the output in the DATA_QUALITY_MONITORING_RESULTS table like previously:

```
select measurement_time, table_name, metric_name, argument_names, value
from SNOWFLAKE.LOCAL.DATA_QUALITY_MONITORING_RESULTS
order by measurement_time desc;
```

The output of this command is shown in table 14.6.

Table 14.6 Output when selecting from the DATA_QUALITY_MONITORING_RESULTS table after the custom data metric function was executed

Measurement time	Table name	Metric name	Argument names	Value
2023-09-03 18:23:16	PRODUCT_TBL	INVALID_CATEGORY	["CATEGORY"]	1

The output shows that the data metric function returned the number of detected values in the VALUE column, which in our case is 1.

Typically, we don't want the data metric functions to execute every 5 minutes. We can schedule them to execute less frequently—for example, every 24 hours. Alternatively, we can schedule them to run after data in the underlying tables changes due to DML operations.

To change the data metric schedule on a table, we must first unset the current schedule and then set the new schedule. To set the schedule to run after DML operations on the table, we can execute the following commands:

```
alter table DWH.PARTNER_TBL unset DATA_METRIC_SCHEDULE;
alter table DWH.PRODUCT_TBL unset DATA_METRIC_SCHEDULE;

alter table DWH.PARTNER_TBL
  set DATA_METRIC_SCHEDULE = 'TRIGGER_ON_CHANGES';
alter table DWH.PRODUCT_TBL
  set DATA_METRIC_SCHEDULE = 'TRIGGER_ON_CHANGES';
```

14.3.3 Viewing data metric function details

After making changes to data metric functions and schedules and applying them to tables, it's easy to lose track of your work. Snowflake provides visibility into these functions and their properties in parameters and table functions.

To see the metric schedule on a table, we can use the `SHOW PARAMETERS` command with the table name as the parameter:

```
show parameters like 'DATA_METRIC_SCHEDULE' in table DWH.PRODUCT_TBL;
```

The output of this command is shown in table 14.7, where the metric schedule on the `PRODUCT_TBL` table is `TRIGGER_ON_CHANGES`.

Table 14.7 Output when showing the `DATA_METRIC_SCHEDULE` parameter on the `PRODUCT_TBL` table

Key	Value	Level	Description
DATA_METRIC_SCHEDULE	TRIGGER_ON_CHANGES	Table	Specifies the schedule on which data metric functions associated with the table must be executed for evaluation

To see the data metric functions associated with a table, we can call the `DATA_METRIC_FUNCTION_REFERENCES` table function with the entity name (in our example, the `PRODUCT_TBL` table) and domain (table or view) as the parameters:

```
select metric_name, ref_entity_name, ref_entity_domain, ref_arguments,
   schedule
from table(
  INFORMATION_SCHEMA.DATA_METRIC_FUNCTION_REFERENCES(
    ref_entity_name => 'BAKERY_DB.DWH.PRODUCT_TBL',
    ref_entity_domain => 'table'
  )
);
```

The entity domain can be a table or a view because data metric functions can be defined on views as well. The output of this command is shown in table 14.8, where we see that the `INVALID_CATEGORY` metric function is associated with the `PRODUCT_TBL` table.

Table 14.8 Output from the DATA_METRIC_FUNCTION_REFERENCES table function

Metric name	Entity name	Entity domain	Arguments	Schedule
INVALID_CATEGORY	PRODUCT_TBL	Table	[{"domain":"COLUMN","id":"202755","name":"CATEGORY"}]	TRIGGER_ON_CHANGES

14.4 Alerting users when data metrics exceed thresholds

Once you have your data metric functions configured and functioning on schedule, you want to avoid monitoring them manually by selecting from the data quality monitoring results on an ongoing basis. Most likely you will define thresholds that the outputs of the data metric functions should not exceed and send alerts when they do.

If you need to send a notification or perform an action when data in Snowflake meets certain conditions, you can set up a Snowflake *alert*. An alert is a schema-level object that specifies a condition that triggers the alert, the action to perform when the condition is met, and the schedule for when the condition is evaluated.

In our example, we will create an alert that checks whether the data metric functions reported any data quality violations within the last hour on tables in the DWH schema, and if so, sends an email to a designated recipient.

> **NOTE** For more information about Snowflake alerts, refer to the documentation at https://docs.snowflake.com/en/user-guide/alerts.

To allow the DATA_ENGINEER role to execute alerts, we must grant it the EXECUTE ALERT privilege by executing the following commands:

```
use role ACCOUNTADMIN;
grant execute alert on account to role DATA_ENGINEER;
```

Then we will use the DATA_ENGINEER role and the DQ schema to continue working:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema DQ;
```

Before we create an alert, let's first construct and test the query that we will specify as the condition in the alert. We need a query that sums the values reported by the data metric functions on all tables in the DWH schema within the last hour. The query should return results only when the sum of the values is greater than 0 because that will be the condition for sending the alert. The query that meets these requirements is shown in the following listing.

Listing 14.3 Selecting and filtering values reported by data metric functions

```
select sum(value)
  from SNOWFLAKE.LOCAL.DATA_QUALITY_MONITORING_RESULTS
```

```
  where table_database = 'BAKERY_DB'
  and table_schema = 'DWH'
  and measurement_time > dateadd('hour', -1, current_timestamp())
  having sum(value) > 0;
```

The output of this query should be a number greater than 0 if any data metric functions reported values greater than 0 within the last hour. Depending on when you executed your data metric functions, you may have to adjust the time interval in which your data metric functions reported values greater than 0 to include them in your result.

Now we will create an alert named `DATA_QUALITY_MONITORING_ALERT`, which we will initially schedule to execute every 5 minutes so we can test it. The condition in the alert is the query from listing 14.3, and the action to perform is sending an email using the `SYSTEM$SEND_EMAIL` function as described in chapter 13. The command that creates the alert is

```
create alert DATA_QUALITY_MONITORING_ALERT
  warehouse = BAKERY_WH
  schedule = '5 minute'
if (exists(
  select sum(value)
  from SNOWFLAKE.LOCAL.DATA_QUALITY_MONITORING_RESULTS
  where table_database = 'BAKERY_DB'
  and table_schema = 'DWH'
  and measurement_time > dateadd('hour', -1, current_timestamp())
  having sum(value) > 0
))
then
  call SYSTEM$SEND_EMAIL(
  'PIPELINE_EMAIL_INT',
  'firstname.lastname@youremail.com',
  'Data quality monitoring alert',
  'Data metric functions reported invalid values since ' ||
    to_char(dateadd('hour', -1, current_timestamp()),
    'YYYY-MM-DD HH24:MI:SS') || '.'
);
```

The query from listing 14.3

Sends an email

Substitutes your email address

Like tasks, alerts are suspended when we create them, and we must resume them so they start working:

```
alter alert DATA_QUALITY_MONITORING_ALERT resume;
```

To check the execution status of an alert, we can select from the `ALERT_HISTORY()` table function:

```
select * from table(information_schema.alert_history())
order by scheduled_time desc;
```

The output of the previous command looks something like table 14.9 (not all columns are shown).

Table 14.9 Output from the ALERT_HISTORY() table function

Name	Condition	Action	State	Scheduled time
DATA_QUALITY_MONITORING_ALERT	select sum(value) from…	call SYSTEM$SEND_EMAIL(…	SCHEDULED	2023-09-03 20:37:22

Table 14.9 shows that the alert is scheduled. It should trigger after about 5 minutes, which is the alert schedule. If the condition in the alert is met, it should send an email to your email address. The email should look something like the following listing.

Listing 14.4 Sample email from an alert

```
From: Snowflake Computing <no-reply@snowflake.net>
To: firstname.lastname@youremail.com
Subject: Data quality monitoring alert
--
Data metric functions reported invalid values since 2023-09-03 19:37:12.
```

We set the initial schedule in the alert to execute every 5 minutes so that we could test it. Now we want to change the schedule to execute every 60 minutes. To do that, we must suspend the task and change its schedule by executing the ALTER task command:

```
alter alert DATA_QUALITY_MONITORING_ALERT suspend;
alter alert DATA_QUALITY_MONITORING_ALERT set schedule = '60 minute';
```

To continue testing the alert, you should resume it so it starts working. Otherwise, leave it in a suspended state so it doesn't continue to consume resources.

14.5 Detecting data volume anomalies

In addition to testing for data integrity, data engineers also test for data completeness to verify that no data was lost due to various reasons, such as failures in the ingestion pipeline, network outages, or interruptions in the source system.

Data engineers can test whether the data volume ingested in the last period—for example, the last day, week, or month—matches the expected data volume based on historical trends. We can use machine learning (ML) algorithms to calculate the expected data volumes and compare them with the actual ingested volumes. Data engineers don't need specialized ML skills to do this because they can use the Snowflake ML *anomaly detection* functionality.

Anomaly detection is the process of identifying outliers in time-series data. We prepare historical data that contains a timestamp column and a value column. Then we train a ML model on the historical data and apply it to newly ingested data to check for anomalies against the expected values.

> **NOTE** Anomaly detection is one of the Snowflake ML functions. For more information, refer to the documentation at https://mng.bz/XVlp.

To illustrate anomaly detection with an example, we need more data than the small order data files we have been ingesting in previous chapters. We will continue working with the bakery example, but this time, we will simulate the order data the bakery might receive daily from a supermarket chain over several months.

14.5.1 Generating random data

For this exercise, we will generate random data representing supermarket orders and save this data in a table in the staging layer, simulating the order data a supermarket might provide.

> **TIP** Generating random data is a valuable skill for data engineers. Sometimes, real data may not be accessible due to privacy concerns or technical problems, but a large volume of data is still necessary to test performance or conduct demonstrations. In such cases, data engineers can generate random data.

We will start by using the `DATA_ENGINEER` role and working in the `STG` schema:

```
use role DATA_ENGINEER;
use warehouse BAKERY_WH;
use database BAKERY_DB;
use schema STG;
```

We will create a table named `COUNTRY_MARKET_ORDERS` and populate it with randomly generated values representing supermarket orders on individual days starting from November 1, 2023 and randomly adding days for the next six months or approximately 180 days.

We can calculate a random order delivery date using the `RANDOM()` function to generate a random number and the `UNIFORM()` function to distribute the random value uniformly between the provided lower and upper bounds—in our example, between 1 and 180—and adding this random value to the initial date represented as `'2023-11-01'`:

```
dateadd('day', uniform(1, 180, random()), '2023-11-01'::date)
```

We will generate a product ID as a random number uniformly distributed between 1 and 14, which is the range of product ID values in the `PRODUCT_TBL` table. The quantity of the ordered products will be a random number uniformly distributed between 500 and 1,000 (this range is arbitrary; you can experiment with different ranges if you wish).

We want a sufficient volume of data to work with, so we will generate 10,000 rows using the `GENERATOR()` table function. We will also summarize the data by day and product, using a common table expression named `RAW_DATA` to generate the data and then summarize it in the main SQL clause. The command that creates the `COUNTRY_MARKET_ORDERS` table and populates it with random data is

```
create or replace table STG.COUNTRY_MARKET_ORDERS as
with raw_data as (
  select
    dateadd('day', uniform(1, 180, random()),
      '2023-11-01'::date) as delivery_date,
    uniform(1, 14, random()) as product_id,
    uniform(500, 1000, random()) as quantity
  from table(generator(rowcount => 10000))
)
select
  'Country Market' as customer,
  delivery_date,
  product_id,
  sum(quantity) as quantity
from raw_data
group by all;
```

This table now contains uniformly generated random data. A subset of this data is shown in table 14.10. Your data will vary because it is randomly generated.

Table 14.10 Subset of data in the `COUNTRY_MARKET_ORDERS` table

Customer	Delivery date	Product ID	Quantity
Country Market	2023-11-02	3	2,373
Country Market	2023-11-02	4	3,860
Country Market	2023-11-02	5	4,410

In this exercise, we want to simulate data outages to demonstrate how the anomaly detection algorithm flags them. Therefore, we must remove some of the data to simulate the outages. Say the supermarket had major renovations across most of the stores between March 10 and March 15 when they were closed. As a result, they placed significantly fewer orders during this time—only about 20% of the normal quantities. To simulate this effect, we will update the `COUNTRY_MARKET_ORDERS` table by multiplying the quantity amount by 0.2 between these dates:

```
update STG.COUNTRY_MARKET_ORDERS
  set quantity = 0.2*quantity
  where delivery_date between '2024-03-10' and '2024-03-15';
```

Additionally, there might have been a network outage on March 21 and 22 when the supermarket couldn't deliver the order data. We will update the quantity to 0 on these dates:

```
update STG.COUNTRY_MARKET_ORDERS
  set quantity = 0
  where delivery_date between '2024-03-21' and '2024-03-22';
```

14.5.2 Displaying data as a line chart in Snowsight

Before we continue, let's look at the data distribution as a line chart in the Snowsight user interface. We will display the total quantity of all products ordered by day by selecting all data from the table and formatting the output as a chart:

```
select * from STG.COUNTRY_MARKET_ORDERS;
```

The results appear in the Results pane at the bottom in a tabular format by default in the Snowsight user interface. When you click the Chart option in the Results pane, you should see the results in graphical format. Choose `Line` as the chart type, `ORDER_DATE` as the x-axis column, and `QUANTITY` as the value column. The output is a chart that looks like figure 14.3. (your chart may vary somewhat due to the random nature of the data).

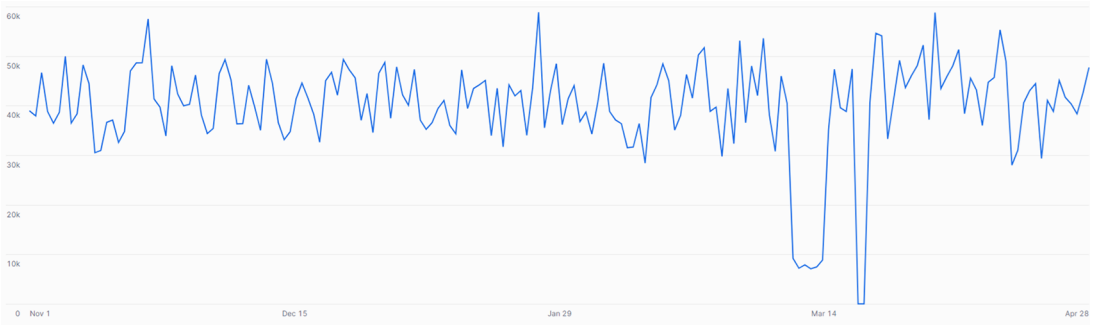


Figure 14.3 Line chart representing the total quantity of all products ordered by day. Because we randomly generated the data, we see fluctuating values between roughly 30,000 and 60,000 products per day. We also see that the quantities are significantly lower on March 10 to 15 and March 21 to 22 due to store renovations and network outages.

The chart in figure 14.3 clearly shows the dip in ordered quantities on March 10 to 15 and March 21 to 22, which represent anomalies in data distribution. We will continue the exercise to learn how the anomaly detection model flags them.

14.5.3 Working with the anomaly detection model

To work with the anomaly detection model, we must grant the `DATA_ENGINEER` role the `ANOMALY_DETECTION` privilege on a schema using the `ACCOUNTADMIN` role. We will grant the privilege in the `DQ` schema because that's where we will work with the model:

```
use role ACCOUNTADMIN;
grant create SNOWFLAKE.ML.ANOMALY_DETECTION
  on schema BAKERY_DB.DQ
  to role DATA_ENGINEER;
```

Then we continue working with the DATA_ENGINEER role in the DQ schema:

```
use role DATA_ENGINEER;
use schema DQ;
```

We will take all data ingested before March 1, 2024, as the historical data to train the model. The data ingested on or after March 1 is the new data where we will look for anomalies. To split the order data into historical and new data sets, we can create two views, named ORDERS_HISTORICAL_DATA and ORDERS_NEW_DATA, each summarizing the total quantity of all products ordered by day. We must convert the column representing the day to the TIMESTAMP data type because the anomaly detection model requires it. The queries that create the views are

```
create or replace view ORDERS_HISTORICAL_DATA as
  select delivery_date::timestamp as delivery_ts,
    sum(quantity) as quantity
  from STG.COUNTRY_MARKET_ORDERS
  where delivery_date < '2024-03-01'
  group by delivery_ts;

create or replace view ORDERS_NEW_DATA as
  select delivery_date::timestamp as delivery_ts,
    sum(quantity) as quantity
  from STG.COUNTRY_MARKET_ORDERS
  where delivery_date >= '2024-03-01'
  group by delivery_ts;
```

Now we can train an anomaly detection model named ORDERS_MODEL. The model takes the ORDERS_HISTORICAL_DATA view as the input data, DELIVERY_TS as the timestamp column, and QUANTITY as the target column. We can leave the label column empty when training this model (we would have used the label column if we were training the model for each product ordered).

> **NOTE** We must use the SYSTEM$REFERENCE() function when specifying the input data parameter because the anomaly detection model doesn't have access to the ORDERS_HISTORICAL_DATA view. That's why we are providing it as a reference using the role of the user executing the command. For more information, refer to the documentation at https://mng.bz/yojy.

To train the model, we can call the ANOMALY_DETECTION model in the ML schema in the SNOWFLAKE database by executing the following command:

```
create or replace SNOWFLAKE.ML.ANOMALY_DETECTION orders_model(
  input_data => SYSTEM$REFERENCE('VIEW', 'ORDERS_HISTORICAL_DATA'),
  timestamp_colname => 'delivery_ts',
  target_colname => 'quantity',
  label_colname => '');
```

The trained model is stored as an object named ORDERS_MODEL in the DQ schema, which is the current schema. We can now use this model that we trained on historical

data to detect anomalies in the new data by calling the DETECT_ANOMALIES method, providing ORDERS_NEW_DATA as the input data, DELIVERY_TS as the timestamp column, and QUANTITY as the target column:

```
call orders_model!DETECT_ANOMALIES(
  input_data => SYSTEM$REFERENCE('VIEW', 'ORDERS_NEW_DATA'),
  timestamp_colname =>'delivery_ts',
  target_colname => 'quantity'
);
```

The output of the model is displayed in the results pane and looks like table 14.11 (a few rows and columns are shown for illustration). Your results may differ due to the random nature of the data.

Table 14.11 Output of the anomaly detection model

Timestamp	Y	Forecast	Lower bound	Upper bound	Is anomaly
2024-03-09 00:00:00	44138	44066.78211	27590.13250	60543.43172	FALSE
2024-03-10 00:00:00	8359	38713.12106	22236.47145	55189.77067	TRUE
2024-03-11 00:00:00	8247	41360.97774	24884.32813	57837.62735	TRUE
2024-03-12 00:00:00	9006	41548.65266	25072.00305	58025.30227	TRUE
2024-03-13 00:00:00	8727	38155.40186	21678.75225	54632.05147	TRUE
2024-03-14 00:00:00	8538	39290.73241	22814.08279	55767.38202	TRUE
2024-03-15 00:00:00	6048	37984.52279	21507.87318	54461.17240	TRUE
2024-03-16 00:00:00	40013	43966.23691	27489.58729	60442.88652	FALSE
2024-03-17 00:00:00	39752	38435.63136	21958.98175	54912.28097	FALSE
2024-03-18 00:00:00	45663	36295.49338	19818.84377	52772.14299	FALSE
2024-03-19 00:00:00	42644	30039.25741	13562.60780	46515.90702	FALSE
2024-03-20 00:00:00	48265	39961.44904	25183.97253	54738.92555	FALSE
2024-03-21 00:00:00	0	40695.38879	25917.91228	55472.86530	TRUE
2024-03-22 00:00:00	0	42933.93034	28156.45383	57711.40684	TRUE
2024-03-23 00:00:00	41535	43236.03100	28458.55449	58013.50751	FALSE

As shown in table 14.10, the results include a forecast value for each day with a lower and upper bound and a flag indicating whether the actual value (column Y in the output) is an anomaly compared to the forecast value. In this sample output, we can see that the zero values on March 21 and 22 are marked as anomalies, as indicated by the value TRUE in the last column. The 20% values between March 10 and 15 are also marked as anomalies, as indicated by the value TRUE in the last column.

We will save the model output so we can access it later to a table named ORDERS_MODEL_ANOMALIES using the RESULT_SCAN() table function on the last query that was executed:

```
create or replace table ORDERS_MODEL_ANOMALIES as
select * from table(result_scan(last_query_id()));
```

With the results saved in the ORDERS_MODEL_ANOMALIES table, the data engineer can build an alert like in the previous exercise to send an email whenever any data volumes are unusual, as flagged by the anomaly detection model. The condition in the alert would be a query that checks if any values in this table contain the value TRUE in the IS_ANOMALY column.

Summary

- Data testing involves adding steps to data pipelines that validate data quality and identify data quality violations. These steps can be executed synchronously by including them in the pipeline or asynchronously by separating the data ingestion functionality from the data testing functionality.
- Based on the outcome of the data testing steps, the data engineer can implement various actions in the pipeline, such as stopping the pipeline execution, notifying designated recipients of the test results, or logging the test results for later inspection.
- To test for data quality as a step in the data pipeline, we can construct a query that performs data testing and implement it as a task that inserts the results into a logging table.
- We can insert data into the logging table only when rows that violate the data quality rules exist. Depending on the user requirements, we can insert rows even when no records violate the data quality rules to reassure the business users that a data testing step was performed but no errors were found.
- Snowflake provides built-in system data metric functions that measure standard data quality metrics. You can also define custom data metric functions according to your specific requirements, which you store in a designated schema.
- Data metric functions are assigned to tables or views together with a schedule for how frequently they are evaluated. The results from the data metric function execution are stored in the event table in the Snowflake account. Since the data metric functions execute according to their defined schedule, we don't include them as steps in data pipelines.
- If you need to send a notification or perform an action when data in Snowflake meets certain conditions, you can set up a Snowflake alert. An alert is a schema-level object that specifies a condition that triggers the alert, the action to perform when the condition is met, and the schedule for when the condition is evaluated.

- We can create an alert that checks whether the data metric functions reported any data quality violations within a chosen time interval, and if so, sends an email to a designated recipient.
- Data engineers test for data completeness to verify that no data was lost due to various reasons, such as failures in the ingestion pipeline, network outages, or interruptions in the source system.
- We can use ML algorithms to calculate the expected data volumes and compare them with the actual ingested volumes. Data engineers can use the Snowflake ML anomaly detection functionality, which identifies outliers in time series data.
- To use anomaly detection, we prepare historical data that contains a timestamp column and a value column. Then we train an ML model on the historical data and apply it to newly ingested data to check for anomalies against the expected values.
- Generating random data is a valuable skill for data engineers. Sometimes, real data may not be accessible due to privacy concerns or technical issues, but a large volume of data is still necessary to test performance or conduct demonstrations.

15 Data pipeline continuous integration

This chapter covers

- Separating the data engineering environments
- Database change management
- Configuring Snowflake to use Git
- Using the Snowflake CLI command line interface
- Connecting to Snowflake securely

In previous chapters, we gradually built data pipelines by adding various pieces of functionality. As our knowledge expanded, we created many scripts and files, saving them across multiple chapter folders in the accompanying GitHub repository. This has made it challenging to locate a specific script for maintenance. A more practical solution for organizing the data pipeline code would be to store the scripts in an organized manner in the repository. A centralized code repository is essential when multiple data engineers work on the same data pipelines, allowing them to locate scripts effortlessly and merge their code changes into the shared codebase.

Continuous integration is a software development practice in which data engineers frequently merge their code changes into the repository. After the merge, automated scripts execute the code, create database objects, perform integration tests, and carry out other actions as needed. Data engineers typically develop data

pipelines in a dedicated development environment. Once they finish development, they deploy the code into a test environment, where the quality assurance team tests it before deploying it to a production environment. The process where data engineers automate the deployment process between the environments is called *continuous delivery*.

CI/CD and DevOps

Continuous integration and continuous delivery are usually collectively called CI/CD. Another related term is *DevOps*, which encompasses the technological components of CI/CD and the cultural and people-oriented aspects of modern agile development. DevOps streamlines the creation, testing, and delivery of software solutions, like data engineering pipelines.

In this chapter, we will learn how to organize the data pipeline code in a Git repository, integrate it with Snowflake, and execute it using the SnowCLI command line interface. In addition to code, data pipelines interact with various database objects, so we will also cover how to maintain them. Additionally, we will discuss secure methods of connecting to Snowflake.

To illustrate the examples in this chapter, we will continue working with the fictional bakery introduced in chapter 2. To briefly recap, the bakery makes bread and pastries and delivers these baked goods to small businesses, such as grocery stores, coffee shops, restaurants, and hotels in the neighborhood. In chapter 11, we built a data pipeline that ingests data from two sources, the bakery's operational IT system and order information from the bakery's customers, stored as JSON files in cloud storage. The pipeline transforms the data through the extraction, staging, data warehouse, and presentation layers. In chapter 12, we modified the data pipeline to ingest data incrementally to avoid processing all data every time the pipeline executes. In chapter 13, we added orchestration to the data pipeline in the form of Snowflake tasks so that all steps are executed in sequence and on schedule. In this chapter, we will reorganize the code in the repository and add scripts enabling us to perform continuous integration.

NOTE All code and scripts for this chapter are available in the accompanying GitHub repository in the Chapter_15 folder at https://mng.bz/M1Go. The files are organized in folders described in more detail in this chapter. Unlike in previous chapters, where the exercises relied on you having completed the exercises in preceding chapters, this chapter assumes that none of the objects from the earlier chapters exist. But don't fret if you are working with the same Snowflake account where you have already created the objects from exercises in the previous chapters. The code in the exercises in this chapter is designed to overwrite objects if they already exist.

15.1 Separating the data engineering environments

Data engineers usually work in different environments to ensure a controlled development and testing process before the code is implemented in production. In the *development environment*, they create, unit test, and maintain the data pipeline code. In this environment, they can build prototypes, compare different approaches, and optimize the code before saving it in the repository.

In the *test environment*, developers and quality assurance specialists test the data pipeline's functionality, performance, and reliability. To perform such tests, they need data that the pipelines can operate on. As described in table 15.1, they require different data depending on the type of test they are performing.

Table 15.1 Types of data testing and the data they require

Test type	Data requirements
Data pipeline functionality	Small data sample
Data pipeline performance	Data volume similar to the production data volume
Data integrity	Data that closely resembles the data from the production environment
Data governance features	When production data contains sensitive information that should not be shared in the development and test environments, generated data or masked data can be used for testing.

The *user acceptance test environment* (UAT) is where the business users validate the data pipeline functionality, data correctness and completeness, data governance features, and any other pipeline features before the pipelines are released to production.

A *production environment* is where the data pipelines execute on live data and provide the required data structures for downstream consumers. It is configured to ensure optimal performance, reliability, and security.

Not all environments are required in all situations. In small organizations, a development and a production environment are usually sufficient. In this case, the data pipelines are developed and tested in the development environment before they are released to production.

Most organizations use the development, test, and production environments. The test environment is needed to perform various tests to ensure the pipelines are error-free before being deployed to production.

Large organizations often have a *user acceptance test environment* in addition to the development, test, and production environments. This environment allows the business users to perform acceptance tests within a separate environment where they have access to data with similar security and performance requirements as in the production environment.

Figure 15.1 illustrates the development, test, and production environments where two data engineers develop data pipeline code and share it in the development environment using continuous integration.

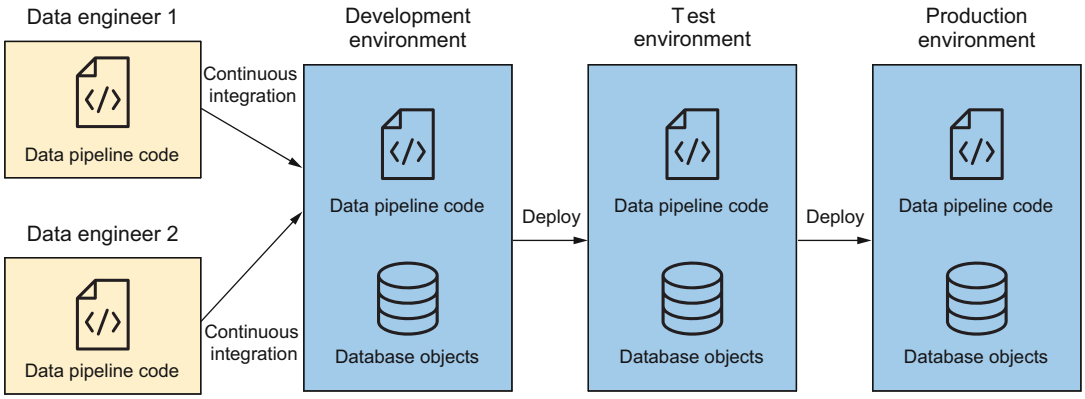


Figure 15.1 Two data engineers develop data pipeline code and share the code in the development environment using continuous integration. Once the development is completed, the code is deployed to the test environment to ensure it is error-free before it is deployed to the production environment.

15.2 Database change management

In addition to data pipeline code, data engineering solutions work with database objects, such as tables, views, dynamic tables, streams, tasks, and so on. We created many of these database objects in the data pipelines we developed in the previous chapters. Data engineers maintain these objects together with the data pipeline code, keeping track of the versions between releases and integrating them with the code.

Database change management (DCM) is the practice of defining all database objects in code and deploying them, including changes to them, to a database. This is an integral part of the automated continuous integration process.

DCM third-party tools

This chapter explains the Snowflake native database change management functionality to deploy and maintain database objects.

An alternative approach to the Snowflake native functionality is to use third-party database change management tools. Many such tools exist, including the `schemachange` library, Flyway, DataOps.live, SqlDBM, and others.

Some IaS (infrastructure as code) tools, such as Terraform, Pulumi, and others, are sometimes used to deploy Snowflake objects.

15.2.1 Comparing the imperative and the declarative approach to DCM

There are generally two approaches to maintaining database objects, imperative and declarative, as described in the following sections.

IMPERATIVE APPROACH

In the *imperative* approach, we create an object—for example, a table with columns—by executing a `CREATE TABLE` statement. When we change the table, such as by adding

a column, we execute an ALTER TABLE statement. Subsequent changes to the table require additional ALTER statements that must be executed in order.

There are many challenges with the imperative approach, including

- Manually writing the ALTER scripts for each change.
- Versioning the scripts to ensure they are applied in the correct order.
- Recreating an object at a specific point in time after a partial rollback is complicated and error-prone, requiring manual intervention by a data engineer.

DECLARATIVE APPROACH

In the *declarative* approach, we define the database objects according to the state they should be in at a given time. The database change management tool maintains the objects by figuring out the changes required from the previous version of the object. Snowflake supports the declarative approach to database change management with the CREATE OR ALTER command.

For example, we create an initial table using the CREATE TABLE command. When we change the table, such as by adding a column, we again use the CREATE TABLE command with the new table structure, adding the OR ALTER clause. When executed, the CREATE OR ALTER command creates the object if it doesn't yet exist or alters the existing object to match the new definition of the object.

The advantages of the declarative approach are that you can

- Write the script that defines the object with the specific version without having to know the previous version
- Apply the scripts as needed for the current version, regardless of the sequence of previous changes
- Recreate an object at a specific point in time after a partial rollback by only running the object definition script for that time

Table 15.2 illustrates the comparison between the imperative and declarative approaches to database change management.

Table 15.2 An example illustrating the comparison between the imperative and declarative approaches to database change management

Step	Imperative approach	Declarative approach

Creating a table	`create table EMPLOYEE(`
	`  id integer,`
	`  name varchar,`
	`  SSN varchar);`
		`create table EMPLOYEE(`
		`  id integer,`
		`  name varchar,`
		`  SSN varchar);`

Adding a column	`alter table EMPLOYEE`
	`add column birth_dt date;`
		`create or alter`
		`table EMPLOYEE(`
		`  id integer,`
		`  name varchar,`
		`  SSN varchar,`
		`  birth_dt date);`

Table 15.2 An example illustrating the comparison between the imperative and declarative approaches to database change management *(continued)*

Step	Imperative approach	Declarative approach
Dropping a column	`alter table EMPLOYEE drop column SSN;`	`create or alter table EMPLOYEE( id integer, name varchar, birth_dt date);`

As shown in table 15.2, in the imperative approach, we start with a `CREATE TABLE` statement, followed by a series of `ALTER TABLE` statements defining the changes to the table. In the declarative approach, we begin with the same `CREATE TABLE` statement as in the imperative approach, followed by a series of `CREATE OR ALTER` table statements defining the table state after each change.

15.2.2 Organizing the code in the repository

For efficient database change management, we should store the code that maintains the database objects in a well-structured format in the Git repository. An intuitive way to organize the code is to structure it in folders that follow the same hierarchy as the objects in Snowflake.

For example, when we open the Snowsight user interface and navigate to the Databases pane either in the main menu or from a worksheet, we see a list of databases. When we expand each database, we see a list of schemas. When we further expand each schema, we see folders for the types of objects the schema contains, such as tables, views, stages, and so on. We will organize the code to maintain the database objects used in our pipelines in the Git repository similarly.

Since we use only one database, `BAKERY_DB`, in our data pipelines, we will skip a databases folder to keep the organization simple for the exercises in this chapter. We will start with a schemas folder, the highest level in the folder structure. Under the schemas folder, we will create a folder for each schema that stores the database objects, like `ext`, `stg`, `dwh`, `mgmt`, and `orchestration`. Under each schema folder, we will create additional folders for the object types contained within the schema, and finally, under each folder, we will add the SQL scripts that create each object.

This folder structure is in the accompanying GitHub repository in the Chapter_15 folder. There is a folder named Snowflake_objects, and when you expand this folder, you should see the structure described in the previous paragraph. A sample representation of the folder structure and object creation scripts is shown in figure 15.2 (not all schemas and not all object types are included).

To support the declarative approach to database change management, each SQL script that creates an object is designed to represent the object state at the time the script is executed. For creating tables, the scripts use the `CREATE OR ALTER` command. For creating other types of objects, the scripts contain the `CREATE OR REPLACE` command so that the script doesn't result in an error if the object already exists.

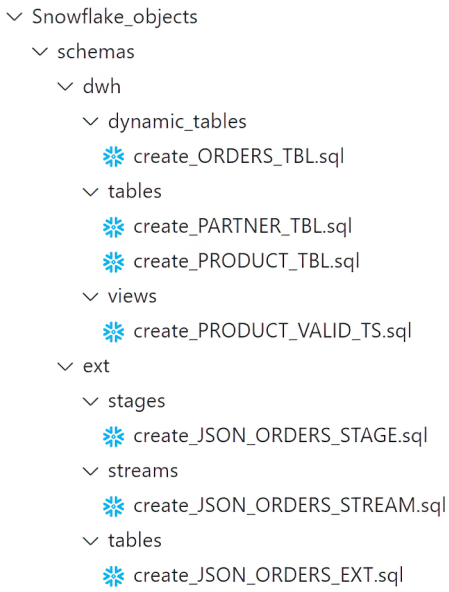


Figure 15.2 Sample representation of the folder structure and object creation scripts (not all schemas and not all object types are included)

> **TIP** When data engineers develop data pipeline code, they often make multiple changes to the object definitions until they are satisfied with the solution. As a result, they need to run and rerun the database change management scripts many times. Therefore, it is beneficial to create the scripts in a way that allows them to be re-executed multiple times without throwing an error if an object already exists.

You will recognize the database objects in the SQL scripts because we worked with them in chapters 11, 12, and 13. For example, the `EXT` schema contains the following objects required for ingesting data from orders stored as JSON files in an external object storage location: the `JSON_ORDERS_STAGE` external stage, the `JSON_ORDERS_EXT` table, and the `JSON_ORDERS_STREAM` stream.

> **TIP** The database object creation scripts include scripts that create Snowflake tasks, as in chapter 13. The `PIPELINE_START_TASK` and `PIPELINE_END_TASK` tasks in the `ORCHESTRATION` schema are configured to send emails when the pipeline execution starts and ends. Please edit these files using your email address before continuing with the exercise.

Appendix B contains a complete list of the objects we are working with in this exercise, along with their descriptions.

15.3 Configuring Snowflake to use Git

Now that the database object creation scripts are organized neatly in a Git repository, we want to connect the repository with our Snowflake account so we can execute the code in Snowflake.

Working with Git repositories

This chapter assumes you have the knowledge to work with Git repositories, including the basic functionality such as committing, pushing, and pulling code, creating branches, raising pull requests, merging branches, and forking a repository.

If you are not familiar with this functionality, you can learn about working with Git repositories from numerous tutorials that you can find by searching the Internet. You can also learn by completing a project using the *Getting Started with Git* resource at https://mng.bz/aV2o.

15.3.1 Creating a Git repository stage

To configure Snowflake to use Git, we will create the following objects:

- A secret to store the credentials that Snowflake uses to authenticate with the repository, such as a username and password of the Git account or a personal access token
- An API integration to provide details about how Snowflake interacts with the Git repository API using `git_https_api` as the provider
- A Git repository stage to expose the files in the repository to Snowflake

The described objects and their relationships are illustrated in figure 15.3.

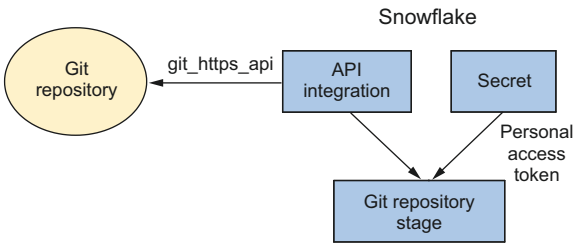


Figure 15.3 Configuring Snowflake to use Git consists of creating a secret containing credentials that Snowflake uses to authenticate with the repository, an API integration with the Git repository API, and a Git repository stage that exposes the files in the repository to Snowflake.

We will now proceed to configure our Snowflake account to use Git. For the examples in this exercise, we will use the same Git repository that hosts the sample code for this book located in the author's GitHub account at https://mng.bz/gAnx.

To follow along, you must use your own Git repository with your account. If you use GitHub, you can fork the author's repository into your account. If you use a different Git hosting provider, such as GitLab, Bitbucket, or others, you can export the code from the GitHub repository and import it into your account.

Before we create the secret, API integration, and Git repository stage objects, let's consider where we will store them. Initially, no databases and schemas are created in a new Snowflake account. We must create a database and a schema first before we can configure the Git repository, which hosts the code that creates additional database objects.

NOTE: The commands for creating the Git repository integration objects are in the `Chapter_15_Part1_setup.sql` script in the Chapter_15 folder in the accompanying GitHub repository.

To keep the exercise simple, we will use the `SYSADMIN` role to create a database named `ADMIN_DB` and a schema named `GIT_INTEGRATION`:

```
use role SYSADMIN;
create database ADMIN_DB;
create schema GIT_INTEGRATION;
```

Then, continuing to use the `SYSADMIN` role, we will create a secret named `GIT_SECRET` to store the username of your Git hosting account and the password or personal access token by executing the following command (substitute your username and password in the brackets and remove the brackets):

```
create or replace secret GIT_SECRET
  type = password
  username = <your Git username>
  password = <your password>;
```

Next, we will create an API integration named `GIT_API_INTEGRATION`. To keep the exercise simple, we will use the `ACCOUNTADMIN` role to create the integration by executing the following commands:

```
use role ACCOUNTADMIN;
create or replace api integration GIT_API_INTEGRATION
  API_PROVIDER = git_https_api
-- replace your Git account in the next line
  API_ALLOWED_PREFIXES = ('https://<your Git host>/<your Git account>')
  ALLOWED_AUTHENTICATION_SECRETS = (GIT_SECRET)
  ENABLED = TRUE;
```

We will grant usage on the API integration to the `SYSADMIN` role because we will continue to work with this role going forward:

```
grant usage on integration GIT_API_INTEGRATION to role SYSADMIN;
```

Using the `SYSADMIN` role, we will create a Git repository stage named `SF_DE` (this is an acronym for the title of this book, *Snowflake Data Engineering*), providing the `GIT_API_INTEGRATION API` integration and the `GIT_SECRET` secret we created earlier (substitute the URL to your repository in the `ORIGIN` parameter):

```
use role SYSADMIN;
create or replace git repository SF_DE
  api_integration = GIT_API_INTEGRATION
  git_credentials = GIT_SECRET
-- replace the URL to your repository in the next line
  ORIGIN = 'https:// <your Git host>/<your Git account>/SF_DE.git';
```

> **TIP** In a real-life scenario, you should create one or more custom roles and grant them the `CREATE SECRET ON SCHEMA`, `CREATE INTEGRATION ON ACCOUNT`, and `CREATE GIT REPOSITORY ON SCHEMA` privileges; then use the custom roles to create the respective objects.

Now that the Git repository is configured, we can fetch the latest files by executing the `ALTER GIT REPOSITORY` command and providing the `FETCH` keyword:

```
alter git repository SF_DE fetch;
```

We can use the `SHOW GIT BRANCHES` command to show the branches in the Git repository:

```
show git branches in SF_DE;
```

If you forked or imported this book's repository into your Git hosting account and haven't made any changes, it should contain only the main branch, giving output like in table 15.3 (the last column is abbreviated).

Table 15.3 Sample output when showing branches in the Git repository

Name	Path	Checkouts	Commit hash
main	/branches/main		`638e43e6…`

You can list the files in the repository main branch by executing the `LS` command:

```
ls @SF_DE/branches/main;
```

The output of this command should list all the files in the repository's main branch, as shown in table 15.4 (the first few rows and columns are shown).

Table 15.4 Sample output when listing the files in the repository main branch

Name	Size
sf_de/branches/main/.gitignore	23
sf_de/branches/main/Chapter_02/Chapter_02.sql	6,249
sf_de/branches/main/Chapter_02/Orders_2023-07-07.csv	3,310

> **NOTE** For more information about configuring Snowflake to use Git and a detailed reference about the commands you can execute to work with the files in the repository stage, refer to the Snowflake documentation at https://mng.bz/5OVZ.

15.3.2 Executing commands from a Git repository stage

After configuring the Git repository stage, we want to execute the code in the repository. We can do that with the `EXECUTE IMMEDIATE FROM` command in a Snowflake worksheet. Let's do this next.

As we learned in chapter 10, we will set up role-based access control (RBAC) in our data pipelines to have custom roles that work with database objects rather than the Snowflake built-in roles. You should see the `Chapter_15_Part2_RBAC.sql` script in the accompanying GitHub repository in the Chapter_15 folder. This script creates objects like the `BAKERY_WH` virtual warehouse; the `BAKERY_DB` database; the `EXT`, `STG`, `DWH`, `MGMT`, and `ORCHESTRATION` schemas; the access roles associated with the schemas; and the `DATA_ENGINEER` and `DATA_ANALYST` functional roles. It also grants usage on the Git repository stage to the `DATA_ENGINEER` role.

As we haven't created any custom roles yet, we will use the `SYSADMIN` role, the `ADMIN_DB` database, and the `GIT_INTEGRATION` schema to execute this script from the Git repository stage. Before executing the `Chapter_15_Part2_RBAC.sql` script with the `EXECUTE IMMEDIATE FROM` command, we will fetch the latest changes from the Git repository.

> **TIP** Although we haven't made any changes to the Git repository, it's best practice to always fetch the latest files from the repository in case any other data engineers made changes to the code in the meantime.

The commands we will execute are

```
use role SYSADMIN;
use database ADMIN_DB;
use schema GIT_INTEGRATION;

alter git repository SF_DE fetch;

execute immediate from
     @SF_DE/branches/main/Chapter_15/Chapter_15_Part2_RBAC.sql;
```

The output from this command should be a status indicating that the statement was executed successfully.

After executing the RBAC script, the `DATA_ENGINEER` and `DATA_ANALYST` functional roles should be created. We will grant both functional roles to our current user to be able to use them by executing the following commands (replace your username before executing the commands):

