› Hands-on preparation and practice
› Practical skills advancement for practitioners
› Prescriptive guidance from expert voices

CERTIFICATION STUDY COMPANION SERIES

SnowPro™ Core Certification Companion

Hands-on Preparation and Practice

›
Maja Ferle

Apress®

Certification Study Companion Series

The Apress Certification Study Companion Series offers guidance and hands-on practice to support technical and business professionals who are studying for an exam in the pursuit of an industry certification. Professionals worldwide seek to achieve certifications in order to advance in a career role, reinforce knowledge in a specific discipline, or to apply for or change jobs. This series focuses on the most widely taken certification exams in a given field. It is designed to be user friendly, tracking to topics as they appear in a given exam. Authors for this series are experts and instructors who not only possess a deep understanding of the content, but also have experience teaching the key concepts that support readers in the practical application of the skills learned in their day-to-day roles.

More information about this series at https://link.springer.com/bookseries/17100

SnowPro™ Core Certification Companion

Hands-on Preparation and Practice

Maja Ferle

Apress®

SnowPro™ Core Certification Companion: Hands-on Preparation and Practice

Maja Ferle
Ljubljana, Slovenia

ISBN-13 (pbk): 978-1-4842-9077-4 ISBN-13 (electronic): 978-1-4842-9078-1
https://doi.org/10.1007/978-1-4842-9078-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Editorial Assistant: Shaul Elson
Copy Editor: Kezia Endsley

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

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 Core Certification and is a Snowflake Subject Matter Expert and a Snowflake Data Superhero.

About the Technical Reviewer

Nadir Doctor is a database and data warehousing architect, and a DBA, who has worked in various industries with multiple OLTP and OLAP technologies. He has also worked on primary data platforms, including Snowflake, Databricks, CockroachDB, DataStax, Cassandra, ScyllaDB, Redis, MS SQL Server, Oracle, Db2 Cloud, AWS, Azure, and GCP. His major focus is health-check scripting for security, high availability, performance optimization, cost reduction, and operational excellence. He has presented at several technical conference events, is active in user group participation, and can be reached on LinkedIn.

Thank you to Maja and all the staff at Springer. I'm grateful for the immense support of my loving wife, children, and family during the technical review of this book. I hope that you all find the content enjoyable, inspiring, and useful.

Acknowledgments

When I was getting ready for my own Snowflake SnowPro Core Certification a few years ago, study materials were scarce. I decided to create notes that I thought would come in handy some day for others to use, but I never got around to forming my scribbles into a guide that I could share.

Then Jonathan Gennick approached me about writing this study companion. I knew immediately that it was the incentive I needed to rewrite my notes. I'm grateful to Jonathan for getting me excited about this project and for his guidance, support, and invaluable feedback as I started writing the initial chapters.

Special thanks to Sowmya Thodur for her conscientious project management, always responding to me promptly when I needed help and ensuring that the writing process progressed smoothly.

Thanks also to my technical reviewer Nadir Doctor for suggesting improvements and pointing out details that I hadn't explained clearly enough.

Finally, I'm pleased that the team at Apress took care of every detail related to book production so that I only had to focus on the writing.

Introduction

These are exciting times for anyone working with cloud technologies. The Snowflake Data Platform is one of these cloud technologies. Because it was natively designed for the cloud, Snowflake offers some unique features and capabilities.

Along with an increasing interest in Snowflake across the globe, the demand for qualified Snowflake professionals is rising. In a sea of candidates, a Snowflake certification will distinguish you from the crowd. You can take the first step by achieving the SnowPro Core Certification.

The SnowPro Core Certification exam will validate your knowledge to apply core expertise in implementing and migrating to Snowflake. You will have a thorough understanding of the Snowflake Cloud Data platform and will have the knowledge necessary to develop and manage secure, scalable Snowflake solutions.

This study companion guides you in your journey to preparing for the SnowPro Core Certification exam. Even if you are not looking to take the exam yet, you can use this study companion to learn Snowflake essentials to the same level of completeness as someone who has taken the exam.

The structure of this study companion follows the topics that are listed in the SnowPro Core Exam Study Guide. Each chapter explains the relevant topics along with code samples and guided steps through the user interface that allow you to practice using the functionality.

Chapter 1 provides overview information about the exam and the process to register for the exam. It also guides you in setting up a free trial Snowflake account so that you can practice what you have learned.

Chapter 2 introduces Snowflake Data Platform core features, including Snowsight, the web user interface that you use to work with Snowflake.

Chapters 3-6 delve further into the Snowflake architecture layers, including storage and performance concepts, virtual warehouses, and access and security considerations.

Chapters 7-11 focus on working with data, starting with data loading and unloading, followed by data transformation, data protection, and data sharing, and ending with SQL scripting and Snowpark.

Great care was taken to ensure that the information presented is up to date. Considering that Snowflake is constantly evolving, it's possible that some of the newer Snowflake features have been updated since the study companion was written. You are encouraged to review the Snowflake documentation after you complete each chapter to check that you have the latest information. You can also reinforce your learning by working through the examples presented in each chapter using your trial Snowflake account.

CHAPTER 1

Exam Overview

You are starting your preparation for the SnowPro Core Certification. This certification validates your knowledge to migrate to and implement the Snowflake Data Cloud.

This chapter sets the direction for getting ready for the SnowPro Core exam. It outlines the resources that will aid you in your learning strategy. It explains how you can obtain access to a trial Snowflake account, which allows you to practice what you have learned. The chapter also provides links to useful additional study materials and describes how to sign up for the exam.

Exam Content

The SnowPro Core Certification is designed for individuals who want to validate their knowledge of Snowflake. You are expected to have a thorough understanding of the following subject areas:

- Data loading and transformation in Snowflake
- Virtual warehouse performance and concurrency
- DDL and DML queries
- Using semi-structured and unstructured data
- Cloning and time travel
- Data sharing
- Snowflake account structure and management

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_1

The exam does not cover cloud fundamentals or basics of SQL syntax, but some questions on the exam assume knowledge of these concepts. Some of the broad knowledge areas that you are expected to be familiar with include:

- Basic concepts about databases, database terminology, and SQL, including selecting and manipulating data
- Familiarity with database objects, such as tables, data types, views, stored procedures, and functions
- Security topics, such as the difference between authentication and authorization
- Overview knowledge about cloud computing, cloud services, and cloud architectures

Exam Subject Areas

The main subject areas that are covered on the exam and their weighting ranges are listed in Table 1-1.

Table 1-1. *Subject Areas and Their Weighting Ranges*

Domain	Estimated Weighting Range
Snowflake Cloud Data Platform Features and Architecture	20 – 25%
Account Access and Security	20 – 25%
Performance Concepts	10 – 15%
Data Loading and Unloading	5 – 10%
Data Transformations	20 – 25%
Data Protection and Data Sharing	5 – 10%

The weighting ranges indicate the percentage of questions on the exam that cover each domain. According to the Snowflake Exam Guide, the domains do not represent a comprehensive listing of all the content that will be included in the exam. You are expected to learn all topics according to the exam study guide and any included in this study companion.

Exam Format

The exam consists of 100 questions, all of which are in one of the following formats:

- **Multiple choice.** Select the most appropriate answer.
- **Multiple select.** Select all answers that apply. The question will tell you how many answers are to be selected.

The time limit for completing all questions on the exam is 115 minutes. As long as you come well prepared to take the exam, you should be able to complete all questions in the allotted time. Some questions on the exam may be unscored items to gather statistical information. These items are not identified to you and do not affect your score.

The passing score for the exam is 750 in a range of 0-1000.

For the latest information about the exam, navigate to the Snowflake Certifications page at the following URL:

`www.snowflake.com/certifications/`

Supplementary Study Materials

To be well prepared for this exam, you will utilize this study companion, as well as other materials, including hands-on experience, on-demand training courses, the Snowflake documentation, and other self-study assets.

While this study companion provides enough information to help you prepare for all topics that are covered on the exam, you may decide to supplement your learning materials with additional free self-study resources:

- You may enroll in the Snowflake learning platform at https://learn.snowflake.com/ and review the learning tracks.
- You should also sign up for a 30-day free trial Snowflake account.

In addition to giving you a boost in preparing for the SnowPro Core exam, some of the workshops in the Snowflake learning platform award badges that you may share on social media so that you can showcase your Snowflake learning progress even before you take the exam.

An important aspect of preparing for the exam is hands-on experience with Snowflake. Particularly if you are not already working with Snowflake in your day job, you should become familiar with the user interface and have a working knowledge of its basic functionality.

To get the hands-on experience that is needed in preparing for the SnowPro Core Certification, sign up for a 30-day free Snowflake trial account. The details of the sign-up process are described in the next section.

Signing Up for a Free Trial

Snowflake offers a 30-day free trial account that enables you to work with Snowflake's full range of features. You don't have to provide any payment information when you sign up and there are no contractual obligations. All you need to sign up for a trial account is a valid email address.

Sign-up Process

To start your 30-day free Snowflake trial, navigate to this URL:

`https://signup.snowflake.com/`

You will be asked to provide your first name, last name, email address, the name of your company, your job role, and the country from where you will be accessing your Snowflake account.

After you enter this information and click the Continue button, a screen like the one in Figure 1-1 will appear.

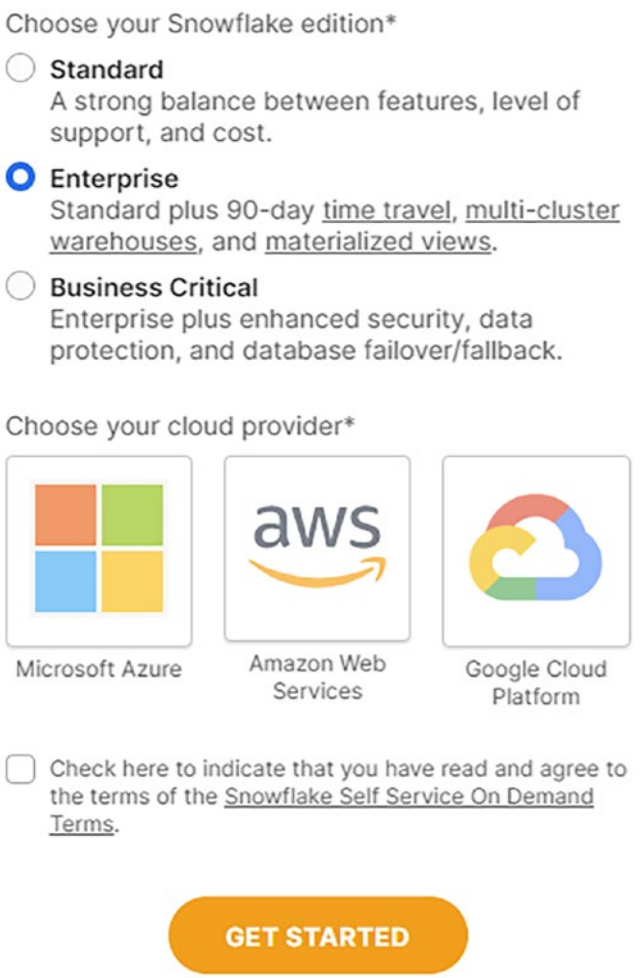

Figure 1-1. *Sign up for a 30-day free Snowflake trial*

You will be asked to choose your Snowflake edition from these options: Standard, Enterprise, or Business Critical. For the purposes of studying for the SnowPro Core Certification exam, the Standard edition is sufficient, although a few features that are included in the Enterprise edition, such as multi-cluster warehouses and materialized views, are also covered on the exam. Since this is a free trial account, you don't have to worry about the cost; you can choose Enterprise to have the full range of features.

You must also choose a cloud provider. Just pick one; it doesn't matter which one because the core Snowflake functionality is the same in all cloud providers.

After you choose a cloud provider, you will be asked to choose the region of the cloud provider. It is best to choose the region that is closest to you geographically. Then click the checkbox that asks you to agree to the terms and click the Get Started button.

After you complete the previous steps, you will receive an email with an activation link and your Snowflake account URL. Bookmark this URL for future use, as this will become your main point for accessing Snowflake. After activation, you will create a username and password that you will use to log in to your Snowflake account.

Along with your activation email, you will also receive a Welcome to Snowflake email with useful information, such as links to learning resources and the Snowflake Community. You are encouraged to join the Snowflake Community, where you can ask questions in case you get stuck during your learning journey.

Trial Expiration

The 30-day free Snowflake trial account expires 30 days after you sign up or when you have used all the free credits included in the account, whichever occurs first. But don't worry, because it's unlikely that you will use all your free credits if you only follow the exercises required to prepare for the exam.

At the end of the trial period, the account is suspended. You will be able to log in to a suspended account, but only to enter your credit card information, which will allow you to continue to use it as a paid account. Without credit card information, you will no longer be able to use any features in your free trial account.

While you are learning, you don't have to switch to a paid account. You may sign up for a new free trial account again, even using the same email address. Just keep in mind that any work that you do in your previous free trial account will not be accessible in your new free trial account, so be sure to save any SQL commands or scripts you write to local files if you want to save them for later use.

Registering for the Exam

Snowflake's SnowPro Certification exams are delivered through Pearson VUE. The exams can be taken remotely with a virtual proctor or at a testing center. There are more than 1,000 Pearson VUE testing centers located across the globe, so you should be able to find one near you if you want to take the test at a testing center.

During the exam registration process you will be asked to review and accept Snowflake's Certification terms and conditions. You must agree to these terms to be allowed to register for any Snowflake exam.

The exam registration process is described in more detail in the following sections.

Create a Snowflake Certification Portal Account

Navigate to the Snowflake Certification Portal at the following URL:

`https://snowflake.useclarus.com/`

Click the Create New Profile option to create a new account. You will be asked to fill in your personal information. Keep in mind that your first name and last name must match your government-issued identification document that you will present when you take the exam.

Once you complete all the required fields and agree to the Snowflake Certification terms and conditions, click the Create Account button to create your account.

In the Snowflake Certification Portal, your default page is the dashboard where you will find links to download the exam study guides, watch free and paid on-demand training, enroll in one of Snowflake's instructor-led training courses, and review the Snowflake documentation.

Start the Application

To start your registration process, you must first create an application in the Snowflake Certification Portal. To do that, click the Available Certifications option in the left menu. You should see the SnowPro Core Certification in the list of available options, as shown in Figure 1-2.

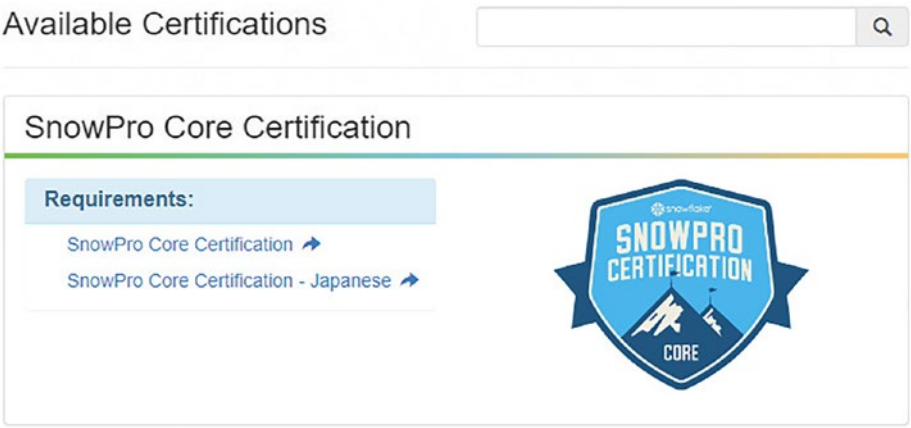

Figure 1-2. *List of available certifications*

Click the SnowPro Core Certification option, which will take you to registration steps for the exam in the English language.

On the next page, enter your contact details and click the Finalize Application button. You will see a window directing you to schedule your exam with Pearson VUE, as shown in Figure 1-3.

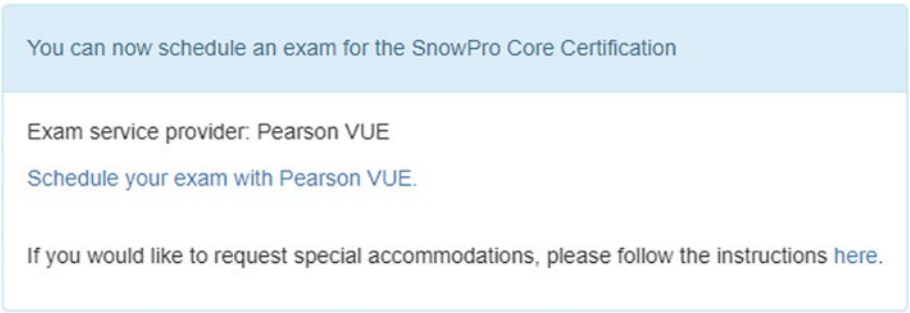

Figure 1-3. *Schedule your exam with Pearson VUE*

Click the "Schedule Your Exam with Pearson VUE" hyperlink. You will see a list of your pre-approved exams for which you have applied on the Snowflake Certification Portal, as shown in Figure 1-4.

Figure 1-4. *Pre-approved exams*

Schedule the Exam

Click the "COF-C02: SnowPro Core Certification Exam - ENGLISH ONLY" hyperlink.

You will be asked whether you want to take the exam at a test center, online at home, or at the office. Select your choice.

Regardless of where you take the exam, you need to present a government-issued identification (ID) before you start your exam.

If you will take your exam online at your home or office, you also need a personal computer that has a reliable webcam and Internet connection and a suitable, distraction-free room or space to take your exam.

Tip If you will be taking the exam online, make sure that your computer has a functioning webcam, which the proctor will use to monitor you during the exam.

The next screen asks you to review and agree to Snowflake's Certification terms and conditions. If you will be taking your exam online, you will also be asked to agree to the online exam policies.

Once you agree, you will see the next screen, where you can select the date and time of your exam. If you will be taking your exam at a test center, you will see a list of test centers, ordered by proximity to the address that you provided while creating your account. You may select one or more test centers to compare availability of timeslots. If you will be taking your exam online, you will be provided with the available timeslots.

Select the date and time when you want to take the exam according to the available timeslots. Finally, you will be directed to check out, where you will pay your exam fee.

Rescheduling and Cancellation Policy

If you will be taking your exam at a testing center, you must reschedule or cancel at least 24 hours in advance before the scheduled start time of your exam. If you reschedule or cancel after that time, your exam fee will not be refunded.

If you will take your exam online, you may reschedule or cancel your online registration without incurring an additional fee any time before the scheduled start time of your exam. If you do not show up for the online exam, your exam fee will not be refunded.

Exam Results and Retake Policy

You are expected to take the exam at the scheduled place and time. After the completion of the exam, you will receive a score report via email that contains information regarding the outcome of the exam.

If you achieve a pass result, your transcript will record the exam as *Pass* and you will also receive your score. Within 72 hours of passing your exam, you will receive an email from Credly. Make sure to accept your digital badge and set up your verifiable certification.

If you do not achieve a passing result, your transcript will record the exam as *Fail* and you will also receive your score with scoring feedback. Don't give up if you don't pass the exam on your first try. Review all the study materials again, taking into consideration any weak areas that you identify after reviewing your scoring feedback, and retake the exam.

If you fail your first attempt at the SnowPro Core Certification exam, you must wait seven calendar days from the date of your last attempt to retake the exam. In a 12-month period, you are allowed to retake the exam no more than four times. You must pay for each exam attempt. There are no free retakes or discounts on retakes.

Recertification

Snowflake is continuously updated so you will need to keep your certification status up to date. All Snowflake exams expire after two years. To stay current, you can sign up for the abbreviated SnowPro Core Recertification Exam and achieve a passing score, which will extend your SnowPro Core Certification status for an additional two years.

An alternative way to keep your SnowPro Core Certification current is to sign up for a SnowPro Advanced Certification exam and pass, which will roll up your SnowPro Core Certification status with your latest SnowPro Advanced Certification's expiration date.

Summary

This chapter covered all the areas that help prepare you for the SnowPro Core Certification exam. It provided an overview of the exam content and the type of questions you will find on the exam. It also explained how to sign up for a free 30-day trial Snowflake account. The free trial account will allow you to gain hands-on experience working with Snowflake as well as to complete the hands-on exercises throughout this study companion.

The next chapter provides an introduction to the Snowflake Cloud Data platform to get you started on your Snowflake learning journey.

CHAPTER 2

Introduction to the Snowflake Cloud Data Platform

Snowflake is the first analytic database natively designed and built for the cloud. It is delivered in the form of Software-as-a-Service (SaaS). Compared to traditional databases, Snowflake is generally faster, easier to use, and more flexible.

This chapter introduces the basic features of the Snowflake Cloud Data platform. It provides an overview of the various Snowflake editions, cloud platforms, and regions where it can be hosted. It also reviews the different ways you can interact with Snowflake, including the web user interface, command-line interface, and connectors.

So that you can use your Snowflake account with the practice examples presented in subsequent chapters, this chapter also reviews a brief initial setup.

Snowflake as a Cloud Service

The main functionalities of Snowflake are data storage, data processing such as loading and unloading, and analytic solutions that can be

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_2

accomplished via SQL queries or supported programming languages. Additionally, Snowflake provides features of traditional databases, including user authentication and authorization, data protection, high availability, and performance optimization. Snowflake also provides certain capabilities that are specific to the cloud:

- It runs completely on cloud infrastructure (with the exception of optional command-line clients, drivers, and connectors) and doesn't require any hardware to maintain
- Users don't have to install or configure any software because Snowflake takes care of software installation and updates as needed
- There is no need for ongoing maintenance and upgrades because it is all handled by Snowflake

Supported Cloud Platforms

Since Snowflake runs completely on cloud infrastructure, a Snowflake account can be hosted on any of the following cloud platforms:

- Amazon Web Services (AWS)
- Microsoft Azure (Azure)
- Google Cloud Platform (GCP)

The cloud platform is chosen at the time of requesting a Snowflake account. On each cloud platform, Snowflake provides one or more regions where the account is provisioned.

Supported Cloud Regions

Each Snowflake account is hosted in a single geographical region, which is chosen at the time of requesting the account. Regions allow the owner of the account to control where the data will be stored geographically. The compute resources will also be provisioned in the chosen region. Snowflake supports all regions that are available in each of the cloud platforms.

A Snowflake account can be located in a particular region to address latency concerns as well as to serve as backup and disaster recovery. Global organizations may decide to create the account in a region that is closest in geographic proximity to their end users.

Snowflake also manages two dedicated government regions that are available to government agencies or commercial organizations. They must comply with specific privacy and security requirements of the U.S. government.

Snowflake Editions

Snowflake offers multiple editions to fit various usage requirements depending on the needs of the organization requesting the account. Each successive edition includes all features of the previous edition and additional features or higher levels of service. When an organization's needs change, they may change the Snowflake edition in the same Snowflake account.

The Snowflake edition determines the unit cost for the credits and data storage that have been consumed. Other factors that impact cost are the region where the Snowflake account is located and the type of account, which may be either of these:

- **On-demand account.** Pricing is usage-based without long-term licensing requirements.
- **Capacity account.** Pricing is discounted based on an up-front capacity commitment.

The following sections describe each of the Snowflake editions in more detail.

Standard Edition

The Standard edition includes all standard features that are offered by Snowflake, without limitation. This edition is suitable for organizations that are looking for a good balance between features, support, and cost.

Enterprise Edition

The Enterprise edition includes all the features that are available in the Standard edition, as well as additional features that fulfill the needs of large organizations. Some of these features are time travel of up to 90 days, periodic rekeying of encrypted data, column-level security, row access policies, object tagging and classification, multi-cluster virtual warehouses, search optimization, materialized views, and user access history.

Business Critical Edition

The Business Critical edition includes all the features that are available in the Enterprise edition, as well as higher levels of data protection to fulfill the needs of organizations with highly sensitive data. For example, organizations can store personal health information (PHI) data that must comply with industry regulations. Additional features include customer-managed encryption keys, private connectivity to the cloud providers, and database failover and failback between Snowflake accounts to enable business continuity and disaster recovery.