```
use role USERADMIN;
grant role DATA_ENGINEER to user <your username>;
grant role DATA_ANALYST to user <your username>;
```

To test, we can now switch to the `DATA_ENGINEER` role and see if we can list the files in the Git repository stage:

```
use role DATA_ENGINEER;
use database ADMIN_DB;
use schema GIT_INTEGRATION;
ls @SF_DE/branches/main;
```

You should see output like in table 15.4.

The code in the scripts we executed until now represents the setup steps required to create the environment in which we will develop the data pipelines and the associated database objects. In the future, we will assume that data engineers work on the data pipeline code development in a development environment integrated with a Git repository where they commit their code and perform continuous integration.

Rather than continuing to work in the Snowsight user interface, where we can execute SQL commands interactively, we will switch to an integrated development environment (IDE) to build the data pipelines, as described in the next section.

15.4 Using the Snowflake CLI command line interface

As mentioned in chapter 6, where we introduced Snowpark, we can use Microsoft Visual Studio Code as the IDE, but you can also use other development environments.

Executing code using a command-line interface is convenient when developing in an IDE, especially since the same commands are used for continuous integration. Snowflake provides the Snowflake CLI, an open-source command-line tool designed for this purpose.

15.4.1 Installing and configuring Snowflake CLI

Since Snowflake CLI is a Python package, you must have installed Python version 3.8 or later. Then install the `snowflake-cli-labs` package using `pip`.

> **TIP** For detailed instructions on installing the Snowflake CLI, refer to the Snowflake documentation at https://mng.bz/6YxZ.

After installing the Snowflake CLI, check that it is working by executing the `snow` command in the command line interface in your IDE, using the `--help` option as the parameter:

```
snow --help
```

This command displays the help text that looks like the following listing (the first few lines are shown).

Listing 15.1 Sample output from the Snowflake CLI `help` command

```
Snowflake CLI tool for developers.
--version             Shows version of the Snowflake CLI
--info                Shows information about the Snowflake CLI
--config-file   FILE  Specifies Snowflake CLI configuration file
--help    -h          Show this message and exit
```

The next step is configuring the Snowflake connection, where we specify the username, password, and Snowflake account to which we will connect. We can use options in the `snow` command to provide these parameters, but storing them in a configuration file so we don't have to specify them each time we run a command is more convenient. The configuration file, config.toml, is in the ~/.snowflake folder.

Open the config.toml file with an editor of your choice and configure a default connection named `my_connection` by adding the following lines, replacing your Snowflake account, username, and password:

```
default_connection_name = "my_connection"

[connections.my_connection]
account = "<your Snowflake account>"
user = "<your username>"
password = "<your password>"
database = "BAKERY_DB"
warehouse = "BAKERY_WH"
role = "DATA_ENGINEER"
```

Save the file, and test the connection by executing the following `snow` command in the command line interface in your IDE:

```
snow connection test
```

This command returns information about your Snowflake connection, as shown in the following listing (not all lines are shown).

Listing 15.2 Sample output from the Snowflake CLI `connection` command

```
+-------------------------------------------------------------+
| key             | value                                     |
|-----------------+-------------------------------------------|
| Status          | OK                                        |
| Account         | <your account>                            |
| User            | <your username>                           |
| Role            | DATA_ENGINEER                             |
| Database        | BAKERY_DB                                 |
| Warehouse       | BAKERY_WH                                 |
+-------------------------------------------------------------+
```

With the Snowflake CLI configured, we can now execute additional commands that execute SQL statements in the Snowflake account, operate on objects in the account, execute scripts in the repository, and much more.

TIP For a complete guide of the Snowflake CLI commands, refer to the documentation at https://mng.bz/o0g2.

Snowflake extension for Visual Studio Code

If you use Visual Studio Code as your IDE, you can take advantage of the Snowflake extension. This extension enables data engineers to execute Snowflake SQL statements directly in Visual Studio Code.

The Snowflake extension is not a replacement for Snowflake CLI. The difference is that the Snowflake extension is used interactively during development, while the Snowflake CLI is needed for continuous integration.

For more information about the Snowflake extension for Visual Studio Code, refer to the documentation at https://docs.snowflake.com/en/user-guide/vscode-ext.

15.4.2 Executing scripts with Snowflake CLI

We will use the Snowflake CLI to execute scripts in the repository in two different ways:

- To execute the scripts from the local files in the IDE in order to test them while we are developing
- To execute the scripts from the Snowflake Git repository stage after committing them into the repository

For the first option, to test the scripts in the IDE local files while we are developing, we use the `snow sql -f` command with the script file name as the parameter. For example, let's take the script that creates the `JSON_ORDERS_TBL_STG` table in the `STG` schema. According to the way we organized the object creation scripts in the repository as described earlier in this chapter, the script `create_JSON_ORDERS_TBL_STG.sql` that creates this object is in the Chapter_15/Snowflake_objects/schemas/stg/tables folder.

We can execute this script (assuming that the current working directory is `Chapter_15`) with the following command:

```
snow sql -f "Snowflake_objects/schemas/stg/tables/create_JSON_ORDERS_TBL_STG.sql"
```

This command executes the script stored in the `create_JSON_ORDERS_TBL_STG.sql` file, which contains a `CREATE OR ALTER` command to create this table.

During development, data engineers create many scripts and files, which they test individually using the `snow sql -f` command. Once the development is done, they commit their code to the Git repository. Depending on the workflow, they usually create a pull request that another developer reviews and approves before merging into the target branch in the repository.

After the code is committed and merged, we can fetch it in the Snowflake Git repository stage and execute it from the stage using the `EXECUTE IMMEDIATE FROM` syntax, as described earlier. To do this, we use the `snow sql -q` command to execute an SQL query and provide the query as the parameter.

To fetch the latest files from the repository into the Snowflake Git repository stage, we execute the following command:

```
snow sql -q "alter git repository ADMIN_DB.GIT_INTEGRATION.SF_DE fetch"
```

Then, to execute the script that creates the `JSON_ORDERS_TBL_STG` table, we can execute it from the Git repository stage with the following command:

```
snow sql -q "execute immediate from
     @ADMIN_DB.GIT_INTEGRATION.SF_DE/branches/main/Chapter_15/
Snowflake_objects/schemas/stg/tables/create_JSON_ORDERS_TBL_STG.sql"
```

The output of this command is a message indicating that the `JSON_ORDERS_TBL_STG` table was successfully created.

15.4.3 Continuous integration with Snowflake CLI

Executing scripts that create each object individually is convenient for testing, but when we want to create all objects during continuous integration or deployment to a new environment, we would instead execute a single command that calls all object creation scripts.

Snowflake supports using the `EXECUTE IMMEDIATE FROM` command in scripts, which enables us to create a single script that calls individual object creation scripts. An example of this script is in the `deploy_objects.sql` file in the Chapter_15/Snowflake_objects folder in the accompanying GitHub repository. The first few lines of this script are shown in the following listing.

Listing 15.3 The first few lines of the script that calls other scripts to create all objects

```
use database BAKERY_DB;
-- EXT schema
execute immediate from
     '../Snowflake_objects/schemas/ext/stages/create_JSON_ORDERS_STAGE.sql';
execute immediate from
     '../Snowflake_objects/schemas/ext/tables/create_JSON_ORDERS_EXT.sql';
execute immediate from
     '../Snowflake_objects/schemas/ext/streams/create_JSON_ORDERS_STREAM.sql'
     ;
```

As we can see in listing 15.3, the script uses the `BAKERY_DB` database and executes each object creation script using the `EXECUTE IMMEDIATE FROM` command. The path to the file in the command is relative to the current script's path, which is the `deploy_objects.sql` file.

To execute this script, we again fetch from the Git repository stage to ensure we have the latest version of the files like earlier:

```
snow sql -q "alter git repository ADMIN_DB.GIT_INTEGRATION.SF_DE fetch"
```

Then execute the script using the `EXECUTE IMMEDIATE FROM` command and providing the script name along with its path in the Git repository stage:

```
snow sql -q "execute immediate from
     @ADMIN_DB.GIT_INTEGRATION.SF_DE/branches/main/Chapter_15/Snowflake_objec
     ts/deploy_objects.sql"
```

The output of this command should indicate that the objects were created successfully. The deployment script creates all objects, including Snowflake tasks. Because Snowflake tasks are suspended upon creation, we must resume them for the pipelines to start working. We could have included statements to resume the tasks together with the deployment script. However, when data engineers perform continuous integration, they don't always want to execute the data pipelines. Sometimes, they just want to ensure that the code doesn't break, or they want to test individual pieces of the pipeline manually.

Therefore, we included two additional scripts named `resume_tasks.sql` and `suspend_tasks.sql` that resume and suspend the tasks, respectively. When we are ready to resume the pipelines so they start executing, we can execute the `resume_tasks.sql` script that resumes each task with the following command:

```
snow sql -q "execute immediate from
     @ADMIN_DB.GIT_INTEGRATION.SF_DE/branches/main/Chapter_15/Snowflake_objec
     ts/resume_tasks.sql"
```

When we want to stop pipeline execution, we execute the `suspend_tasks.sql` script that suspends the parent task with the following command:

```
snow sql -q "execute immediate from
     @ADMIN_DB.GIT_INTEGRATION.SF_DE/branches/main/Chapter_15/Snowflake_objec
     ts/suspend_tasks.sql"
```

Continuous delivery

In the examples in this chapter, we manually execute the `deploy_objects.sql` script, which creates all objects after committing our data pipeline code to the Git repository to perform continuous integration or the CI part of CI/CD.

To add continuous delivery, or the CD part of CI/CD, data engineers can automate execution of the deployment script by using functionality such as GitHub actions, GitLab CI, Bitbucket pipelines, Azure DevOps pipelines, or others, depending on their Git hosting provider.

Data engineers can also configure workflows that deploy the code and database objects to different environments, such as test or production environments, by changing the database connection parameters and executing the deploy scripts in the target environment.

15.5 Connecting to Snowflake securely

In the exercises throughout this book, we have used the username and password as the authentication method when connecting to Snowflake accounts. Admittedly, this is the easiest authentication method, especially when using Snowflake trial accounts that expire after 30 days and are disabled for further use after that time. However, this is also the least secure way of authenticating to Snowflake accounts.

Some of the more secure authentication methods include

- *Single sign-on* (SSO)—When working in a corporate environment, you might have SSO configured by your system administrators. In this method, you authenticate with your corporate identity provider, such as Microsoft AD or Okta, which passes the authentication to Snowflake.
- *Multifactor authentication* (MFA)—Each Snowflake user can enroll in MFA and configure their smartphone so they are asked to approve the login request each time they connect with Snowflake.
- *OAuth*—You can use OAuth in Snowflake by configuring a security integration object that acts as an interface between Snowflake and your application by generating access tokens to connect with Snowflake.
- *Key-pair authentication*—This authentication method requires a public and a private key, which you generate using the OpenSSL utility. You assign the public key to the Snowflake user who authenticates to Snowflake and the private key from your application when you connect with Snowflake.

Some authentication methods, such as MFA, are not practical in automated data pipeline deployments because they require manual user involvement to approve the login request. We will describe key-pair authentication as a more secure method of authenticating to Snowflake from data pipelines.

> **NOTE** For more information about Snowflake authentication methods, refer to the documentation at https://mng.bz/n0ae.

15.5.1 Configuring key-pair authentication

Before using key-pair authentication, you must generate a public and a private key using the OpenSSL software, available at https://www.openssl.org/. Instructions on generating the keys are available in the Snowflake documentation at https://mng.bz/vJ01.

> **TIP** To keep the exercise simple, use the unencrypted option and omit the passphrase when generating your private key.

Once your public and private keys are ready, you can proceed with the configuration. You will assign the public key to the Snowflake user who is connecting to Snowflake by executing the `ALTER USER` command. If you are using a Snowflake free trial account, you have the privileges to execute this command yourself, but if you are working in a corporate environment, you might have to ask your Snowflake administrator to execute

the following command, replacing the value of the public key that starts with `MII` with your public key that you generated earlier:

```
use role SECURITYADMIN;
alter user <your username> set rsa_public_key = 'MII...';
```

Then edit the config.toml file that stores your Snowflake CLI connection parameters as follows:

- Remove the line that contains your password since you will no longer be using this authentication method.
- Add a line that indicates your authentication method is `SNOWFLAKE_JWT` (this is used for key-pair authentication).
- Add a line that points to the file that contains your private key.

After making these changes, your config.toml file should contain the following lines, replacing your Snowflake account, username, and private key path:

```
default_connection_name = "my_connection"

[connections.my_connection]
account = "<your Snowflake account>"
user = "<your username>"
authenticator = "SNOWFLAKE_JWT"
private_key_path = "<path to your private key>"
database = "BAKERY_DB"
warehouse = "BAKERY_WH"
role = "DATA_ENGINEER"
```

Save the file, and then test the connection by executing the following `snow` command in the command line interface in your IDE:

```
snow connection test
```

If you configured the connection correctly, the output of this command should be like in listing 15.2, indicating that the key-pair authentication method is working. You can now execute the Snowflake CLI commands as you did earlier when we used the username and password as the authentication method.

15.6 Applying what we learned in real-world scenarios

As this is the final chapter of the book, let's recap what we learned. We started with a basic .csv file ingestion scenario and developed a complete data pipeline that ingests data from different sources in different formats, performs data transformation, and executes on schedule. We enhanced the data pipeline by incorporating outputs from large language models, implemented data testing steps, and optimized it for query performance and cost efficiency. We also learned how to manage the code in a repository.

Throughout the book, we used the bakery example to demonstrate various concepts. Data engineering pipelines in the real world are more complex than the simple bakery example. They deal with more data sources, larger data volumes, additional data transformation steps, and more intricate requirements from many business users.

While the book focuses on native Snowflake functionality, you may also utilize third-party tools in your data pipelines. Let this book be the foundation for the next steps in your data engineering journey in the real world.

Summary

- Continuous integration is a software development practice in which data engineers frequently merge their code changes into the repository. Continuous delivery happens after the merge when automated scripts execute the code, create database objects, perform integration tests, and carry out other actions as needed.
- Data engineers usually work in different environments to ensure a controlled development and testing process before the code is implemented in production. The environments are provisioned based on the needs of the organization and may include development, test, user acceptance test, or production environments.
- Database change management (DCM) is the practice of defining all database objects in code and deploying them, including changes to them, to a database. This is an integral part of the automated continuous integration process.
- Snowflake supports the declarative approach to database change management with the `CREATE OR ALTER` command.
- A Git repository stage is a Snowflake object that enables data engineers to execute code stored in the repository using the `EXECUTE IMMEDIATE FROM` command.
- Snowflake CLI is an open-source command-line tool designed for executing code. This is convenient when developing in an IDE, especially since the same commands are used for continuous integration.
- Using Snowflake CLI, we can execute scripts in the Git repository from the local files in the IDE to test them during development or from the Snowflake Git repository stage after committing them into the repository.
- Snowflake supports using the `EXECUTE IMMEDIATE FROM` command in scripts, which enables us to create a single script that calls individual object creation scripts. This script can be used for continuous integration and deployment.
- Using the username and password is the least secure method for authenticating to Snowflake. More secure methods include single sign-on (SSO), multifactor authentication (MFA), OAuth, and key-pair.
- Key-pair authentication is a more secure method of authenticating to Snowflake from data pipelines. This method requires generating a public and private key, assigning the public key to the user who will authenticate to Snowflake, and providing the path to the private key in the connection parameters.

appendix A Configuring your Snowflake environment

To learn data engineering with Snowflake, you can set up a Snowflake free trial account and configure your development environment.

A Snowflake free trial account includes 400 USD worth of credits, which is sufficient for the exercises in this book. The free trial account expires after 30 days. If you need more time to complete the exercises, you can sign up for another free trial account using the same information (name, email address, etc.) as in your previous free trial account. You cannot transfer your work from the old to the new Snowflake account.

> **TIP** Keep all your work organized in scripts in your Git repository or on your local file system so that you can execute them in a new Snowflake trial account after your previous account expires.

Instead of creating a new Snowflake free trial account when your previous account expires, you can convert your free trial account to a paid Snowflake account by providing your credit card details. After your initial 400 USD worth of credits is used up, you will be charged for any additional credits consumed every month.

> **WARNING** If you upgrade your free trial Snowflake account to a paid account and later decide you no longer need it, you will have to contact Snowflake support to deactivate the account.

The following sections describe the Snowflake free trial sign-up process and configuration in more detail.

A.1 Signing up for a Snowflake free trial account

To sign up for a Snowflake free trial account, navigate to the following URL: https://signup.snowflake.com/.

You will see a form asking you to fill in your personal information as follows:

- *First name*—Enter your first name(s).
- *Last name*—Enter your last name(s).
- *Company email*—Enter your company email address. If you don't currently work for a company, you can use your personal email address.
- *Company name*—Enter your company name. If you don't work for a company currently, you can enter your university name if you're a student or the title of this book if you are signing up to do the exercises in this book.
- *Role*—Select the role in your company from the drop-down list. If you don't currently work for a company, you can choose any role, such as "Student."
- *Country*—Select the location from where you are signing up for the free trial account from the drop-down list.
- *(Optional)*—Opt-out of emails about products, services, and events that Snowflake thinks may interest you; depending on the location from where you are signing up (e.g., a country in the European Union), you can choose to opt out of emails from Snowflake.

When you finish filling out the form, click the Continue button. You will see a new form asking you to choose details about your Snowflake account as follows:

- *Snowflake edition*—Choose your Snowflake edition as Standard, Enterprise, or Business Critical, with Enterprise as the default. You can keep this default selection because the Enterprise edition supports all the functionality this book covers.
- *Cloud provider*—Select the Snowflake cloud provider from Microsoft Azure, Amazon Web Services, or Google Cloud Platform. For most exercises in this book, it doesn't matter which cloud provider you choose. However, some newly released functionality, such as Snowflake Cortex LLM functions, may not yet be available on all cloud providers and in all regions.
- *Cloud provider region*—Select the region of your cloud provider. Although it doesn't matter which region you choose, you will experience better performance if you select a region that is the closest to your location geographically. An exception is when you want to work with newly released functionality that is not supported in all regions; then choose one of the supported regions.

WARNING If you want to work with some of the newly released functionality, such as Snowflake Cortex LLM functions, check the documentation to find out its availability in cloud providers and regions, and then choose one of the supported cloud providers and regions.

Once you have filled out all the information, click the box indicating that you have read and agree to the Snowflake Self Service On Demand Terms, and then click the Get Started button.

Soon after that, you will receive an email from Snowflake to the email account you used to sign up, asking you to activate your account.

Click the activation link. You will be directed to a page asking you to enter your Snowflake username and password.

> **NOTE** You will use the username and password you entered on the activation link page to log in to your Snowflake account going forward. Be sure to also take note of the URL to access your Snowflake account provided in the email.

Your new Snowflake account will likely already contain several worksheets with sample code for different types of workflows. You will also see a `SNOWFLAKE_SAMPLE_DATA` database you can use to test before populating your data.

You work with Snowflake via the web interface called Snowsight. You can find detailed instructions for working with the interface at https://mng.bz/4prD.

A.2 Installing and configuring Snowflake CLI

Snowflake CLI is an open-source command-line tool designed for modern development practices, such as DevOps, in addition to executing SQL statements.

You execute Snowflake CLI commands in your operating system command line interface or from the command line in your integrated development environment (IDE).

Since Snowflake CLI is a Python package, you install it as a package using `pip`, `pipx`, or `Homebrew`, depending on your operating system.

> **TIP** For detailed instructions on installing the Snowflake CLI, refer to the Snowflake documentation at https://mng.bz/QV9R.

After installing the Snowflake CLI, check that it is working by executing the `snow` command in the command line interface, using the `--help` option as the parameter:

```
snow --help
```

This command displays the help text as shown in the Snowflake documentation.

After a successful installation, you must configure your Snowflake connection, where you provide the username, password, and Snowflake account to which you will connect. You can use options in the `snow` command to specify these parameters, but storing them in a configuration file is more convenient.

> **TIP** For detailed instructions on configuring your Snowflake connection used with the Snowflake CLI, refer to the Snowflake documentation at https://mng.bz/XVl1.

After completing the configuration as per the instructions in the documentation, save the configuration file; then test the connection by executing the following `snow` command in the command line interface:

```
snow connection test
```

This command returns information about your Snowflake connection, as shown in the Snowflake documentation.

TIP For a complete reference on the Snowflake CLI commands, refer to the documentation at https://mng.bz/yojo.

appendix B Snowflake objects used in the examples

The examples in this book are based on the scenario of a fictional bakery that makes bread and pastries. The bakery delivers these baked goods to small businesses such as grocery stores, coffee shops, and restaurants in the neighborhood. The bakery data engineers ingest order data from the bakery's customers while the data analysts and the bakery manager view reports derived from the data.

To illustrate the examples, we made up various fictional use cases, such as ingesting data continuously from a food delivery service, using data from the Snowflake Marketplace to research areas of expansion, and augmenting data with outputs from large language models (LLMs).

In each chapter, we created Snowflake objects, such as tables, views, stages, streams, functions, and so on, to illustrate the examples. This appendix summarizes the objects used in each chapter and their descriptions for your reference.

B.1 Ingesting and transforming data from CSV files (chapter 2)

In chapter 2, we ingested data from CSV files by uploading the files manually to a Snowflake internal stage. We executed the `COPY` command to copy the data from the staged files to a Snowflake table and then transformed the data and automated the process with a task. The objects used are summarized in table B.1.

Table B.1 Objects used for ingesting data from a CSV file, copying it into a Snowflake table, transforming it, and automating the process

Database	Schema	Object type	Object name	Description
BAKERY_DB	ORDERS	Internal stage	ORDERS_STAGE	CSV files are manually uploaded to the internal stage via the Snowsight user interface.
BAKERY_DB	ORDERS	Table	ORDERS_STG	Data from the internal stage is copied into this table.
BAKERY_DB	ORDERS	Table	CUSTOMER_ORDERS	Data from the ORDERS_STG table is merged into this table.
BAKERY_DB	ORDERS	Table	SUMMARY_ORDERS	Data from the CUSTOMER_ORDERS table is transformed and summarized in this table.
BAKERY_DB	ORDERS	Task	PROCESS_ORDERS	A task that automates the previous steps

B.2 Ingesting data from a cloud storage provider (chapter 3)

In chapter 3, we ingested order files from a cloud storage provider(Amazon S3, Google Cloud Storage, or Microsoft Azure Blob Storage). We created a Snowflake storage integration object that authenticates Snowflake to the cloud storage provider. We used it to create an external stage with Microsoft Azure Blob Storage as the cloud storage provider. We proceeded to ingest data in two different ways:

1. By executing the COPY command to copy data from the external stage into a table in Snowflake
2. By creating an external table and materializing the data in Snowflake with a materialized view

The objects used are summarized in table B.2.

Table B.2 Objects used for ingesting data from CSV files hosted in a cloud storage provider using the COPY command to copy data from the external location into a Snowflake table and by creating an external table and materializing the data with a materialized view

Database	Schema	Object type	Object name	Description
N/A	N/A	Storage integration	BISTRO_INTEGRATION	A storage integration object that authenticates Snowflake to the cloud storage provider
BAKERY_DB	EXTERNAL_ORDERS	External stage	BISTRO_STAGE	CSV files are manually uploaded to the blob storage location referenced by the stage.

Table B.2 Objects used for ingesting data from CSV files hosted in a cloud storage provider using the COPY command to copy data from the external location into a Snowflake table and by creating an external table and materializing the data with a materialized view *(continued)*

Database	Schema	Object type	Object name	Description
BAKERY_DB	EXTERNAL_ORDERS	File format	ORDERS_CSV_FORMAT	The file format defines the format of the CSV files in the external stage.
BAKERY_DB	EXTERNAL_ORDERS	Table	ORDERS_BISTRO_STG	Data from the external stage is copied into this table.
BAKERY_DB	EXTERNAL_ORDERS	External table	ORDERS_BISTRO_EXT	An external table enables you to query data from an external stage, just like any other table in Snowflake.
BAKERY_DB	EXTERNAL_ORDERS	Materialized view	ORDERS_BISTRO_MV	A materialized view persists the data to improve query performance when using external tables.

B.3 Ingesting and flattening semi-structured data (chapter 4)

In chapter 4, we ingested semistructured data in JSON format from a cloud storage provider. Instead of Microsoft Azure Blob Storage, like in chapter 3, we used Amazon S3. We created an external stage and specified JSON as the file format. We then ingested the JSON data into a table in Snowflake and stored it in a VARIANT data type. We flattened the semistructured data into a relational format with a view.

We combined all the data we ingested so far (CSV files from an internal stage, CSV files from an external object storage location, and JSON files from an external object storage location) into a single view. We demonstrated how to encapsulate data transformations with stored procedures and implemented exception handling and logging. The objects used are summarized in table B.3.

Table B.3 Objects used for ingesting data from JSON files hosted in an external cloud storage provider, flattening the data with a view, and encapsulating data transformations with stored procedures

Database	Schema	Object type	Object name	Description
N/A	N/A	Storage integration	PARK_INN_INTEGRATION	A storage integration object that authenticates Snowflake to the cloud storage provider
BAKERY_DB	EXTERNAL_JSON_ORDERS	External stage	PARK_INN_STAGE	JSON files are manually uploaded to the blob storage location referenced by the stage.
BAKERY_DB	EXTERNAL_JSON_ORDERS	Table	ORDERS_PARK_INN_RAW_STG	Semistructured data from the external stage is copied into this table into a VARIANT data type.

Table B.3 Objects used for ingesting data from JSON files hosted in an external cloud storage provider, flattening the data with a view, and encapsulating data transformations with stored procedures *(continued)*

Database	Schema	Object type	Object name	Description
BAKERY_DB	EXTERNAL_JSON_ORDERS	View	ORDERS_PARK_INN_STG	View that flattens the semi-structured data into a relational format
BAKERY_DB	TRANSFORM	View	ORDERS_COMBINED_STG	View that combines data from all sources ingested so far
BAKERY_DB	TRANSFORM	Table	CUSTOMER_ORDERS_COMBINED	Combined data from the view is merged into this table.
BAKERY_DB	TRANSFORM	Stored procedure	LOAD_CUSTOMER_ORDERS	The stored procedure executes the merge command from the previous line.
BAKERY_DB	TRANSFORM	Event table	BAKERY_EVENTS	Event table that stores events logged from stored procedures
BAKERY_DB	TRANSFORM	Table	SUMMARY_ORDERS	Data from the CUSTOMER_ORDERS_COMBINED table is summarized in this table.
BAKERY_DB	TRANSFORM	Stored procedure	LOAD_CUSTOMER_SUMMARY_ORDERS	The stored procedure executes the summarization command from the previous line.

B.4 Continuous data ingestion and dynamic tables (chapter 5)

In chapter 5, we built a data pipeline that continuously ingests data from files whenever they appear in the external cloud storage using the Snowpipe functionality. Like ingesting files in bulk from a cloud storage location, as in the previous chapters, Snowpipe also requires a storage integration object and an external stage that points to the location of the files in the cloud storage location.

We then used dynamic tables to perform data transformation continuously after ingestion with Snowpipe. The objects used are summarized in table B.4.

Table B.4 Objects used for ingesting data continuously with Snowpipe and transforming with dynamic tables

Database	Schema	Object type	Object name	Description
N/A	N/A	Storage integration	SPEEDY_INTEGRATION	A storage integration object that authenticates Snowflake to the cloud storage provider
BAKERY_DB	DELIVERY_ORDERS	External stage	SPEEDY_STAGE	JSON files arrive continuously in the blob storage location referenced by the stage.

Table B.4 Objects used for ingesting data continuously with Snowpipe and transforming with dynamic tables *(continued)*

Database	Schema	Object type	Object name	Description
BAKERY_DB	DELIVERY_ORDERS	Table	SPEEDY_ORDERS_RAW_STG	Snowpipe copies data from the external stage into this table.
N/A	N/A	Notification integration	SPEEDY_QUEUE_INTEGRATION	A notification integration object configured with cloud messaging notifies Snowpipe when new files arrive in the object storage location.
BAKERY_DB	DELIVERY_ORDERS	Pipe	SPEEDY_PIPE	A Snowflake pipe object that contains a COPY statement used to load data from the external stage into a relational table
BAKERY_DB	DELIVERY_ORDERS	Dynamic table	SPEEDY_ORDERS	A dynamic table that materializes the results of a query specified in the dynamic table definition; in this case, it flattens and materializes the data in the SPEEDY_ORDERS_RAW_STG staging table

B.5 Executing code natively with Snowpark (chapter 6)

In chapter 6, we used Snowpark to streamline data pipeline development in an integrated development environment maintained in a shared code repository. We automated manual steps that must be performed by bakery employees, such as uploading a CSV file to an internal stage, with a pipeline developed using Snowpark Python. To illustrate some of the Snowpark functionality, we created and populated a date dimension with a flag indicating whether the date is a holiday. The objects used are summarized in table B.5.

Table B.5 Objects used for building data pipelines with Snowpark Python

Database	Schema	Object type	Object name	Description
BAKERY_DB	SNOWPARK	Internal stage	ORDERS_STAGE	An internal stage used by Snowpark to upload order files
BAKERY_DB	SNOWPARK	Stored procedure	PROC_IS_HOLIDAY	A stored procedure that returns TRUE or FALSE depending on whether the input date is a holiday
BAKERY_DB	SNOWPARK	Table	DIM_DATE	Generated date dimension with a flag indicating whether the date is a holiday

Table B.5 Objects used for building data pipelines with Snowpark Python *(continued)*

Database	Schema	Object type	Object name	Description
BAKERY_DB	SNOWPARK	Table	ORDERS_STG	The Snowpark code copies data from the internal stage into this table.
BAKERY_DB	SNOWPARK	View	ORDERS_HOLIDAY_FLG	View that joins the order data with the date dimension, adding a flag indicating whether the date is a holiday

B.6 Calling API endpoints and LLM functions (chapter 7)

In chapter 7, we used the external network access functionality to call API endpoints, such as retrieving customer reviews from an external website. We stored the results in a Snowflake table for further analysis. We called a Snowflake Cortex LLM function to help us understand the reviews and classify their sentiment. We also used an LLM function to help us interpret unstructured text from the body of an email and derive structured information from it. The objects used are summarized in table B.6.

Table B.6 Objects used for creating external network access and calling Snowflake Cortex LLM functions

Database	Schema	Object type	Object name	Description
BAKERY_DB	REVIEWS	Network rule	YELP_API_NETWORK_RULE	A network rule that represents an external network location, in this case the Yelp website
BAKERY_DB	REVIEWS	Secret	YELP_API_TOKEN	A secret that stores credentials for accessing the external network location
BAKERY_DB	REVIEWS	External access integration	YELP_API_INTEGRATION	An external access integration that encapsulates the network rule and the secret
BAKERY_DB	REVIEWS	Function	GET_CUSTOMER_REVIEWS	The function calls the Yelp API endpoint using the external access integration and returns the result in JSON format.
BAKERY_DB	REVIEWS	Table	CUSTOMER_REVIEWS	The result from calling the GET_CUSTOMER_REVIEWS function is flattened and stored in this table.
BAKERY_DB	REVIEWS	Stored procedure	READ_EMAIL_PROC	The stored procedure interprets unstructured text from customer order emails and returns results in structured format as a data frame.

Table B.6 Objects used for creating external network access and calling Snowflake Cortex LLM functions *(continued)*

Database	Schema	Object type	Object name	Description
BAKERY_DB	REVIEWS	Table	COLLECTED_ORDERS_FROM_EMAIL	The READ_EMAIL_PROC stored procedures save the data frame into this table.

B.7 Optimizing query performance and controlling cost (chapters 8 and 9)

In chapter 8, we executed queries using large amounts of data from the Snowflake Marketplace. We used Snowflake's query profile tool to understand the mechanics of query execution. We applied clustering and search optimization to tables to improve query performance.

In chapter 9, we discovered what contributes to Snowflake's cost and how to monitor credit consumption. We used Snowflake objects created in chapter 8, which are summarized in table B.7.

Table B.7 Objects used for optimizing query performance

Database	Schema	Object type	Object name	Description
CPG_RETAILERS_AND_DISTRIBUTORS	N/A	Share	CPG_RETAILERS_AND_DISTRIBUTORS	Shared database from the Snowflake Marketplace
CPG_RETAILERS_AND_DISTRIBUTORS	PUBLIC	Table	HARMONIZED_RETAILER_SALES	Table in the shared database used for the exercise
BAKERY_DB	RETAIL_ANALYSIS	Table	RETAILER_SALES	Table with enriched data from the shared database

B.8 Data governance and access control (chapter 10)

In chapter 10, we implemented the Snowflake role-based access control (RBAC) model, where access privileges are assigned to roles that are granted to users. We used row access policies so that users from different business units see only the data from their business unit. We also applied masking policies that mask sensitive data, such as personal or financial information. The objects used are summarized in table B.8.

Table B.8 Objects used for RBAC, row access policies, and masking policies

Database	Schema	Object type	Object name	Description
BAKERY_DB	RAW	Schema	RAW	Schema with managed access
BAKERY_DB	RPT	Schema	RPT	Schema with managed access

Table B.8 Objects used for RBAC, row access policies, and masking policies *(continued)*

Database	Schema	Object type	Object name	Description
BAKERY_DB	RAW	Table	EMPLOYEE	Table created and manually populated with sample data
BAKERY_DB	RPT	View	EMPLOYEE	View that selects data from the EMPLOYEE table
BAKERY_DB	DG	Row access policy	RAP_BUSINES_UNIT	A row access policy that grants access based on the mapping between the user's role and the business unit
BAKERY_DB	DG	Masking policy	ADDRESS_MASK	A masking policy that masks sensitive data from users with unauthorized roles

B.9 Designing data pipelines (chapter 11)

In chapter 11, we designed a data pipeline that ingests data from multiple sources and propagates the data through standard data transformation layers, such as extract, staging, data warehouse, and presentation. We set up role-based access control so that only authorized users can access data in the various layers of the pipeline. The objects used are summarized in table B.9.

Table B.9 Objects used in a data pipeline that ingests data from multiple sources and propagates the data through standard data transformation layers, including extract, staging, data warehouse, and presentation

Database	Schema	Object type	Object name	Description
BAKERY_DB	EXT	Schema	EXT	Schema with managed access
BAKERY_DB	STG	Schema	STG	Schema with managed access
BAKERY_DB	DWH	Schema	DWH	Schema with managed access
BAKERY_DB	MGMT	Schema	MGMT	Schema with managed access
N/A	N/A	Storage integration	PARK_INN_INTEGRATION	Created in chapter 4
BAKERY_DB	EXT	External stage	JSON_ORDERS_STAGE	An external stage that references the object storage location with JSON files
BAKERY_DB	EXT	Table	JSON_ORDERS_EXT	Data from the external stage is copied into this table.
BAKERY_DB	STG	View	JSON_ORDERS_STG	View that flattens the data from the JSON_ORDERS_EXT table
BAKERY_DB	STG	Table	PARTNER	Table created and manually populated with sample data, simulating data from the source system

Table B.9 Objects used in a data pipeline that ingests data from multiple sources and propagates the data through standard data transformation layers, including extract, staging, data warehouse, and presentation *(continued)*

Database	Schema	Object type	Object name	Description
BAKERY_DB	STG	Table	PRODUCT	Table created and manually populated with sample data, simulating data from the source system
BAKERY_DB	DWH	View	PARTNER	View exposing data from the PARTNER table
BAKERY_DB	DWH	View	PRODUCT	View exposing data from the PRODUCT table
BAKERY_DB	DWH	View	ORDERS	A view representing normalized order data, adding primary keys from the PARTNER and PRODUCT tables
BAKERY_DB	MGMT	View	ORDERS_SUMMARY	A view that summarizes orders by the delivery date and baked good type, adding the baked good category