Virtual Private Snowflake (VPS)

Virtual Private Snowflake includes all the features that are available in the Business Critical edition. In addition, to provide the highest level of security for highly sensitive data, accounts using this Snowflake edition are hosted in a separate environment that is physically isolated from other Snowflake accounts. This edition is suitable for organizations such as financial institutions that have the highest data protection requirements.

Tip There may be questions on the exam that expect you to know the features associated with each of these editions. Refer to the Snowflake documentation for the latest list of features per edition.

Snowflake Releases

Snowflake ensures an always up-to-date environment while enhancing the features through continuous development and innovation. New Snowflake releases that may include new features, enhancements, and fixes are deployed each week. The deployments happen transparently in the background. Users experience no downtime or service disruptions during this time.

Snowflake releases may also include preview features that have been implemented and tested in Snowflake, but their full usability has not been proven. Preview features are provided primarily for evaluation and testing purposes; they should not be used in production systems or with production data.

The Snowflake Web Interface

When you first log in to Snowflake, you will see the Snowflake web interface, named Snowsight. With the Snowsight web interface, you can perform Snowflake operations, including:

- Create and execute SQL queries
- Load data into tables
- Monitor query performance and view query history
- Create and manage users
- Manage your own user preferences and change your password
- Create and use virtual warehouses
- Create and modify databases and database objects
- Share data with other Snowflake accounts
- Use the Snowflake Marketplace

Note Prior to Snowsight, the Classic Web UI was used to interact with Snowflake. When you search for information about Snowflake features, you may still find images that use the Classic Web UI, which looks different from Snowsight. Don't let that deter you; just keep in mind that the basic Snowflake functionality is similar in both interfaces.

The left navigation bar, as shown in Figure 2-1, is the primary way to move through Snowsight.

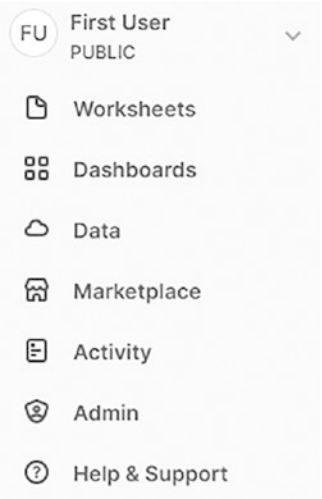

Figure 2-1. *Left navigation bar in Snowsight*

To help you get started using Snowflake quickly, the following sections briefly review the user interface. Most of the functionality available from the menu is covered in more detail in the following chapters.

User Menu

You can access the user menu by clicking the down arrow next to the username at the top of the left navigation bar. This menu contains the options shown in Figure 2-2.

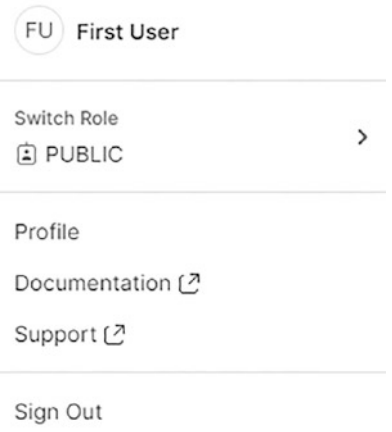

Figure 2-2. *User menu in Snowsight*

You can perform the following actions via the user menu:

- **Switch Role.** Switch the role that you want to use.
- **Profile.** Modify your user preferences or change your password.
- **Documentation.** Open the Snowflake documentation.
- **Support.** Submit and follow up on your support cases. Although this menu option is available, you cannot submit support cases when you are using the 30-day free trial Snowflake account.
- **Sign Out.** Sign out of the Snowflake web interface.

Worksheets

You will spend most of your time writing SQL queries and running commands in worksheets. Not surprisingly, the Worksheets page is the first page in Snowsight. It lets you create, view, and manage your worksheets.

To create a new worksheet, click the +Worksheet button. Each new worksheet that you create is named according to the timestamp when you created it. You can rename the worksheet by clicking the worksheet name at the top left of the screen, next to the Home icon (the gray house), and typing a name. For example, you can name it **First worksheet,** as shown in Figure 2-3.

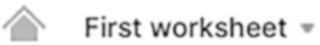

Figure 2-3. *Worksheet name*

Tip Rename each new worksheet with a meaningful name. This is better than keeping the default name, which is a timestamp. This will help you and others find your work later, by using the descriptive name.

As soon as you create a new worksheet, you can write, modify, and execute your SQL queries from that worksheet. You can use the auto-complete functionality when typing SQL keywords and pressing the Tab key.

In the left side bar, there are two tabs — Worksheets and Databases — as shown in Figure 2-4. When you click the Databases tab, you will see a list of all Snowflake objects, such as databases, schemas, tables, views, stages, and so on, to which your role is authorized.

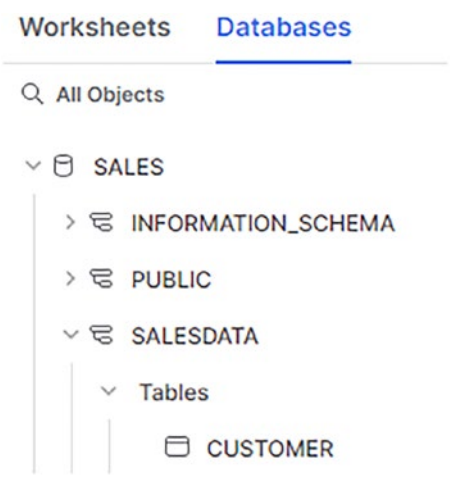

Figure 2-4. *Databases tab*

When you hover on an object name, you can use the three dots menu to place the fully qualified name of the object into your SQL query.

To execute your query, click the Play button (the blue circle with the right-pointing arrow) at the top right of the screen. Once the SQL query is executed, you will see the resulting data records in the lower portion of the screen, similar to what is shown in Figure 2-5.

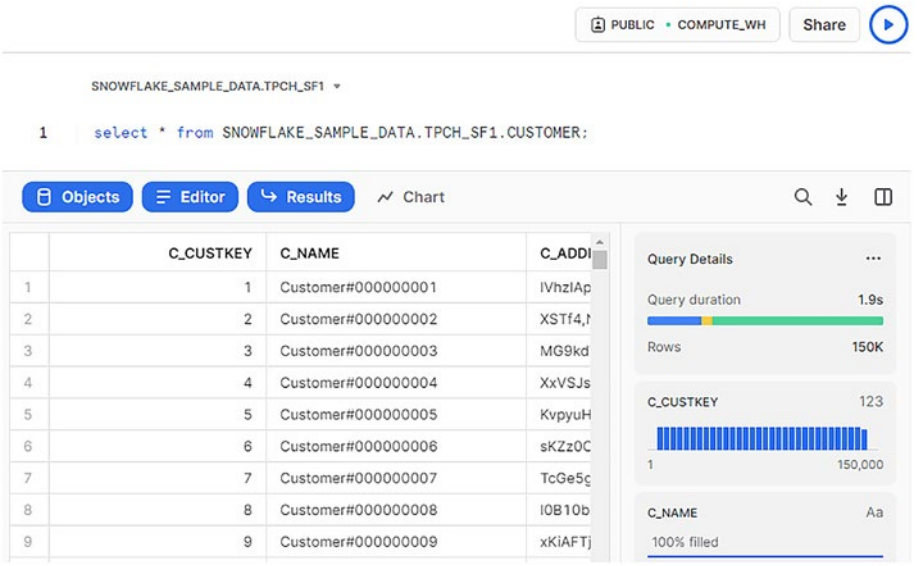

Figure 2-5. *Using a worksheet in Snowsight*

You don't have to use fully qualified object names in your SQL queries if you set the context on the worksheet. By default, each new worksheet will be created with "No Database Selected" at the top. When you click the down arrow next to this text, you will see a drop-down list where you can choose the database and the schema that you want to use in all queries in the worksheet.

In Figure 2-5, the database and schema name has been changed to `SNOWFLAKE_SAMPLE_DATA.TPCH_SF1`. You can still write SQL queries that access other databases and schemas on this worksheet, but you have to fully qualify them.

When you are done using the worksheet, click the Home icon (the gray house) at the top left of the screen to go back to the default screen. You will see a list of your worksheets. From this screen, you can perform additional actions, such as create new worksheets, group worksheets into folders, share worksheets with other users, and so on. You can also delete worksheets when you no longer need them.

Dashboards

The Dashboards page is the entry point to curating collections of visualized data in Snowsight. This functionality is not covered on the SnowPro Core exam.

Data

The Data area of Snowsight allows you to explore your databases and objects, including tables, views, stages, stored procedures, and so on. When you click an object, additional information will appear on the right pane of the screen, depending on the type of the object. You will be able to grant or revoke privileges on the object to roles within the Snowflake account as well as perform other actions that are specific to each type of object. Chapters 3 and 4 explore the various objects in Snowflake and their properties.

Snowflake Marketplace

Snowflake Marketplace enables you to discover and query third-party data directly from your Snowflake account. You can use queries to join third-party data with your own data as well as use data from different vendors, all from within a single location in your Snowflake account. Chapter 10 describes the Snowflake Marketplace in more detail.

Activity

The Activity area of Snowsight allows you to monitor activities such as queries executed by users, query performance data, and the status of data loading. Chapters 5 and 7 provide more information about the Activity area.

Admin

The Admin area of Snowsight is used by the Snowflake administrator to perform typical administration tasks, such as:

- Understand Snowflake usage
- Manage Snowflake warehouses
- Set up and view details about resource monitors
- Manage users and roles
- Administer network policies and monitor session activity
- View billing information and set up payment methods

More details about some of these tasks are provided in subsequent chapters, where the relevant functionality is introduced.

Partner Connect

One of the options in the Admin area of Snowsight is called Partner Connect. This is a convenient feature provided by Snowflake that allows you to easily try various third-party tools and services from selected Snowflake business partners. You do this by creating a trial account with your chosen partner where you can test their tools and services before you commit to buying anything.

Partner Connect includes partners that provide tools and services in the following categories:

- Machine learning and data science
- Security and governance
- SQL development and management
- Data integration
- Business intelligence (BI)

Partner Connect can be accessed only by account administrators (users with the `ACCOUNTADMIN` role) who have a verified email address with Snowflake.

Snowflake Partner Connect is constantly changing, with new partners added. To get a complete list of currently supported partners along with their categories, go to Snowflake's website for the most up-to-date information.

Help & Support

When you click the Help & Support area of Snowsight, you can choose the Learn option. This page provides links to documentation and additional resources that will help you learn Snowflake. It's a supplement to what is covered in this study companion.

SnowSQL

SnowSQL is the command-line interface (CLI) client for connecting to Snowflake. Its primary purpose is to load data into Snowflake and unload data out of Snowflake, but you can execute any SQL queries and commands to work with Snowflake.

Snowflake provides SnowSQL clients for the following platforms:

- Microsoft Windows
- macOS
- Linux

SnowSQL can be used interactively by executing individual commands or calling them in batches.

Install SnowSQL

You use SnowSQL to load and unload data in Snowflake, as described in Chapter 7. This section reviews the steps for downloading and running the installer package to install SnowSQL on Microsoft Windows. To install SnowSQL on macOS or Linux, refer to the Snowflake documentation.

To download the SnowSQL installer for Windows, go to the SnowSQL download page at the following URL:

`https://developers.snowflake.com/snowsql/`

Click the SnowSQL for Windows button. The installer MSI file will be downloaded to your computer. Double-click the MSI file and follow the instructions provided by the installer.

Configure SnowSQL

To be able to use SnowSQL with your Snowflake account, you have to provide the connection information. You can do this in the command line each time you start SnowSQL. To avoid having to do this every time, you can edit the SnowSQL configuration file with your connection information.

Navigate to the following folder:

```
%USERPROFILE%\.snowsql
```

You should see a file named `config` in this folder. Edit this file with a text editor of your choice. Uncomment the configuration parameters that you will use, such as the account name, username, and password. If you want to, you can also define your default database name, schema name, warehouse name, or role.

Connectors and Drivers

To access Snowflake programmatically, many connectors and drivers are available. These connectors and drivers support popular development platforms and programming languages, particularly Python, Spark, JDBC, ODBC, Kafka, Node.js, Go, .NET, PHP PDO, and Snowflake SQL API. Depending on the development platform or programming language that you want to use, you will download the respective connector or driver and follow the instructions provided in the Snowflake documentation.