B.10 Ingesting data incrementally (chapter 12)

In chapter 12, we learned that processing all data every time the pipeline executes is inefficient when dealing with large data volumes. We ingested data incrementally, which is faster than full ingestion, as it involves moving less data, resulting in lower storage and compute costs. We utilized Snowflake streams to detect changes in source data and incorporate them into the data pipeline, and we used dynamic tables to maintain data in the data warehouse. The objects used are summarized in table B.10.

Table B.10 Objects used for incremental data ingestion

Database	Schema	Object type	Object name	Description
BAKERY_DB	EXT	Table	JSON_ORDERS_EXT	Data from the external stage is copied into this table.
BAKERY_DB	EXT	Stream	JSON_ORDERS_STREAM	A stream that detects changes in the JSON_ORDERS_EXT table
BAKERY_DB	STG	Table	JSON_ORDERS_TBL_STG	A table that stores the flattened semi-structured data from the extraction layer
BAKERY_DB	STG	Stream	PRODUCT_STREAM	A stream that detects changes in the PRODUCT table
BAKERY_DB	STG	Stream	PARTNER_STREAM	A stream that detects changes in the PARTNER table

Table B.10 Objects used for incremental data ingestion *(continued)*

Database	Schema	Object type	Object name	Description
BAKERY_DB	DWH	Table	PRODUCT_TBL	A table in the data warehouse layer initially populated with data from the staging layer
BAKERY_DB	DWH	Table	PARTNER_TBL	A table in the data warehouse layer initially populated with data from the staging layer
BAKERY_DB	DWH	View	PRODUCT_VALID_TS	A view that calculates the end timestamp of the validity interval
BAKERY_DB	DWH	Dynamic table	ORDERS_TBL	A dynamic table that adds primary keys from the PARTNER and PRODUCT tables to the orders
BAKERY_DB	MGMT	Dynamic table	ORDERS_SUMMARY_TBL	A dynamic table that summarizes data from the ORDERS_TBL table

B.11 Orchestrating data pipelines (chapter 13)

In chapter 13, we scheduled data pipelines so that data engineers don't have to run them manually. We created Snowflake and added task dependencies, allowing the entire pipeline to be scheduled at a specific time and execute all steps required to transform the data in the pipeline in order, one after another. We also sent email notifications from tasks to inform the data engineer if the pipeline execution was completed successfully. The objects used are summarized in table B.11.

Table B.11 Objects used for orchestrating data pipelines and sending notifications

Database	Schema	Object type	Object name	Description
BAKERY_DB	ORCHESTRATION	Task	COPY_ORDERS_TASK	A task that copies data from the external stage into a table
BAKERY_DB	ORCHESTRATION	Task	INSERT_ORDERS_STG_TASK	A task that inserts data from the stream into the staging table
N/A	N/A	Notification integration	PIPELINE_EMAIL_INT	A notification integration that enables sending emails to recipients who verified their email addresses in the same Snowflake account from where the email originates
BAKERY_DB	ORCHESTRATION	Task	PIPELINE_START_TASK	Root task that sends a notification when the pipeline starts executing

Table B.11 Objects used for orchestrating data pipelines and sending notifications *(continued)*

Database	Schema	Object type	Object name	Description
BAKERY_DB	ORCHESTRATION	Task	INSERT_PRODUCT_TASK	A task that inserts the product data from the stream to the target table
BAKERY_DB	ORCHESTRATION	Task	INSERT_PARTNER_TASK	A task that inserts the partner data from the stream to the target table
BAKERY_DB	ORCHESTRATION	Task	PIPELINE_END_TASK	Finalizer task that sends a notification when the pipeline finishes executing
BAKERY_DB	ORCHESTRATION	Table	PIPELINE_LOG	A table that stores logging information

B.12 Testing for data integrity and completeness (chapter 14)

In chapter 14, we incorporated data testing into data pipelines and added data testing steps to the pipeline. We used the Snowflake data metric functions to monitor data quality and added user-defined data metric functions. We created alerts that notified data engineers and business users when data quality metrics exceeded the defined thresholds. Finally, we used the Snowflake Cortex anomaly detection functionality to monitor data ingestion volumes and flag when data volumes deviate from the expected values. The objects used are summarized in table B.12.

Table B.12 Objects used for adding testing steps to the data pipeline, implementing Snowflake data metric functions, and applying the Snowflake machine learning anomaly detection functionality

Database	Schema	Object type	Object name	Description
BAKERY_DB	DQ	Table	DQ_LOG	A table that stores data quality information
BAKERY_DB	ORCHESTRATION	Task	PARTNER_DQ_TASK	A task that checks for null values in the partner rating column
BAKERY_DB	ORCHESTRATION	Task	PRODUCT_DQ_TASK	A task that checks for products whose category is not one of the allowed values: Bread or Pastry
BAKERY_DB	DQ	Data metric function	INVALID_CATEGORY	A function that checks for values in the specified column in the specified table that are not one of the allowed values: Bread or Pastry

Table B.12 Objects used for adding testing steps to the data pipeline, implementing Snowflake data metric functions, and applying the Snowflake machine learning anomaly detection functionality *(continued)*

Database	Schema	Object type	Object name	Description
BAKERY_DB	DQ	Alert	DATA_QUALITY_MONITORING_ALERT	An alert that sends an email when errors are logged in the data quality monitoring table
BAKERY_DB	STG	Table	COUNTRY_MARKET_ORDERS	Randomly generated data representing supermarket orders
BAKERY_DB	DQ	View	ORDERS_HISTORICAL_DATA	View containing historical data on which the anomaly detection model trains
BAKERY_DB	DQ	View	ORDERS_NEW_DATA	View containing new data on which the model detects anomalies
BAKERY_DB	DQ	Model	ORDERS_MODEL	Trained anomaly detection model
BAKERY_DB	DQ	Table	ORDERS_MODEL_ANOMALIES	A table that stores the anomaly detection results based on the new data

B.13 Data pipeline continuous integration (chapter 15)

In chapter 15, we organized the data pipeline code in a Git repository, integrated it with Snowflake, and executed it using the SnowCLI command line interface. The objects used are summarized in table B.13.

Table B.13 Objects used for integrating a GIT repository with Snowflake

Database	Schema	Object type	Object name	Description
BAKERY_DB	GIT_INTEGRATION	Secret	GIT_SECRET	A secret that stores the credentials for authenticating Snowflake with the Git repository
N/A	N/A	API integration	GIT_API_INTEGRATION	An API integration that provides details about how Snowflake interacts with the Git repository API
BAKERY_DB	GIT_INTEGRATION	Git repository	SF_DE	A Git repository stage that exposes the files in the repository to Snowflake

index

A

access control, securing data with row access policies 197–201
access roles, defined 192
  See also specific roles
ACCOUNTADMIN role 19, 31, 40, 43, 77, 146, 160, 184, 191, 216, 252, 292, 305
ACCOUNT_USAGE schema 47
ADDRESS_MASK masking policy 202
ADMIN_DB database 305, 307
AFTER keyword 255, 261
alerts 287–289
ALTER commands 50, 289, 301–302, 306, 314
Amazon S3 40, 90
ANOMALY_DETECTION privilege 292
API (application programming interface) endpoints 326
APIs (application programming interfaces), calling endpoints from Snowpark functions 129–135
  constructing UDFs to retrieve customer reviews 130–132
  interpreting results from UDFs 132
  storing customer reviews in tables 134
architecture, Snowflake 4
ARRAY_AGG() function 278
ARRAY data type 65
AUTOMATIC_CLUSTERING_HISTORY function 160
AUTO_SUSPEND parameter 183
Avro format 65
AZURE properties 42, 87, 93

B

BAKERY_DB database 20, 43, 45, 63, 87, 106, 109, 111, 117, 122–123, 128, 150, 173, 193–194, 214–216, 232, 276, 302, 307, 311
BAKERY_EVENTS event table 77
BAKERY_FULL access role 193–194, 198, 215
BAKERY_READ access role 193–194, 198, 215
BAKERY_WH warehouse 20, 30, 106, 111, 150, 174–178, 185, 193, 195, 232, 242, 276, 307
BEGIN statement 73
BISTRO_INTEGRATION storage integration 43
BISTRO_SAS_STAGE external stage 44–45
BISTRO_STAGE external stage 43, 45, 48
bulk data ingestion 85
business_alias parameter 130–131
business intelligence 15

C

CALL command 73–74, 78–79, 141
CATEGORY column 276, 280
centralized governance approach 197
CI/CD (continuous integration/continuous deployment) pipeline, objects used in examples 332
CLI (command line interface), using Snowflake CLI 308–312
  continuous integration with 311
  executing scripts with 310
  installing and configuring 308

cloud storage
  ingesting files from incrementally 231
  ingesting semistructured data from 60–66
  preparing files in 85–89
clustering, optimizing query performance with 157–162
  adding clustering keys to tables 159
  monitoring clustering process 160
  viewing clustering information 157
  viewing improved query execution after clustering 161
collect() method 116
COLLECTED_ORDERS_FROM_EMAIL table 140–142
compute consumption, monitoring 186
compute resources cost 172
conda channel 105
configuring
  Snowflake CLI 319
  storage integration 40–43
continuous data ingestion
  configuring Snowpipe with cloud messaging 89–97
  preparing files in cloud storage 85–89
continuous delivery 298
continuous integration
  data pipelines 303–308
  summary 315
  using Snowflake CLI 308–312
COPY command 14, 24, 26, 37, 39, 45–49, 51–53, 55, 60, 65, 70, 85, 89, 93–95, 118–119, 233, 235, 321–322
copying data, from shared database 150–152
COPY INTO command 24, 252–253
COPY_ORDERS_TASK task 258, 262, 265
cost control
  optimizing performance with data caching 181–183
  reducing query queuing 183–186
  sizing virtual warehouses 173–181
costs 169–173
  compute resources cost 172
  monitoring compute consumption 186
  total Snowflake cost 171
  virtual warehouse credits 172
COUNTRY_MARKET_ORDERS table 290–291
CREATE DATABASE privilege 146
CREATE GIT REPOSITORY ON SCHEMA privilege 306
CREATE INTEGRATION ON ACCOUNT privilege 306
CREATE INTEGRATION privilege 40, 128
CREATE NETWORK RULE privilege 128
CREATE OR ALTER command 301–302, 310, 315
CREATE OR REPLACE command 302
create_or_replace_view() method 122
CREATE PROCEDURE command 73
CREATE ROLE privilege 191, 193
CREATE SECRET ON SCHEMA privilege 306
CREATE SECRET privilege 128
CREATE STAGE command 50
CREATE STORAGE INTEGRATION command 40
CREATE TABLE command 301
CSV (comma-separated values) files
  ingesting and transforming data from 321
  ingesting data from into Snowflake table 118–121
CURRENT parameters 265
CURRENT_TIMESTAMP() function 25
CURRENT_USER() function 195
CUSTOMER_ORDERS tables 26–27, 65–66, 71–73, 79
CUSTOMER_REVIEWS table 134–135, 137

D

DAC (discretionary access control) 190
DAGS (directed acyclic graphs) 12, 259
data
  pipelines 208–215
  transforming with SQL commands 28
DATA_ANALYST roles 193–194, 196–200, 202, 215, 222, 307
data augmentation
  calling API endpoints from Snowpark functions 129–135
  configuring external network access 126–129
  interpreting order emails using LLMs 138–142
  using LLMs 15
data caching 181–183
  metadata cache 182
  warehouse cache 183
DATA_ENGINEER functional role 193–194, 307
data engineering
  building data pipelines 13
  data engineer responsibilities 6–13
  performing data transformations 9
  presenting data to downstream consumers 10
data frames 104, 115–118
  transforming data with 122–123

data governance 11
  and access control 189
  masking policies 201–203
  RBAC 190–197
  securing data with row access policies 197–201
data ingestion
  continuous 83
  continuous, and dynamic tables 324
  continuous vs. bulk 85
  incremental 228–230, 329
data integrity and completeness
  alerting users when data metrics exceed thresholds 287–289
  applying data metric functions 282–286
data integrity and completeness, testing for 331
  data testing methods 273–275
  incorporating data testing steps in pipeline 275–282
data lake, defined 14
data mart, defined 14
data mesh, defined 15
data metric functions 282–286
  system-defined 283
  user-defined 284
  viewing details 286
data modeling 11
DataOps 13
data pipelines 13, 17, 207
  building robust 78–81
  building sample 216–223
  connecting to Snowflake securely 313–315
  continuous integration 297, 299, 303–312, 315
  DCM (database change management) 300–303
  designing 328
  loading data from staged file into target table, overview 22–27
  orchestrating 248, 250–257, 264–269, 330
  staging CSV files 20–21
  transforming data with SQL commands 28
DATA_QUALITY_MONITORING_ALERT 288
data science 15
data transformations, tools for 9
data validation, defined 10
data volume anomalies 289–295
  displaying data as line chart in Snowsight 292
  generating random data 290–291
  working with anomaly detection model 292–295
data warehouses
  defined 14
  implementing layers 220–222
  layer 212
  organizing layers 213
date dimension, generating in Snowpark Python 113–118
  working with data frames 115–118
date module 114
datetime package 113–114
DCM (database change management) 300–303, 315
  declarative approach 300
  imperative approach 300
  organizing code in repository 302
DEBUG log level 77
decentralized governance approach 197
declarative approach 300
DELETE operation 238
DELIVERY_ORDERS schema 87
DELIVERY_TS timestamp column 293–294
DEPARTMENT column 199–200
DESCRIBE INTEGRATION command 87, 93
DESCRIBE STORAGE INTEGRATION command 41, 61
DETECT_ANOMALIES method 294
development environment 299
DevOps 298
DG schema 197–199, 202
DIM_DATE table 117–118, 122
directory tables 50
DISTINCT keyword 151
DML (data manipulation language) 230
  commands 236
  operations 53
DQ_LOG table 277–279, 281
DQ schema 277, 287, 292–294
DWH schema 214–215, 287, 307
dynamic tables 99
  continuous data ingestion and 324
  maintaining data with 242–246
  transforming data with 98–100

E

EGRESS mode 128
ELT (extract-load-transform) 211, 275
email notifications 257
  summarizing logging information in 266–268
EMPLOYEE
  table 195–197, 201
  view 196, 198, 200, 202
END statement 73
ERROR log level 77
ETag (entity tag) 47
ETL (extract-transform-load) 210, 274

ETLT (extract, transform, load, and then transform) pattern 211
Event Grid Resource Provider, enabling 91
event grid subscription, creating 91
EXCEPTION block 75
exception handling, implementing in stored procedures 75
EXECUTE ALERT privilege 287
EXECUTE IMMEDIATE FROM command 307, 310–312, 315
EXECUTE TASK
  command 253–255, 262
  privilege 31, 252
EXECUTING state 254
external access integration 126
EXTERNAL_JSON_ORDERS schema 63
external network access 125–129
EXTERNAL_ORDERS schema 43, 45
external stages 21
  avoiding duplication when loading data from staged files 47
  configuring storage integration 40–43
  creating 39–49, 63, 87–89
  creating using credentials 44
  creating using storage integration 43
  loading data from staged files into staging table 45–47
  named file format 48
  querying data in with 54
external tables, building pipelines with 53–56
extracting data 208
extraction layer 212
  implementing 216
EXT schema 214–216, 303, 307

F

FAILED state 31
FATAL log level 77
FETCH keyword 306
file_format option 49
FILE_FORMAT parameter 63, 87
files
  ingesting from cloud storage incrementally 231
  preparing in cloud storage 85–89
finalizer task 259
FLATTEN
  function 68, 96
  keyword 133
flattening
  semi-structured data 323
  semi-structured data into relational tables 66–70
FORCE option 47
free trial account, signing up for 318–319
functional roles 192
  See also specific roles

G

GenAI (generative artificial intelligence) 15
GENERATOR() table function 290
GENERIC_STRING type 129
geographical data analysis 148–153
  copying data from shared database 150–152
  Snowflake geography functions 148–150
  viewing query execution parameters using query profile 152
geography functions 148–150
get() method 131
GET_CUSTOMER_REVIEWS UDF 130, 138
GET_ORDER_INFO_FROM_EMAIL handler function 139
Git
  configuring Snowflake to use 303–308
  creating Git repository stage 304–306
  executing commands from Git repository stage 307
GIT_API_INTEGRATION API 305
GIT_INTEGRATION schema 305, 307
GIT_SECRET secret 305
Google Cloud Storage 40, 90
governance
  access control and 327
  data governance features 6
GOVERNANCE_VIEWER database role 166
group by clause 28

H

HARMONIZED_RETAILER_SALES table 150
holidays 115–116
  library 105
  package 106–107, 113–114
HOME_ADDRESS column 201–202
HOST_PORT type 128
hybrid governance approach 198

I

IDE (integrated development environment) 110, 308, 319
IDENTIFIER() function 195
imperative approach 300
IMPORT SHARE privilege 146

incremental data ingestion 9, 224, 329
  comparing approaches 226–227
  detecting changes with streams 230
  ingesting files from cloud storage 231
  maintaining data with dynamic tables 242–246
  preserving history when ingesting data incrementally 236
  preserving history with slowly changing dimensions 228–230
INFER_SCHEMA function 120
INFO log level 77
INFORMATION_SCHEMA 31, 47