Setting Up Your Snowflake Account

To use your Snowflake account for the practice examples presented in subsequent chapters, you must set up an initial virtual warehouse and configure the Snowflake sample database.

Create a Virtual Warehouse

To be able to run queries, you need a virtual warehouse. For this purpose, create a virtual warehouse named `COMPUTE_WH` with size X-Small and grant usage on this warehouse to everyone in the account. Create a new worksheet in Snowsight, as described earlier. Type the following commands and execute each command using the Play button:

```
use role SYSADMIN;
create warehouse COMPUTE_WH with warehouse_size = 'X-Small';
grant usage on warehouse COMPUTE_WH to PUBLIC;
```

Creating virtual warehouses is explained in more detail in Chapter 6.

Configure the Snowflake Sample Database

Click the Data option in the left menu in Snowsight and choose Databases, as shown in Figure 2-6.

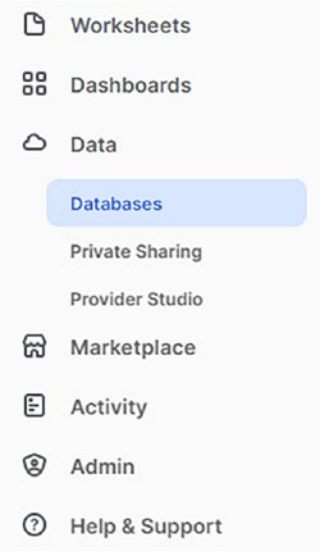

Figure 2-6. *Data and Databases option in Snowsight*

Check whether the SNOWFLAKE_SAMPLE_DATA database is available in the list of databases that appear. If it is available, you don't have to do anything.

If the SNOWFLAKE_SAMPLE_DATA database is not available, you have to create it from the Snowflake share and grant usage to everyone in the account. Create a new worksheet in Snowsight, as described earlier. Type the following commands and execute each command using the Play button:

```
use role ACCOUNTADMIN;
create database SNOWFLAKE_SAMPLE_DATA
  from share sfc_samples.sample_data;
grant imported privileges on database SNOWFLAKE_SAMPLE_DATA
  to role PUBLIC;
```

Summary

This chapter reviewed the basic features of the Snowflake Cloud Data platform. It described various Snowflake editions, cloud platforms, and regions where it can be hosted. It explored the web user interface as a means to interact with Snowflake and introduced the command-line interface and Snowflake connectors.

It also performed a brief initial setup of the Snowflake account in preparation for practice examples presented in subsequent chapters.

The next chapter delves into the Snowflake architecture and explains the different components that make up the architecture in more detail.

CHAPTER 3

Snowflake Architecture

Snowflake's unique architecture consists of three layers:

- Database storage
- Query processing
- Cloud services

The separation into distinct layers allows superior flexibility and cost effectiveness as compared to traditional databases.

This chapter describes each of the architecture layers and their purposes in more detail. Then it walks through and reviews various object types that are stored in Snowflake.

Architecture Layers

Snowflake stores data in a central data repository that is accessible from all compute nodes in the platform, similar to traditional shared-disk architectures. Queries are processed using MPP (massively parallel processing) compute clusters, where each node in the cluster stores a portion of the data locally, similar to shared-nothing architectures. This combination offers the data management simplicity of a shared-disk architecture, with better performance and scalability of a shared-nothing architecture.

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_3

In addition to storing data and processing queries, Snowflake takes care of cloud services, including authentication, access control, security, infrastructure management, query optimization, and metadata management. The three architecture layers are depicted in Figure 3-1 and described in more detail in the following sections.

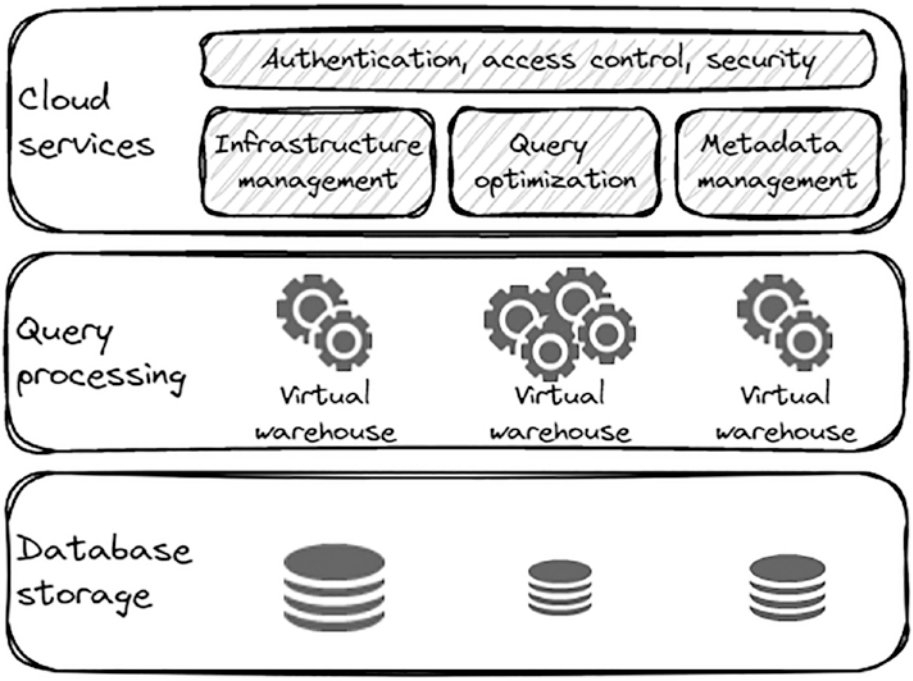

Figure 3-1. *Snowflake's architecture layers*

Database Storage

The first layer of Snowflake's architecture is database storage. Data is stored in the object storage of the cloud provider (AWS, Azure, or GCP), where the Snowflake account is hosted. When data is loaded into Snowflake—whether it is structured, semi-structured, or unstructured—it is reorganized into Snowflake's internal optimized, compressed, columnar format.

Snowflake takes care of how the data is stored internally. These internal data storage components are not visible to the users. Data can only be accessed by executing SQL queries and statements.

Query Processing

The second layer of Snowflake's architecture is the query processing layer. This is where queries are processed. Snowflake processes queries using *virtual warehouses* (not to be confused with data warehouses). Each virtual warehouse is an MPP (massively parallel processing) compute cluster composed of multiple compute nodes allocated by Snowflake from a cloud provider.

Each virtual warehouse is an independent compute cluster that does not share compute resources with other virtual warehouses. This is the basis of Snowflake scalability, because each virtual warehouse has no impact on the performance of other virtual warehouses. When more users are added, more virtual warehouses can be provisioned without degrading overall performance.

Virtual warehouses are described in more detail in Chapter 6.

Cloud Services

The third layer of Snowflake's architecture is the cloud services layer. This is a collection of services that coordinate all activities across Snowflake, such as handling user logins, processing user requests, dispatching queries, and taking care of security. The cloud services layer runs on compute instances provisioned by Snowflake from the cloud provider (AWS, Azure, or GCP) that is hosting the Snowflake account.

Services managed in this layer include:

- Authentication
- Infrastructure management
- Metadata management
- Query parsing and optimization
- Access control

Subsequent chapters describe each of these services in more detail.

Snowflake Objects

All data in Snowflake is maintained in databases. Each database consists of one or more schemas, which are logical groupings of database objects, such as tables and views as well as other types of objects including functions, stored procedures, tasks, streams, sequences, and more. There is no limit to the number of databases, schemas within databases, or objects within schemas that you can create in Snowflake.

Metadata about all objects in each database is stored in a dedicated schema named `INFORMATION_SCHEMA` that Snowflake automatically creates in every database in an account. This schema is maintained by Snowflake and is available as read-only to the user. Metadata about objects in the database is stored in `INFORMATION_SCHEMA` in the following ways:

- Views for all objects contained in the database
- Views for non-database objects such as roles and warehouses
- Table functions for historical usage data

You can view the data that is stored in `INFORMATION_SCHEMA` using standard SQL queries. For example, to retrieve a list of all tables in the Snowflake sample database named `SNOWFLAKE_SAMPLE_DATA`, you would execute the following query:

```
select *
from SNOWFLAKE_SAMPLE_DATA.INFORMATION_SCHEMA.TABLES;
```

Figure 3-2 shows part of the output of this query.

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME
SNOWFLAKE_SAMPLE_DATA	TPCDS_SF10TCL	PROMOTION
SNOWFLAKE_SAMPLE_DATA	TPCDS_SF10TCL	WEB_SITE
SNOWFLAKE_SAMPLE_DATA	TPCH_SF10	LINEITEM
SNOWFLAKE_SAMPLE_DATA	TPCH_SF1000	PART
SNOWFLAKE_SAMPLE_DATA	TPCDS_SF10TCL	WAREHOUSE
SNOWFLAKE_SAMPLE_DATA	TPCH_SF10	CUSTOMER
SNOWFLAKE_SAMPLE_DATA	TPCH_SF1	CUSTOMER
SNOWFLAKE_SAMPLE_DATA	TPCH_SF10	PART

Figure 3-2. *Selecting from the TABLES view in INFORMATION_SCHEMA*

The output of a view or table function in `INFORMATION_SCHEMA` depends on the privileges granted to the user's current role. Only objects for which the current role has been granted access privileges are returned.

The following sections describe the main types of Snowflake objects in more detail.

Tables

Data in Snowflake is stored in database tables. Similar to other relational databases, a table consists of a set of rows and columns. Snowflake supports defining permanent, temporary, or transient tables. Each of these table types is suitable for a particular use case, primarily based on how long the data must be preserved.

Permanent Tables

Permanent tables are the default table type in Snowflake. These tables remain persistent after the user session ends and are accessible to other users depending on granted privileges. The data stored in permanent tables is covered by Snowflake's storage protection and is available for time travel and is failsafe.

To create a table, you use the standard `SQL CREATE TABLE` statement. But before that, since a table is always created in a database and a schema, you'll create a new database named `SALES` and a new schema named `SALESDATA`. For this exercise, you use the `ACCOUNTADMIN` role because it has all the required privileges (you learn how to create custom roles and grant privileges in Chapter 4). You will use this role, database, and schema for all exercises in this chapter, so go ahead and create them now using the following SQL statements:

```
use role ACCOUNTADMIN;
create database SALES;
create schema SALESDATA;
```

Then create a table named `CUSTOMER`, like this:

```
create table CUSTOMER (
    c_custkey NUMBER(38,0) NOT NULL,
    c_name VARCHAR(40),
    c_address VARCHAR(255),
    c_country VARCHAR(3),
    c_mktsegment VARCHAR(255)
);
```

Snowflake supports the `CREATE OR REPLACE` syntax in most statements, including in the `CREATE TABLE` statement, allowing you to re-create any objects without having to drop them first.

Note Although Snowflake supports the standard SQL syntax for creating primary keys, foreign keys, and unique constraints, these constraints are not enforced when data is inserted into tables. Only `NOT NULL` constraints are enforced.

Temporary Tables

Temporary tables are used to store non-permanent data that is used only in the current session. Once the session ends, data stored in temporary tables is gone for good and is not recoverable by Snowflake data protection features. Temporary tables are not visible to other users or sessions.

Temporary tables contribute to storage costs for the period of time that the temporary table exists.

To create a temporary table, specify the `TEMPORARY` keyword in the `CREATE TABLE` statement, for example:

```
create temporary table CUSTOMER_TEMP (
    c_custkey NUMBER(38,0) NOT NULL,
    c_name VARCHAR(40),
    c_address VARCHAR(255),
    c_country VARCHAR(3),
    c_mktsegment VARCHAR(255)
);
```

Snowflake allows you to create a temporary table with the same name as an existing permanent table. If you do so, the temporary table will take precedence over the permanent table with the same name within the session in which the temporary table was created. This means that when you execute a query using a table name that is used by both a temporary and a permanent table, the query will use the temporary table rather than the permanent table.

Transient Tables

Transient tables are somewhere between permanent and temporary tables with respect to how long the data is preserved and their visibility. The key difference between transient and permanent tables is that transient tables don't have a failsafe period. The key difference between transient and temporary tables is that data in transient tables is persisted after the user session ends and is accessible by other users depending on granted privileges.

Transient tables are used for non-permanent data that doesn't need the same level of data protection and recovery as permanent tables. A use case for transient tables is creating intermediate tables that are populated by ETL processes that are truncated and repopulated during each execution of the ETL process.

Transient tables contribute to storage costs. However, in comparison to permanent tables, storage costs are usually lower as there are no additional costs associated with failsafe data protection.

Note In addition to transient tables, Snowflake also supports creating transient databases and transient schemas. All tables created in a transient schema, as well as all schemas created in a transient database, are transient.

To create a transient table, you specify the `TRANSIENT` keyword, for example:

```
create transient table CUSTOMER_TRANSIENT (
    c_custkey NUMBER(38,0) NOT NULL,
    c_name VARCHAR(40),
    c_address VARCHAR(255),
    c_country VARCHAR(3),
    c_mktsegment VARCHAR(255)
);
```

Once you have created a transient table, you will not be able to modify it to a permanent table and vice versa. The reason is that the table type is defined when the table is created. If you want to convert a permanent table to a transient table or a transient table to a permanent table, you must create a new table with the desired type and insert the data from the source table into the new table.

To view a list of all tables that you have created, you can query `INFORMATION_SCHEMA` like earlier. Another way is to use the `SHOW TABLES` command, which is usually more efficient than querying `INFORMATION_SCHEMA` because it doesn't require a running warehouse to execute, and it limits the results to the current schema:

```
show tables;
```

Sample output from this command in the current `SALESDATA` schema is shown in Figure 3-3.

name	database_name	schema_name	kind ···
CUSTOMER	SALES	SALESDATA	TABLE
CUSTOMER_TEMP	SALES	SALESDATA	TEMPORARY
CUSTOMER_TRANSIENT	SALES	SALESDATA	TRANSIENT

Figure 3-3. *Sample output of the SHOW TABLES command*

Table 3-1 summarizes the differences between permanent, transient, and temporary table types.

Table 3-1. *Comparison of Table Types*

Table Type	Persistence	Time Travel	Failsafe
Temporary	Remainder of session	0 or 1 days	0 days
Transient	Until dropped	0 or 1 days	0 days
Permanent (Standard Edition)	Until dropped	0 or 1 days	7 days
Permanent (Enterprise Edition and higher)	Until dropped	0 to 90 days	7 days

Views

Snowflake supports creating standard SQL views. A view is basically a named definition of a query that allows the result of the query to be used as if it were a table. The results of a view are created by executing the query at the time that the view is referenced in another query.

Views serve many purposes, such as protecting data by exposing only a subset of the data in underlying tables or simplifying complex queries by hiding part of the query logic behind a view that is then used in another query.

Practically any SQL query can be used to create a view. The query is specified in the `CREATE VIEW` statement. For example, to create a view that selects all customers from the United States (these are identified by the `c_country` column containing the value `'US'`), the following statement can be executed:

```
create view CUSTOMER_VIEW_US as
select
    c_custkey,
```

```
    c_name,
    c_address,
    c_country,
    c_mktsegment
from CUSTOMER
where c_country = 'US';
```

If the schema is not specified, Snowflake assumes that the table is in the same schema as the view.

Caution In many databases, when you create a view and don't specify the schema, the database will use the current schema. This behavior differs from Snowflake, so be sure you understand this concept thoroughly.

Views in Snowflake are read-only, which means that you cannot execute DML commands such as update a view.

Data Types

Snowflake supports most basic SQL data types that can be used in columns, variables, expressions, or parameters. The main data type groups in Snowflake are:

- Numeric data types
- String and binary data types
- Logical data types
- Date and time data types
- Semi-structured data types
- Geospatial data types

When data types are compatible, a value expressed in one data type can be converted to another data type. For example, INTEGER data can be converted to FLOAT or STRING data can be converted to DATE if the date format in the provided string is recognized by Snowflake.

Data type conversion is done implicitly by Snowflake whenever possible. Users can also explicitly cast a value from one data type to another data type using any of the following functions or operators:

- The CAST function; for example `CAST (amount AS INTEGER)`
- The `::` operator; for example `amount::INTEGER`
- The corresponding SQL function, if available; for example `TO_DATE('2022-07-04')`

Numeric Data Types

Snowflake supports fixed-point and floating-point numeric data types. The most commonly used fixed-point data type is NUMBER; it can store up to 38 digits, with an optional precision (the total number of digits) and scale (the number of digits to the right of the decimal point), for example `NUMBER(18, 2)`. Synonyms for NUMBER are DECIMAL and NUMERIC.

To define integer data types, `NUMBER(38, 0)` can be used, which indicates zero digits to the right of the decimal point. Synonyms for integer data types include INT, INTEGER, BIGINT, SMALLINT, and others, where you don't specify the precision and scale.

Floating point numbers can be defined as FLOAT, FLOAT4, FLOAT8, DOUBLE, REAL, and others. All of these are represented as 64-bit floating-point double precision numbers in Snowflake.

String and Binary Data Types

Snowflake supports data type VARCHAR that stores Unicode UTF-8 characters for character strings. The VARCHAR keyword accepts an optional parameter, which represents the maximum number of characters that will be stored, for example VARCHAR(255). If you don't specify the parameter, the maximum allowed length 16,777,216 will be used.

A VARCHAR is also limited to a maximum number of bytes, which is 16MB.

Note Remember the 16MB limit. This is not only the maximum allowed length of a character string, but also the size of a Snowflake micro-partition, which limits the size of all objects and values that can be stored as a single unit in Snowflake.

Snowflake supports the BINARY data type for binary strings. The maximum length of a binary string is 8MB, regardless of the encoding. The size limit is set at 8MB so that the variable fits within 16MB when converted to a hexadecimal string.

Logical Data Types

Snowflake supports BOOLEAN as a logical data type with TRUE, FALSE, or NULL values. The NULL value represents an unknown value just like in other data types.

Boolean values can be converted from text string and numeric values to logical values according to the following mappings:

- Strings that represent TRUE values: 'true', 't', 'yes', 'y', 'on', '1' (case-insensitive)
- Strings that represent FALSE values: 'false', 'f', 'no', 'n', 'off', '0' (case-insensitive)

- Numbers that represent FALSE values: 0 (zero)
- Numbers that represent TRUE values: Any non-zero value

Date and Time Data Types

Snowflake supports data types for storing dates, times, and timestamps. The DATE data type is used for storing dates with no time parts. It recognizes dates in the most common formats, such as YYYY-MM-DD, DD-MON-YYYY, and more.

Snowflake supports the following data types for storing timestamps:

- TIMESTAMP_LTZ. Internally, the value is stored as a UTC timestamp, but all operations on the value are performed using the current session time zone defined by the TIMEZONE parameter.
- TIMESTAMP_NTZ or DATETIME. Stores the timestamp value without taking a time zone into consideration. Sometimes this is also referred to as "wallclock" time.
- TIMESTAMP_TZ. Stores the timestamp together with a time zone offset. When a time zone is not provided, the value of the current session TIMEZONE parameter is used.

Due to limitations of the Gregorian calendar, Snowflake recommends that you avoid using years prior to 1582 in timestamp values.

The TIME data type is used for storing time values formatted as HH:MI:SS. Fractional seconds are also supported by adding an optional precision parameter, for example TIME(9), which means nanoseconds and is also the default. All operations on TIME values are performed without taking any time zone into consideration.

Semi-Structured Data Types

The following Snowflake data types can be used to store semi-structured data:

- `VARIANT`. Can contain any data type, including `OBJECT` and `ARRAY`. The maximum length of a `VARIANT` value is 16MB.
- `ARRAY`. Contains zero or more elements, just like arrays used in programming languages. Each element is addressed by its position in the array.
- `OBJECT`. Contains key/value pairs where keys are stored as `VARCHAR` data types, and values are stored as `VARIANT` data types.

A combination of `VARIANT`, `ARRAY`, and `OBJECT` data types enable you to represent data in semi-structured formats, such as JSON, Avro, ORC, Parquet, and XML.

Chapter 8 discusses working with semi-structured data types in more detail.

Geospatial Data Types

Snowflake provides geospatial data types including points, lines, and polygons on the Earth's surface. The following geospatial data types are available:

- `GEOGRAPHY`. Uses the WGS 84 standard, where points on the Earth are represented as degrees of longitude from -180 degrees to +180 degrees and latitude from -90 to +90. Altitude is currently not supported.

- `GEOMETRY`. Coordinates are represented as pairs of real numbers with units that are determined by the spatial reference system (SRS).

If you want to store geospatial data in Snowflake using various input formats such as longitude and latitude, WKT, WKB, GeoJSON, and similar, Snowflake recommends that you convert this data into geography or geometry data types. This will improve query performance by taking advantage of geospatial functionality as compared to querying data in its original format.

User-Defined Functions

In addition to system-defined functions provided by Snowflake that you can use in SQL queries, you can also write your own user-defined functions (UDFs) and use them similarly as the system-defined functions.

A key difference between system-defined functions and UDFs is that UDFs are database objects, which means that they reside in a database and schema. When you call a UDF without fully qualifying the name, Snowflake will look for the UDF in the database and schema that is used in the current session. A system-defined function is never fully qualified and can be called from any database or schema.

A UDF can be written in one of these supported programming languages:

- SQL
- JavaScript
- Java
- Python

Many factors can affect which programming language you choose to write a UDF, including:

- The capabilities of the language.
- Whether you already have code in a particular language. For example, if you already have a .JAR file that contains a method that does the work you need, you might prefer Java as your programming language.
- Whether a language has libraries that can help you do the processing that you need. For example, if you use Python machine learning libraries, you might prefer Python as your programming language.
- Whether you are migrating SQL stored procedures from other database systems. Although the syntax may differ, you might prefer SQL as your programming language to minimize the amount of code rewriting.

A UDF can return either scalar or tabular results. The properties of each are:

- **Scalar UDF.** A scalar function returns a single column or value. When you use a scalar function in a query, it results in one output row for each input row.
- **Tabular UDF.** A tabular function, also called a UDTF (user-defined table function), returns zero, one, or many rows for each input row. In a tabular UDF, the return statement contains the `TABLE` keyword. You must also specify the names and data types of the columns that the function will return.

UDF names may be overloaded, meaning that you can have more than one UDF with the same name, but different arguments, either by the number of arguments or the argument data types. When calling an overloaded UDF, the correct function is determined by the arguments that were provided in the function call.

Let's work with some examples. Similarly to creating tables and views, functions must be created in a database and schema. The following examples continue to use the SALES database and the SALESDATA schema.

A UDF can use either a general expression or a query expression. An example of a UDF with a general expression is a function that calculates the cube of an integer. Here is the code that creates this function using SQL:

```
create function CUBED (x integer)
    returns integer
as
$$
    x * x * x
$$
;
```

As you can see from the code, the name of the function is CUBED. The function takes x as an input argument with data type INTEGER. The return data type is also INTEGER. In the body of the UDF, between the $$ signs, you have a single expression that multiplies the input argument by itself three times, thus calculating the cube of the argument. No RETURN expression is needed, because the function knows that the result of the expression in the body is what will be returned.

You execute the function by using it in a query, like so:

```
select CUBED(3);
```

The result is 27, as expected.

Another type of a UDF is one that returns the result of a SQL query where the query expression returns at most one row. As an example, you might create a UDF that counts the number of customers in a given market segment in the Snowflake sample database using the following SQL code:

```
create function CUSTOMERS_BY_MARKET_SEGMENT (segment varchar)
```

```
    returns integer
as
$$
    select count(c_custkey)
    from SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.CUSTOMER
    where c_mktsegment = segment
$$
;
```

As you can see from the code, the name of the function is `CUSTOMERS_BY_MARKET_SEGMENT`. The function takes `segment` as an input argument with data type `VARCHAR`. The return data type is `INTEGER` because you will be returning a whole number as the count. In the body of the UDF, between the `$$` signs, there is a single SQL query that returns exactly one row and one column, which is the value that the UDF will return. The query in the UDF can refer to the input arguments of the function, as in the previous example, where you provided the market segment name as an input argument. Note that there is no semicolon in the body of the UDF.