ingesting data 321
  from cloud storage provider 322
  from CSV file into Snowflake table 118–121
  semi-structured 323
INGRESS mode 128
IN predicate 165
INSERT
  command 14, 252
  operation 238
INSERT SQL statement 26
INSERT tasks 254, 257–258, 260–261, 266, 275–276, 278–279, 281
internal stage 21
IS_ANOMALY column 295
IS_HOLIDAY function 105–107, 114
is_holiday handler 106
ITEMS column 89

J

join() method 122
JSON (JavaScript Object Notation)
  examining structure 63
  flattening structure to relational format 96
  format 23
  ingesting data into VARIANT data type 65
  transformed data 218
json library 112
JSON_ORDERS_STAGE external stage 216–217, 303
JSON_ORDERS_STREAM 231, 233, 254, 257, 303
JSON_ORDERS_TBL_STG staging table 231, 234, 236, 242, 254, 310–311
json package 113

K

Kafka 100
key-pair authentication, configuring 313

L

LANGUAGE SQL clause 73
LATERAL FLATTEN syntax 67–68
LATERAL modifier 68, 96
layers
  data transformation 212–213
  data warehouse 213
LEAD() function 230
LIMIT keyword 151
line charts, displaying data as in Snowsight 292
LIST command 21, 25, 51–52, 88, 217, 233
LLMs (large language models) 15, 125, 321
  calling API endpoints from Snowpark functions 129–135
  deriving customer review sentiments 135–137
  functions 326
  interpreting order emails using 138–142
LOAD_CUSTOMER stored procedures 72–73, 75, 77, 79
LOADED load status 48
LOAD FAILED load status 48
LOAD_HISTORY view 47
local development environment
  creating Snowflake session 110
  installing and configuring 110
  providing credentials in configuration file 111
  querying data and executing SQL commands 113
  using SQL API from 110–113
LOG_CUR cursor 266
logging
  adding logging functionality to tasks 264
  summarizing logging information in email notification 266–268
LS command 306

M

managed access schema 192
MANAGE GRANTS privilege 191–193, 215
Marketplace, Snowflake 6
  getting data from Snowflake Marketplace 145–148
masking policies 201–203, 327
materialized views 53
  using to improve query performance 55
MAX_CONCURRENCY_LEVEL parameter 185
MERGE INTO keywords 26
metadata 47
metadata cache 181–182
MFA (multifactor authentication) 313, 315

MGMT schema 214–215, 246, 307
micro-batch 85
micro-partitions 145, 154–157
  conceptual example of 154
  pruning 156
Microsoft Azure
  blob storage 90
  configuring event grid messages for blob storage events 90–92
  creating notification integration 92
  creating pipe object 93–95
  flattening JSON structure to relational format 96
  ingesting data continuously 95
ML (machine learning) 289
ML schema 293
MODIFY LOG LEVEL ON ACCOUNT privilege 77
MONITOR USAGE privilege 160
multicluster warehouses 173
multitenant architecture 197
my_session session object 111

N

named file format 48
no_days variable 115
notification integration, creating 92
NVL() function 230

O

OAuth 313
OBJECT data type 65
ON clause 26
OR ALTER clause 301
ORC format 65
orchestrating data pipelines 248, 330
  email notifications 257
  monitoring data pipeline execution 264–268
  task graphs 258–264
  troubleshooting data pipeline failures 269
  with Snowflake tasks 250–257
orchestration, defined 11–12
ORCHESTRATION schema 264, 277, 303, 307
ORDER_DATETIME column 89
orderfiles container 40, 43
ORDER_ID column 89
orderjsonfiles container 63
ORDERS_BISTRO_STG staging table 45, 71
Orders by day key 68
ORDERS_COMBINED_STG view 71
ORDERS_CSV_FORMAT file format 48–49, 54
ORDERS_DF data frame 140
ORDERS_HISTORICAL_DATA view 293
ORDERS_HOLIDAY_FLG view 122–123
ORDERS_MODEL anomaly detection model 293–294
ORDERS_PARK_INN_RAW_STG raw staging table 69
ORDERS schema 20
ORDERS_STAGE internal stage 25
ORDERS_STG staging table 45, 71, 121
ORDERS_SUMMARY_TBL dynamic table 246
ORDERS table 220–221
ORDERS_TBL dynamic table 242
ORGADMIN role 105
ORGANIZATION_USAGE views 172
ORIGIN parameter 305
OTHER keyword 75

P

PARK_INN_INTEGRATION storage integration 63
PARK_INN_STAGE external stage 63
Parquet format 23, 65
PARTIALLY LOADED load status 48
Partner Connect, Snowflake 6
partner data quality task 277–279
PARTNER_ID primary key column 220
PARTNER view 221
PIPELINE_EMAIL_INT notification integration 257
PIPELINE_END_TASK 259, 262, 275, 303
PIPELINE_LOG table 264, 266, 268, 277
pipelines
  avoiding delays in execution due to testing 274
  building 13
  building sample data pipeline 216–223
  building with external tables 53–56
  comparing pattern 209
    ELT 211
    ETL 210
    ETLT 211
  creating first data pipeline 19
  data 17, 19, 29–33, 207–215
  designing idempotent data pipelines 230
  flagging data that failed data tests 274
  incorporating data testing steps in 275–282
  notifying recipients about test results 274
  performing data testing as steps in 273
  performing data testing independently of 275
  stopping execution 274

PIPELINE_START_TASK 259, 261–262, 275, 281, 303
pipe object, creating 93–95
preparing data files for efficient ingestion 51–52
  file sizing recommendations 51
  organizing data by path 52
presentation layer 213
procedures, creating in worksheets 105–110
PROCESS_ORDERS task 30
PROC_IS_HOLIDAY procedure 107–109
product data quality task 280
PRODUCT_DQ_TASK task 275, 278, 280–281
PRODUCT_ID primary key column 220
production environment 299
PRODUCT staging table 238
prompt engineering 139
providers, ingesting data from 5, 322
pruning 155
public API endpoints 127
PURGE option 25–26, 46
Python
  creating procedures in worksheets 105–110
  generating date dimension in Snowpark Python 113–118

Q

QUALIFY clause 166
Quantity key 68
QUANTITY target column 293–294
QUERY_HISTORY view 166
query performance optimization 144, 165–167
  identifying queries that are candidates for optimization 166
  micro-partitions 154–157
  optimizing storage with clustering 157–162
  optimizing to reduce spilling 179–181
  search optimization 162–165
  writing efficient SQL queries 165
query profile 152
query queuing 183–186
  examining 184
  limiting concurrently running queries 185
query results
  persisted 176
query results cache 181
Quick Python Book, The (Manning) 104

R

RANDOM() function 290
RAP_BUSINESS_UNIT row access policy 199
RATING column 276–277, 279
RAW schema 195–196
RBAC (role-based access control) 189–197, 277, 307, 327
  custom roles 191
  designing 192–197
  system-defined roles 191
READ_EMAIL_PROC stored procedure 138, 141
READ privilege 129
REFRESH command 94
regular expressions 134
regular schema 191
relational tables, flattening semistructured data into 66–70
REMOVE command 47
reporting layer, implementing 222
repositories, organizing code in 302
requests Python package 130–131
resource monitors 187
REST endpoints 90
RESULT_SCAN() table function 295
RETAIL_ANALYSIS schema 150, 173
RETAILER_SALES table 150–151, 173–174
RETURNS clause 73
REVIEW_SENTIMENT UDF 138
REVIEWS schema 128
root task 259
row access policies 197–201, 327
ROW_NUMBER() function 166
RPT schema 194, 196–198, 200, 202

S

SAS (shared access signature) token 44
SCD (slowly changing dimensions) 228–230
  append-only strategy 229
  designing idempotent data pipelines 230
  type 2 228
SCHEDULED state 254
schemas, creating with access control 215
search optimization 162–165
  adding to tables 164
  reviewing query performance after adding 164
security 10–11
SECURITYADMIN role 191, 193–194, 196, 215
SELECT command 23, 54–55, 63, 88, 131, 233, 237
semi-structured data 65
  ingesting from cloud storage 60–66
    creating external stage 63
    creating storage integration 61
    examining JSON structure 63
    ingesting JSON data into VARIANT data type 65

sentiment analysis 135–137
session object 140
SF_DE Git repository stage 305
SHOW commands 116, 164, 269, 306
SKIPPED state 257
snow command 308–309, 314, 319–320
Snowflake
  configuring environment 317
  connecting to securely 313–315
  creating session 110
  data engineering with 3–6, 14–15
  objects used in examples 327, 332
  signing up for free trial account 318–319
Snowflake CLI (command-line interface)
  continuous integration with 311
  executing scripts with 310
  --help option 308, 319
  installing and configuring 308, 319
SNOWFLAKE database 47, 166, 293
Snowflake extension for Visual Studio Code 310
Snowflake Marketplace, getting data from 145–148
Snowflake objects 321
  continuous data ingestion and dynamic tables 324
  designing data pipelines 328
  ingesting data from cloud storage provider 322
  optimizing query performance and controlling cost 327
:: Snowflake operator 67
snowflake.snowpark package 110, 113, 139
snowflake-snowpark-python package 110, 138
snowflake.snowpark.types package 113
Snowpark 6, 102
  creating procedures in worksheets 105–110
  executing code natively using SQL API from local development environment 110–113
  generating date dimension in Snowpark Python 113–118
  ingesting data from CSV file into Snowflake table 118–121
  overview 104
  transforming data with data frames 122–123
snowpark library 139
SNOWPARK schema 106, 109, 111, 117, 122–123
Snowpipe 5
  configuring with cloud messaging 89–97
Snowsight 19
  displaying data as line chart in 292
  user interface 322
software engineering 12
SPEEDY_INTEGRATION storage integration 87
SPEEDY_ORDERS_RAW_STG staging table 89
SPEEDY_PIPE pipe object 93
SPEEDY_QUEUE_INTEGRATION notification integration 93
SPEEDY_STAGE external stage 87
spilling
  defined 179
  optimizing query performance to reduce 179–181
SQL API, using from local development environment 110–113
  creating Snowflake session 110
  installing and configuring local development environment 110
  providing credentials in configuration file 111
  querying data and executing SQL commands 113
SQL SELECT command 46
SQL (Structured Query Language)
  transforming data with 28
  writing efficient queries 165
SSO (single sign-on) 313, 315
staged files
  loading data from into staging table 24–26
  loading data from into target table 22–27
  merging data from into target table 26–27
stages, Snowflake 21
staging CSV files 20–21
staging data 37
  building pipelines with external tables 54–56
  directory tables 50
  external stages 39–49
  preparing data files for efficient ingestion 51–52
staging layer 212
  implementing 218–220
staging tables
  loading data from staged files into 24–26, 45–47
  merging data from into target table 26–27
START_TIME column 184
ST_DISTANCE function 148–150
STG layer 222
STG schema 214–215, 218–219, 234, 276, 290, 307, 310
storage account names 40
Storage Blob Data roles 42, 46
storage integration
  configuring 40–43
  creating 61, 86
  creating external stages using 43
  object 38
  properties 42

storage optimization, optimizing query performance with clustering 157–162
  adding clustering keys to tables 159
  monitoring clustering process 160
  viewing clustering information 157
  viewing improved query execution after clustering 161
storage queue, creating 91
stored procedures
  adding logging to 76–78
  creating 73
  creating to interpret customer emails 138
  encapsulating transformations with 70–75
  implementing exception handling in 75
  including return values in 74
  See also specific stored procedures
STORE_LOC_GEO column 150
Streaming Data Pipelines with Kafka (book) 100
streams 9
  detecting changes with 230
String return type 106
STRIP_OUTER_ARRAY keyword 70
StructType construct 140
SUCCEEDED state 31, 254
SUMMARY_ORDERS table 78–79
system-defined
  data metric functions 283
  roles 191

T

table()
  method 122
  return type 138
tables
  saving CSV result to 140
  storing customer reviews in 134
TARGET_LAG parameter 98, 242
target tables
  loading data from staged file into 22–27
  merging data from staging table into 26–27
task graphs 248, 259–264
  creating finalizer task 262
  creating root task 261
  designing 259
  viewing 263
TASK_HISTORY() function 31–33, 254, 256, 266–267, 269, 276, 281
tasks 29–33
test environment 299
testing, data integrity and completeness 272, 331
  alerting users when data metrics exceed thresholds 287–289
  applying data metric functions 282–286
  data testing methods 273–275
  incorporating data testing steps in pipeline 275–282
testing data volume anomalies 289–295
  displaying data as line chart in Snowsight 292
  generating random data 290–291
  working with anomaly detection model 292–295
timedelta module 114
TIMESTAMP data type 293
time travel 5
TO_GEOGRAPHY function 148–149
total Snowflake cost 171
transforming data 58, 321
  adding logging to stored procedures 76–78
  building robust data pipelines 78–81
  encapsulating transformations with stored procedures 70–75
  flattening semistructured data into relational tables 58–70
  ingesting semistructured data from cloud storage 60–66
  with dynamic tables 98–100
TRANSFORM schema 71, 77
troubleshooting data pipeline failures 269
TRUNCATE command 79

U

UAT (user acceptance test environment) 299
UDFs (user-defined functions) 125
  constructing to retrieve customer reviews 130–132
  interpreting results from 132
  storing customer reviews in tables 134
UNIFORM() function 290
UPDATE operation 238
USAGE privilege 129
USAGE_VIEWER database role 184
USE_CACHED_RESULT parameter 177, 181
user acceptance test environment 299
USERADMIN role 191, 193, 196, 198
user-defined data metric functions 284
USING keyword 26
UUID (universally unique identifier) 265

V

VALUE column 54, 68
VARIANT data type 54, 67–68, 70, 323
  ingesting JSON data into 65
virtual warehouses, sizing 173–181
  comparing query statistics between differently sized warehouses 177–179
  optimizing query performance to reduce spilling 179–181
  persisted query results 176

W

warehouse cache 183
WAREHOUSE_LOAD_HISTORY table 184
WAREHOUSE parameter 98
WARN log level 77
WHEN clause 75
WHEN MATCHED THEN clause 26
WHEN NOT MATCHED THEN clause 27
WHEN SYSTEM$STREAM_HAS_DATA conditions 261
worksheets, creating procedures in 105–110
  executing commands from 19

X

XML format 65

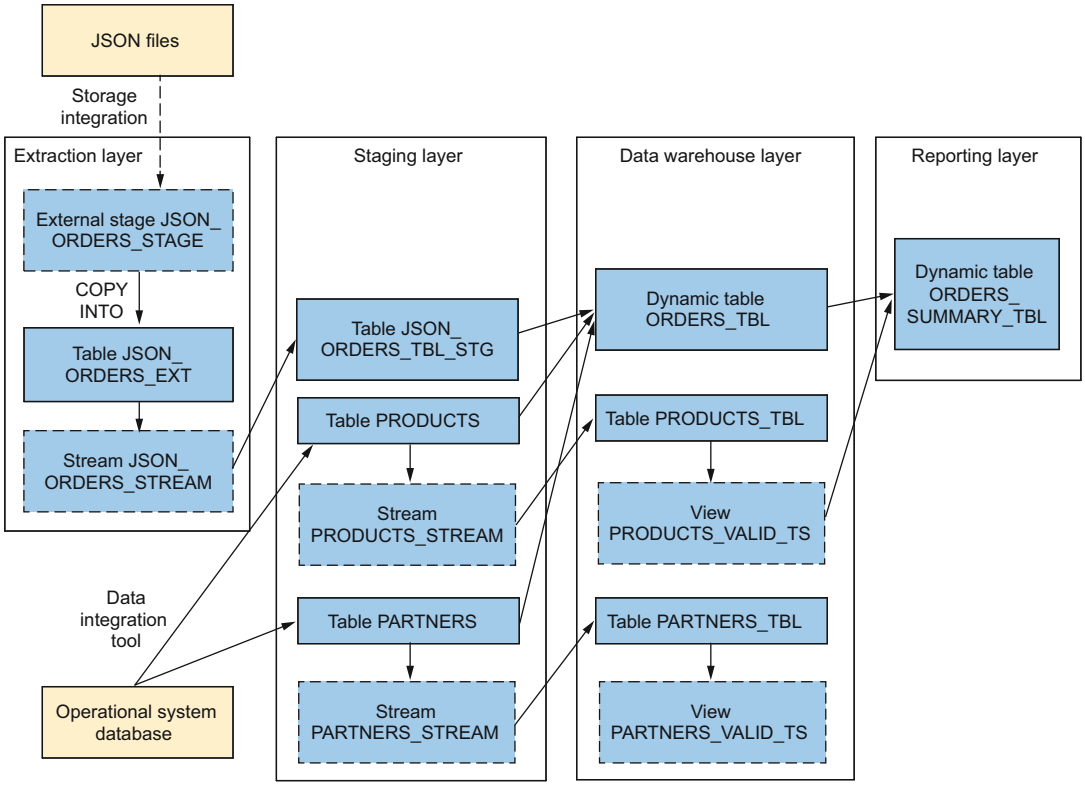
Y

YELP_API_INTEGRATION external access integration 129
YELP_API_NETWORK_RULE network rule 128–129
YELP_API_TOKEN secret 129

Z

zero-copy cloning 4–5

Comprehensive data pipeline we build in the book



In this book, we build a comprehensive data pipeline for a fictional bakery. The pipeline ingests order information from the bakery's customers, stored as JSON files in cloud storage. The data from these files is copied into Snowflake tables in the extraction layer. Then, the JSON data is flattened into a relational structure in the staging layer. A simulation of the bakery's core business data, containing information about business partners and baked goods, is also ingested into the staging layer. The data from the staging layer is normalized in the data warehouse layer. Finally, the data is summarized and presented in the reporting layer.

OPERATIONS & CLOUD

Snowflake DATA ENGINEERING

Maja Ferle • Foreword by Joe Reis

Pipelines that ingest and transform raw data are the lifeblood of business analytics, and data engineers rely on Snowflake to help them deliver those pipelines efficiently. Snowflake is a full-service cloud-based platform that handles everything from near-infinite storage, fast elastic compute services, inbuilt AI/ML capabilities like vector search, text-to-SQL, code generation, and more. This book gives you what you need to create effective data pipelines on the Snowflake platform.

Snowflake Data Engineering guides you skill-by-skill through accomplishing on-the-job data engineering tasks using Snowflake. You'll start by building your first simple pipeline and then expand it by adding increasingly powerful features, including data governance and security, adding CI/CD into your pipelines, and even augmenting data with generative AI. You'll be amazed how far you can go in just a few short chapters!

What's Inside

- Ingest data from the cloud, APIs, or Snowflake Marketplace
- Orchestrate data pipelines with streams and tasks
- Optimize performance and cost

For software developers and data analysts. Readers should know the basics of SQL and the Cloud.

Maja Ferle is a Snowflake Subject Matter Expert and a Snowflake Data Superhero who holds the SnowPro Advanced Data Engineer and the SnowPro Advanced Data Analyst certifications.

The technical editor on this book was Daan Bakboord.

For print book owners, all digital formats are free:
https://www.manning.com/freebook

MANNING

"An incredible guide for going from zero to production with Snowflake."
—Doyle Turner, Microsoft

"A must-have if you're looking to excel in the field of data engineering."
—Isabella Renzetti, Data Analytics Consultant & Trainer

"Masterful! Unlocks the true potential of Snowflake for modern data engineers."
—Shankar Narayanan, Microsoft

"Valuable insights will enhance your data engineering skills and lead to cost-effective solutions. A must read!"
—Frédéric L'Anglais, Maxa

"Comprehensive, up-to-date and packed with real-life code examples."
— Albert Nogués, Danone



ISBN-13: 978-1-63343-685-5