You execute the function by using it in a query, like so:

```
select CUSTOMERS_BY_MARKET_SEGMENT ('MACHINERY');
```

The result is a number that represents the number of customers in that market segment. You can test whether the UDF is returning the correct result by executing the SQL query used in the body of the function manually.

The query expression in a UDF can reference database objects such as tables, views, and sequences. If an object in the expression is unqualified, it will be resolved in the schema that contains the UDF. In case the expression references an object in a different schema, the owner of the function must have sufficient privileges to access the object. Similarly as in views, the user calling the UDF does not need access to the objects referenced in the query expression within the UDF, but only needs the privilege to execute the UDF.

Stored Procedures

Stored procedures allow you to write procedural code that executes SQL statements as well as other programming constructs. Some of the benefits of using stored procedures include:

- Writing procedural logic such as looping and conditional statements that SQL queries don't support.
- Including error handling logic.
- Creating and executing SQL statements that have been constructed dynamically.
- Writing a sequence of multiple SQL statements that are performed as a single unit of code.
- Calling a function or another stored procedure for additional procedural logic.

You can write a stored procedure in one of the following languages:

- Snowflake Scripting
- JavaScript
- Java (using Snowpark)
- Scala (using Snowpark)
- Python (using Snowpark)

Stored procedures can be written so that the code in the procedure is executed either with the privileges of the role that owns the procedure or with the privileges of the role that calls the procedure. We refer to such

stored procedures as either owner's rights stored procedures or caller's rights stored procedures. The properties of each type of stored procedure are as follows:

- **Caller's rights stored procedure.** This type of stored procedure executes with the privileges of the role that calls the procedure. This means that the procedure caller must have sufficient privileges to execute all statements in the stored procedure. The procedure can also access the caller's current session and use session variables if needed.
- **Owner's rights stored procedure.** This type of stored procedure executes with the privileges of the role that owns the procedure. The reason that you might write a procedure with owner's rights is so that the owner can enable the caller to execute specific tasks that require additional privileges without granting those privileges. In this case, the role that owns the stored procedure must have sufficient privileges to execute all statements in the stored procedure.

A stored procedure is designated as a caller's rights stored procedure by specifying the `EXECUTE AS CALLER` keywords when creating the procedure. Alternatively, the `EXECUTE AS OWNER` keywords will designate a stored procedure as an owner's rights stored procedure.

Here is an example of a stored procedure that creates a copy of the `CUSTOMER` table and inserts records into a log table named `LOG_TABLE` (you have to create the log table first):

```
create table LOG_TABLE (log_ts timestamp, message
varchar(255));

create or replace procedure COPY_CUSTOMER()
```

```
    returns integer
    language sql
    execute as caller
as
begin
    create or replace table CUSTOMER_COPY as
        select * from CUSTOMER;
    insert into LOG_TABLE (log_ts, message)
        values (CURRENT_TIMESTAMP(), 'Customer copy
completed');
end;
```

As you can see from the code, the name of the stored procedure is `COPY_CUSTOMER`. The stored procedure doesn't take any input arguments. The return data type must be defined, even though the stored procedure is not meant to return a value. In this code, the return type is defined as `INTEGER`. The stored procedure is written using SQL as the language and it will be executed with caller's rights.

The body of the stored procedure contains two SQL statements. The first statement creates a new table named `CUSTOMER_COPY` as a copy of the table `CUSTOMER`. The second statement inserts a record into the `LOG_TABLE` table.

You can execute the stored procedure using the following statement:

```
call COPY_CUSTOMER();
```

The statement returns `NULL`, because no return value was specified. To verify if the stored procedure succeeded, you can check whether the `CUSTOMER_COPY` table was created and if it contains records. You can also select data from the `LOG_TABLE` table to see if a record with the timestamp when the stored procedure was executed was inserted.

Comparing UDFs and Stored Procedures

UDFs (user-defined functions) and stored procedures can both be used to write modular code. Each of these types of objects serves a different purpose and is executed differently, as summarized in Table 3-2.

Table 3-2. *Comparison of UDFs and Stored Procedures*

Comparison Attribute	UDF	Stored Procedure
Purpose	To calculate and return a value.	To execute SQL statements.
Return value	Always returns a value explicitly by specifying an expression.	May return a value, such as an error indicator, but is not required to do so.
Use	Used in queries where a general expression can be used.	Called as an independent statement.
Execution statement	`select MY_FUNCTION();`	`call MY_PROCEDURE();`

To decide whether to create a UDF or a stored procedure, some of the following factors can be taken into consideration.

When to create a UDF:

- When you are migrating a function from an existing application.
- When you need a function that can be called as part of a SQL statement.
- When you require a single output value for every input row of a SQL statement.

When to create a stored procedure:

- When you are migrating a stored procedure from an existing application.
- When you want to execute SQL statements such as `SELECT`, `UPDATE`, `DELETE`, and so on.
- When you want to execute DDL statements to create or modify database objects.
- When you want to execute any other types of statements such as creating users, creating and granting roles, and so on.

Streams

Streams are Snowflake's way to support CDC (change data capture). A stream can be created on the following types of tables:

- Standard tables
- External tables
- Directory tables
- Underlying tables for a view (in case we have a stream on a view)

When a stream is created on a table, the snapshot of the table at that point in time represents the initial state of the stream. Each subsequent SQL statement that inserts, updates, or deletes data in the table creates a new version of the table. The stream points to each version of the table at a given point in time.

Note Streams don't contain any table data. When a stream is created on a table, hidden columns are added to the table that store change tracking metadata.

Creating Streams

To explain how a stream behaves, let's work with an example. You will start by creating a table and inserting a couple of initial records using the following statements:

```
create table CUSTOMERS_FOR_CDC (
    c_custkey NUMBER(38,0),
    c_name VARCHAR(40),
    c_address VARCHAR(255),
    c_country VARCHAR(3)
);
insert into CUSTOMERS_FOR_CDC
values (1, 'First Customer', 'Street One', 'US');
insert into CUSTOMERS_FOR_CDC
values (2, 'Second Customer', 'Street Two', 'US');
```

Next, create a stream on the table you just created using the `CREATE STREAM` statement:

```
create stream CUSTOMERS_FOR_CDC_STREAM
on table CUSTOMERS_FOR_CDC;
```

Because you just created the stream, the stream will be empty. You can verify that with the following query:

```
select * from CUSTOMERS_FOR_CDC_STREAM;
```

This query returns the message "Query produced no results".

Now you can execute a DML operation on the table. You will insert a new record into the table using the following statement:

```
insert into CUSTOMERS_FOR_CDC
values (3, 'Third Customer', 'Street Three', 'US');
```

Because you inserted new data into the table, the table has changed, thus taking on a new version. The stream will record this change and you can view the contents of the stream using the following statement again:

```
select * from CUSTOMERS_FOR_CDC_STREAM;
```

Now the stream returns the data as shown in Figure 3-4.

C_CUSTKEY	C_NAME	C_ADDRESS	C_COUNTRY	METADATA$ACTION	METADATA$ISUPDATE	METADATA$ROW_ID
3	Third Customer	Street Three	US	INSERT	FALSE	2a121172ea72f4b8c38

Figure 3-4. *Results of the stream after inserting one record*

In Figure 3-4 you can see that the stream returns all columns from the underlying table on which the stream was created. In addition, there are three metadata columns:

- `METADATA$ACTION`. Contains the name of the DML operation that was executed. This column contains either the value `INSERT` or `DELETE`. When an `UPDATE` operation is executed on the table, the stream records the change as a `DELETE` of the previous version of the data before it was updated and an `INSERT` with the new version of the data.
- `METADATA$ISUPDATE`. Contains a Boolean `TRUE` or `FALSE` flag, which indicates whether the operation was part of an `UPDATE` statement.
- `METADATA$ROW_ID`. Contains the unique ID of the row.

Streams record only the difference between two versions of a table. If more than one DML operation is executed on a row, for example when a row is added and then updated, the stream will record only the difference from the previous version and the current version, which is a new row.

Let's check what happens when you update the record that you just inserted. You will update the country from `'US'` to `'FR'` using the following statement:

```
update CUSTOMERS_FOR_CDC
set c_country = 'FR'
where c_custkey = 3;
```

You can check the stream again, using this statement:

```
select * from CUSTOMERS_FOR_CDC_STREAM;
```

The stream returns the data as shown in Figure 3-5.

C_CUSTKEY	C_NAME	C_ADDRESS	C_COUNTRY	METADATA$ACTION	METADATA$ISUPDATE	METADATA$ROW_ID
3	Third Customer	Street Three	FR	INSERT	FALSE	2a121172ea72f4b8c38

Figure 3-5. *Results of the stream after updating the previous record*

In Figure 3-5, you can see that the stream is showing only one row, as if the updated record was initially inserted, which you see from the column `METADATA$ISUPDATE`, which is `FALSE`. The intermediate steps where you first inserted a row and then updated it are not included.

Let's also look at what happens in the stream when you update one of the records in the table that were there before you created the stream. You will update the first record by changing the country from `'US'` to `'IN'`, using this statement:

```
update CUSTOMERS_FOR_CDC
set c_country = 'IN'
where c_custkey = 1;
```

Check the stream again, using the statement:

```
select * from CUSTOMERS_FOR_CDC_STREAM;
```

Now the stream returns the data as shown in Figure 3-6.

C_CUSTKEY	C_NAME	C_ADDRESS	C_COUNTRY	METADATA$ACTION	METADATA$ISUPDATE	METADATA$ROW_ID
3	Third Customer	Street Three	FR	INSERT	FALSE	db4a9d6c288a6392457c1524
1	First Customer	Street One	US	DELETE	TRUE	394660c87bc51a02098ec583
1	First Customer	Street One	IN	INSERT	TRUE	394660c87bc51a02098ec583

Figure 3-6. *Results of the stream after updating a second record*

In Figure 3-6, you can see that the stream now contains two additional records, both related to the record with `c_custkey` = 1 that you updated. There is one record where `METADATA$ACTION` is `DELETE` and `METADATA$UPDATE` is `TRUE`, and another record where `METADATA$ACTION` is `INSERT` and `METADATA$UPDATE` is `TRUE`. This pair of records represents the `UPDATE` operation.

Consuming Streams

To consume the contents of a stream after it has been accumulating changed data for a while, a statement that selects from the stream and writes to a target table is executed. All contents of the stream are used in a transaction that inserts data from the stream into a target table. Just querying a stream using a `SELECT` statement does not consume it.

For example, you can insert the records from the stream into the target table `CUSTOMERS_FINAL` using the following statements (creating the target table first):

```
create table CUSTOMERS_FINAL like CUSTOMERS_FOR_CDC;

insert into CUSTOMERS_FINAL
select c_custkey, c_name, c_address, c_country
from CUSTOMERS_FOR_CDC_STREAM
where METADATA$ACTION = 'INSERT';
```

All data from a stream is consumed in a successfully committed transaction. You may require more than one statement to consume a stream; for example you could write a part of the data from a stream to one target table and another part of the stream to another target table. In such situations, you must surround the `INSERT` statements with an explicit transaction, starting with `BEGIN` and ending with `COMMIT`.

When multiple consumers want to use changed data from the same source table, separate streams can be created for each consumer. Since the stream tracks only the offsets but not the actual data from the underlying table, creating additional streams for an object will not interfere with the way the changed records are consumed by each consumer.

Types of Streams

The following stream types are available in Snowflake:

- **Standard**. Can be created on tables, directory tables, or views. A standard stream tracks all changes to the source object, including inserts, updates, and deletes.
- **Append-only**. Can be created on standard tables, directory tables, or views. An append-only stream tracks inserts only; it does not track updates or deletes.
- **Insert-only**. Can be created on external tables. An insert-only stream tracks inserts only; it does not track deletes, such as removing a file from the cloud storage location of the external table.

Tasks

Tasks are user-defined objects that support scheduled execution of SQL statements. A task can be executed on a schedule that is defined when the task is created. A task can also be executed by a parent task if a task dependency has been defined. Snowflake does not support executing tasks based on events.

Tasks are often combined with streams to create ETL processes for loading data on a regular basis. Streams ensure that only changed data is loaded to the target table each time the ETL process is executed and tasks ensure that loading is performed on schedule.

A task can execute different types of SQL code, such as:

- Execute a single SQL statement
- Call a stored procedure
- Execute procedural logic using Snowflake Scripting

When tasks execute, they require compute resources. These can be either user-managed via a virtual warehouse or Snowflake-managed by allowing Snowflake to manage the compute resources:

- **User-managed**. The user manages compute resources for tasks by specifying an existing virtual warehouse when creating the task.
- **Snowflake-managed**. Snowflake determines the size of compute resources for each task run based on historical runs of the same task. The user specifies a suggested warehouse size when the task is created as an initial size that is used until sufficient history is accumulated. Snowflake-managed compute resources in tasks are also referred to as serverless.

Snowflake always executes only one instance of a scheduled task at one point in time. If a task is still running when the next scheduled time of the same task occurs, that run will be skipped.

Snowflake recommends that you do not schedule tasks to run between 1 AM and 3 AM on Sundays because daylight savings changes happen during those hours and that could cause unexpected task executions. Alternatively, if you do have tasks scheduled during those hours, you should check the schedule manually on Sundays twice a year when the time changes.

To create a task, use the `CREATE TASK` command. Here is an example of creating a task that will execute a stored procedure every five minutes:

```
create task PERIODIC_COPY_CUSTOMER
  warehouse = COMPUTE_WH
  schedule = '5 MINUTE'
as
  call COPY_CUSTOMER()
;
```

As you can see from the code, this task is named `PERIODIC_COPY_CUSTOMER`. The task will run using the warehouse `COMPUTE_WH` (if you are using a 30-day free Snowflake trial account this warehouse should already be created and available) and it will run on schedule every five minutes. The body of the task calls the `COPY_CUSTOMER()` stored procedure that you created earlier.

Note Make sure that the statements that you provide in the body of the task execute as expected before you create the task. This will save you troubleshooting headaches later.

After a task is created, nothing happens. Every task is suspended immediately upon creation. If you want a task to run on schedule, you

must resume it before it will execute. However, if you just want to check that it is performing as expected, you can execute it manually a single time using the `EXECUTE TASK` command:

```
execute task PERIODIC_COPY_CUSTOMER;
```

To resume a task to allow it to run on schedule, use the `ALTER TASK` command, like this:

```
alter task PERIODIC_COPY_CUSTOMER resume;
```

Since this exercise uses the `ACCOUNTADMIN` role, you already have the required privilege to resume the task. Using any other role, this command will likely fail with an error message stating that the user's role doesn't have the `EXECUTE TASK` privilege. This privilege is not enabled by default and unless it has been explicitly granted, a typical user does not have it.

One way to resolve this issue is to grant the `EXECUTE TASK` privilege to the role that the user has set as the current role. Another way, as recommended by Snowflake, is to create a custom role and grant the `EXECUTE TASK` privilege to this role. Then this custom role can be assigned to users who will be executing tasks.

In this example, since you set the schedule to every five minutes, the task will run five minutes after you resume it, and every five minutes thereafter. To check the status of the task and to verify whether it is executing as expected, use the `TASK_HISTORY()` table function in `INFORMATION_SCHEMA` with the following statement:

```
select *
  from table(INFORMATION_SCHEMA.TASK_HISTORY())
  order by scheduled_time desc;
```

The results of this statement may look something like Figure 3-7.

QUERY_ID	NAME	QUERY_TEXT	STATE ···
null	PERIODIC_COPY_CUSTOMER	call COPY_CUSTOMER()	SCHEDULED
01a724fd-0000-29c8-0000-28e5000262aa	PERIODIC_COPY_CUSTOMER	call COPY_CUSTOMER()	SUCCEEDED
01a724f8-0000-29b8-0000-28e5000252ee	PERIODIC_COPY_CUSTOMER	call COPY_CUSTOMER()	SUCCEEDED
01a724f3-0000-29c8-0000-28e50002628a	PERIODIC_COPY_CUSTOMER	call COPY_CUSTOMER()	SUCCEEDED
01a724ee-0000-29b8-0000-28e5000252da	PERIODIC_COPY_CUSTOMER	call COPY_CUSTOMER()	SUCCEEDED

Figure 3-7. *Results from the TASK_HISTORY() table function*

The output from the `TASK_HISTORY()` function shows useful information, such as the next scheduled date and time when the task will run as well as a record for each of the previous executions of the task, where you can see if the task was successful. There will be an error message if the task didn't execute successfully.

Caution After you have completed testing the task, don't forget to set it to suspended or delete the task altogether. If you don't, the task will continue executing every five minutes and will needlessly consume resources.

Scheduling tasks to run every N minutes may not be the most useful way to orchestrate execution. More frequently, you'll schedule tasks to run at a given time on given days, for example, every day at 8 PM or every Saturday at midnight and so on. If you want these types of schedules, you can use a CRON expression in the `SCHEDULE` parameter when you create the task.

In addition to creating and scheduling individual tasks, Snowflake supports creating DAGs (directed acyclic graphs). A DAG is tree of tasks that contains one root task and one or more child tasks in a hierarchical manner. To schedule a DAG, you only schedule the root task. You don't schedule any child tasks because they will execute in turn after their predecessor completes. A DAG can contain a maximum of 1,000 tasks including the root task. A task can have no more than 100 parent tasks and no more than 100 child tasks.

Sequences

Sequences are used to generate unique numbers, for example to generate values for a primary key in a table or to generate unique values such as row numbering. Sequences that are generated by Snowflake consistently increase in value (or decrease in value if the increment is negative) but are not necessarily without gaps. Values generated by a sequence will always be unique, even if the same sequence is used in different SQL statements executed at the same time.

To create a sequence, use the `CREATE SEQUENCE` statement, for example:

```
create sequence CUST_PK_SEQ;
```

The default starting value is 1 and the default increment value is also 1.

You can retrieve the next value of the sequence using the `NEXTVAL` keyword and the following command:

```
select CUST_PK_SEQ.NEXTVAL;
```

After executing a statement using the `NEXTVAL` keyword, the sequence will advance to the next value, and the same value will not be returned from the sequence again.

Sequences can be used in tables as default column values to generate primary or unique keys. For example, you can create a table with the `c_custkey` column that uses `CUST_PK_SEQ.NEXTVAL` as the default value:

```
create table CUSTOMER_WITH_PK (
    c_custkey NUMBER(38,0) DEFAULT CUST_PK_SEQ.NEXTVAL ,
    c_name VARCHAR(40),
    c_address VARCHAR(255),
    c_country VARCHAR(3)
);
```

Insert some values for the `c_name`, `c_address`, and `c_country` columns into the `CUSTOMER_WITH_PK` table, allowing the `c_custkey` column to be generated from the sequence (you do not provide a value for this column because it will be populated using the default value):

```
insert into CUSTOMER_WITH_PK (c_name, c_address, c_country)
values ('First Customer', 'Street One', 'US');
insert into CUSTOMER_WITH_PK (c_name, c_address, c_country)
values ('Second Customer', 'Street Two', 'US');
```

After inserting the values, you can check the records in the table using the following query:

```
select * from CUSTOMER_WITH_PK;
```

The result from this query is shown in Figure 3-8.

C_CUSTKEY	C_NAME	C_ADDRESS	C_COUNTRY
1	First Customer	Street One	US
2	Second Customer	Street Two	US

Figure 3-8. *Result of selecting records from the CUSTOMER_WITH_PK table*

You can see that the values in the `c_custkey` column were populated automatically from the sequence.

Summary

This chapter described the three Snowflake architecture layers (database storage, query processing, and cloud services) and their purposes. It reviewed the main object types that are stored in Snowflake, including tables, views, user-defined functions, stored procedures, streams, tasks,

and sequences. It also walked through the data types that you can use in Snowflake.

The next chapter talks about Snowflake accounts and security features.

CHAPTER 4

Account Access and Security

Snowflake was built for the cloud and, as such, it is subject to many cloud security vulnerabilities. To protect it from unauthorized access, Snowflake provides features that ensure high levels of security for every account.

This chapter discusses Snowflake features related to security. It starts with a description of account identifiers and account parameters. It introduces network policies that allow or deny access to Snowflake from specific IP addresses. Next, it provides an overview of user authentication methods. Finally, it dives into more details about granting privileges to users through roles.

Many tasks that involve setting up and managing security are performed by Snowflake administrators. For the SnowPro Core certification exam, you don't need to know all the details of the administrative tasks, such as setting up federated authentication, for example. But as a Snowflake user, you must understand the security features and know how to use them appropriately. This will be tested on the SnowPro Core exam and is covered in this chapter.

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_4

Account Identifiers

Account identifiers uniquely identify a Snowflake account. They are used in places where you specify the account, for example:

- In URLs for accessing the Snowflake web interface
- In SnowSQL and other connectors or drivers for connecting to Snowflake
- In third-party applications and services that interact with Snowflake
- In API calls with external systems
- In Snowflake features such as secure data sharing or database replication

The URL of a Snowflake account has the following format:

```
account_identifier.snowflakecomputing.com
```

If you are using a 30-day free Snowflake trial account, you already know your account identifier. It is part of the URL that was issued to you when you signed up for your account.

Snowflake supports two formats of account identifiers:

- Legacy format
- Account name within an organization

The following sections review each of these formats in more detail.

Legacy Format

You can identify a Snowflake account by a combination of an account locator and optionally by the cloud region and the cloud name. An account

locator is a random string of characters and numbers that is assigned by Snowflake when the account is created. It cannot be changed afterward.

Depending on the region and cloud platform of the account, any of the following legacy format account identifiers are valid:

```
account_locator.cloud_region.cloud
account_locator.cloud_region
account_locator
```

Although the legacy format using account locators is still supported for identifying accounts in Snowflake, it is no longer preferred. Using the account name within an organization is now the preferred method.

Account Name Within an Organization

An organization is a Snowflake object associated with a business entity. The organization name is chosen when the first Snowflake account is provisioned for the organization. The organization name must be unique with respect to all other organizations in Snowflake.

Organization administrators manage all accounts in their organization. For example, they can create additional accounts in their organization. Account names must be unique within the organization, and they may be changed if required.

The account identifier for an account in an organization is made up of the organization name and the account name. The delimiter between the organization name and the account name may be a hyphen, an underscore, or a dot, depending on where and how the identifier is used:

- When the account identifier is used in URLs and other general-purpose scenarios, a hyphen serves as the delimiter, for example: `organization_name-account_name`

- When the account identifier is used in features where hyphens are not supported in URLs, an underscore serves as the delimiter, for example: `organization_name_account_name`
- When the account identifier is used in SQL commands, a dot serves as the delimiter, for example: `organization_name.account_name`

Parameter Management

Every Snowflake account provides parameters that can be set at different levels:

- Account parameters that affect the entire account
- Session parameters that affect users and sessions
- Object parameters that affect specific objects, such as warehouses, databases, schemas, and tables

All parameters are initially set with default values, which can be subsequently overridden at each of the levels. A Snowflake administrator with sufficient privileges can override parameter values at the account level. Users can override the parameter values at their own session level. Object parameter values can be overridden by users who have sufficient privileges on the object.

To view a list of the parameters and their values in your account, as well as their default values, execute the `SHOW PARAMETERS` command with the `ACCOUNT` keyword:

```
show parameters in account;
```

A subset of the output of this command is shown in Table 4-1.

Table 4-1. *Subset of Account Parameters with Their Current and Default Values*

Key	Value	Default	Type
GEOGRAPHY_OUTPUT_FORMAT	GeoJSON	GeoJSON	STRING
JDBC_TREAT_DECIMAL_AS_INT	TRUE	TRUE	BOOLEAN
JDBC_USE_SESSION_TIMEZONE	TRUE	TRUE	BOOLEAN
JSON_INDENT	2	2	NUMBER
LANGUAGE	en	en	STRING
LOCK_TIMEOUT	43200	43200	NUMBER

If you are using a 30-day free Snowflake trial account, you don't need to override your parameter values. One parameter that you might want to review is the TIMEZONE parameter, which is initially set as America/Los Angeles. You can override it to your local time zone if it is different. You can do this at the account level using the ACCOUNTADMIN role or at the session level using your current role.

To override a parameter value for your session, use the ALTER SESSION command. For example, to set the value of the TIMEZONE parameter to Europe/London, use the following command:

```
alter session set TIMEZONE = 'Europe/London';
```

To view a list of parameters and their values in your session, as well as their default values, execute the SHOW PARAMETERS command with the SESSION keyword:

```
show parameters in session;
```

A subset of the output of this command is shown in Table 4-2.

Table 4-2. *Subset of Session Parameters with Their Current and Default Values*

Key	Value	Default	Level
DATE_INPUT_FORMAT	AUTO	AUTO	
LANGUAGE	en	en	
TIMEZONE	Europe/London	America/Los_Angeles	SESSION
TIME_INPUT_FORMAT	AUTO	AUTO	
TRACE_LEVEL	OFF	OFF	
USE_CACHED_RESULT	TRUE	TRUE	

Network Security and Policies

By default, users are allowed to connect to Snowflake from any IP address. For greater security, a security administrator can create a *network policy* to allow or deny access to specified IP addresses. This can be accomplished by creating IP allowed lists or IP blocked lists or both.

To create network policies, a user must have any of the following privileges:

- A `SECURITYADMIN` role
- A role higher than `SECURITYADMIN`
- A role with the global `CREATE NETWORK POLICY` privilege

Network policies can be assigned to a Snowflake account or to a particular user. Although multiple network policies can be created, only a single network policy can be assigned to an account or a user at a time.

Once the network policy is associated with an account or a user, Snowflake restricts access to that account or that user based on the allowed

IP address list or the blocked IP address list. A user who attempts to log in from an IP address that is on the IP blocked list or is not on the IP allowed list is denied access. In addition, when a network policy is assigned to an account or a user while they are already connected to Snowflake, they cannot execute further statements if their IP address doesn't comply with the network policy.

Note When a network policy includes values for the allowed IP address list and the blocked IP address list, Snowflake checks the blocked IP address list first.

As a Snowflake user without administrative privileges, you will not be able to create, assign, or manage network policies. But you can view whether a network policy is set on your account using the `SHOW PARAMETERS` command:

```
show parameters like 'network_policy' in account;
```

You can also view whether a network policy is set at your user level using the following command, where you provide your username that you use to log in to Snowflake:

```
show parameters like 'network_policy' in user <yourusername>;
```

When even more security is needed, Snowflake Business Critical edition (or higher) supports the AWS PrivateLink service. This is an AWS service for creating private VPC (virtual private cloud) endpoints that allow direct, secure connectivity between AWS VPCs and the Snowflake VPC without traversing the public Internet. To take advantage of this functionality, your Snowflake account must be in the same AWS region as the AWS VPCs.

User Authentication

Authentication is the process of identifying users and validating who they claim to be. Snowflake provides several secure authentication features, described in more detail in the following sections:

- Multi-factor authentication (MFA)
- Federated authentication
- Single sign-on (SSO)

Snowflake administrators who have the `ACCOUNTADMIN`, `SYSADMIN`, or `SECURITYADMIN` role set up federated authentication. This chapter does not describe the administrative tasks in detail because they are tested on the Snowflake advanced exams. However, it does describe the authentication features from a user perspective, which should be familiar to Snowflake users and are included in the SnowPro Core exam.

Multi-Factor Authentication

Snowflake supports multi-factor authentication (abbreviated as MFA) to enable increased login security for users connecting to Snowflake. MFA is provided by Duo Security and is included as a Snowflake service.

You don't have to sign up with Duo Security in order to use MFA with Snowflake because the support is seamless. You only have to install the Duo Mobile application on one of the supported platforms, including iOS, Android, Windows, and others.

You can self-enroll in MFA through the web interface. Once your enrollment is successful, you then log in to Snowflake using the web interface or any of the clients. After you enter your user credentials (login name and password), you have to also provide the authentication passcode generated by the Duo Mobile application.

Caution When you practice enrolling in MFA using your 30-day free Snowflake trial account, create a new user in your account and practice with this new user. This is just a precaution in case something goes wrong, so that you don't lock yourself out of your own account.

To create a new user named FIRSTUSER to practice MFA, use the following command (creating users and assigning roles is explained in more detail later in this chapter):

```
create user FirstUser
  first_name = 'First'
  last_name = 'User'
  password = 'Password123!';
```

Once the user is created, log in to your Snowflake account as FIRSTUSER. To enroll in MFA, click the down-arrow at the top of the left navigation bar in Snowsight to reveal the user menu, as shown in Figure 4-1; then choose the Profile option.

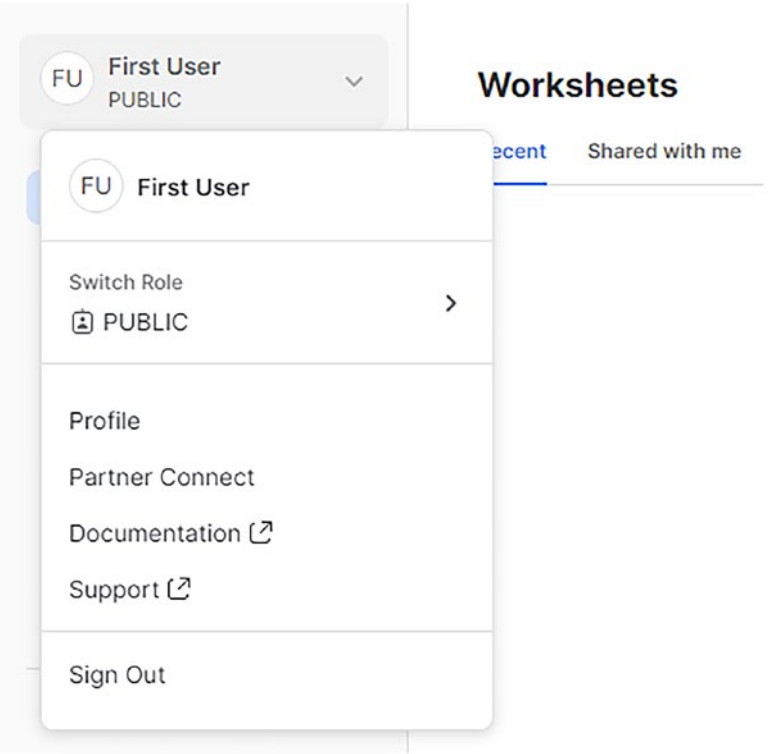

Figure 4-1. *User menu in Snowsight*

In the window that appears, click the Enroll button next to the Multi-factor Authentication section near the bottom of the window, as shown in Figure 4-2.

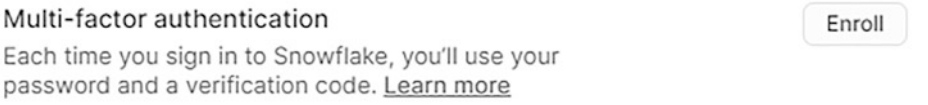

Figure 4-2. *Multi-factor authentication section in the Profile window*

In the next screen, shown in Figure 4-3, click the Start Setup button to start the Duo setup process.

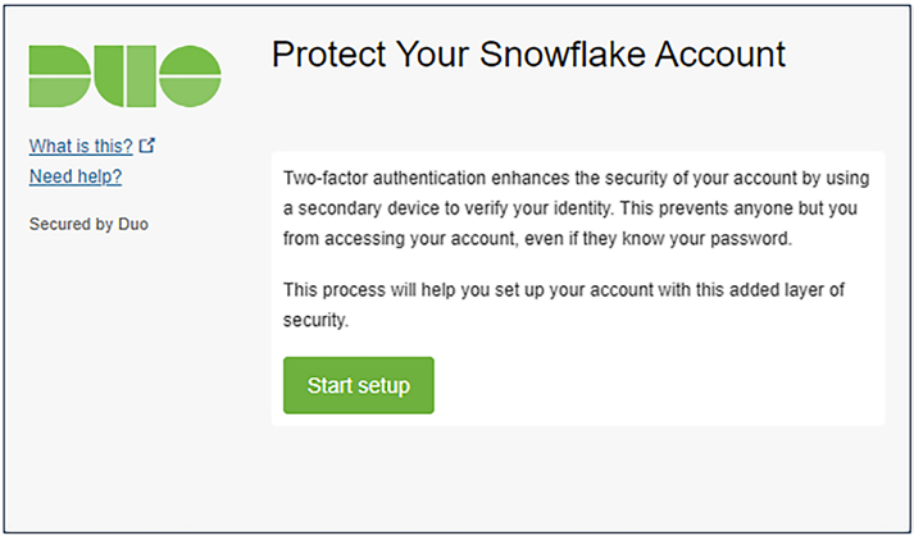

Figure 4-3. *Start Duo setup*

On the following screen, shown in Figure 4-4, choose the type of device that you will use for multi-factor authentication.

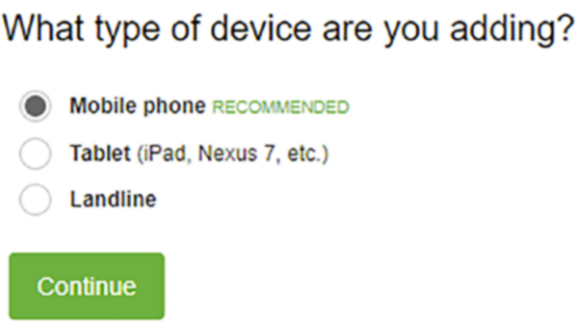

Figure 4-4. *Choose the type of device you will use for MFA*

Depending on the type of device you choose, the install process will guide you through the next steps.

If you choose mobile phone as the type of device, which is the recommended type of device, you will be asked to provide the mobile

phone number and the operating system of your mobile phone. Then you will install the Duo Mobile app. When you open the Duo Mobile app on your mobile phone, you will set up a new account by scanning the QR code that has been provided in the setup process. Alternatively, you can have an activation link sent to your email address during the setup process.

Once you have successfully set up your MFA, the next time you log in to your Snowflake account, you will enter your username and password as usual. Then you will see an additional window, as shown in Figure 4-5. This window will ask you to either provide a passcode that is generated by your Duo Mobile app or to send a push request to the Duo Mobile app where you can approve the login.

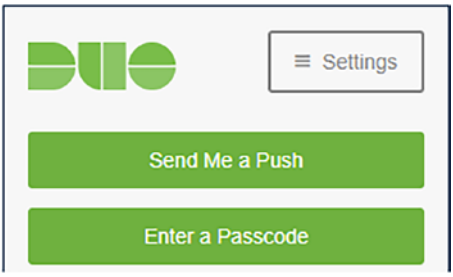

Figure 4-5. *Login screen for MFA*

You will not be able to unenroll from MFA after you have enrolled. To unenroll, you must ask your Snowflake administrator to unenroll you. If you are using the 30-day free Snowflake trial account, you can unroll the `FIRSTUSER` from MFA by logging into your Snowflake account using your initial credentials (the same credentials that were used to create `FIRSTUSER`) and executing this statement:

```
alter user FirstUser set disable_mfa = true;
```

Alternatively, if you are experiencing any technical issues with MFA, you can ask your Snowflake administrator to allow you to temporarily

bypass MFA and log in directly for a limited time. For example, to bypass MFA for ten minutes, you can execute the following statement:

```
alter user FirstUser set mins_to_bypass_mfa = 10;
```

Federated Authentication

In a federated environment, user authentication is separated from user access using an entity that provides independent authentication of user credentials. Snowflake supports authentication through an external identity provider. Once users are authenticated by the identity provider, they can securely initiate Snowflake sessions without having to log in to Snowflake each time.

Snowflake supports most SAML 2.0-compliant vendors as identity providers, including:

- Google G Suite
- Microsoft AzureAD (Active Directory)
- OneLogin
- Ping Identity PingOne

The following vendors provide native Snowflake support for federated authentication:

- Okta
- Microsoft ADFS (Active Directory Federation Services)

For example, when federated authentication is configured in a Snowflake account, the login screen in the Snowflake web interface looks similar to what is shown in Figure 4-6. There is an additional option, called Sign in Using AzureAD, because federated authentication was set up using Microsoft AzureAD as the identity provider.

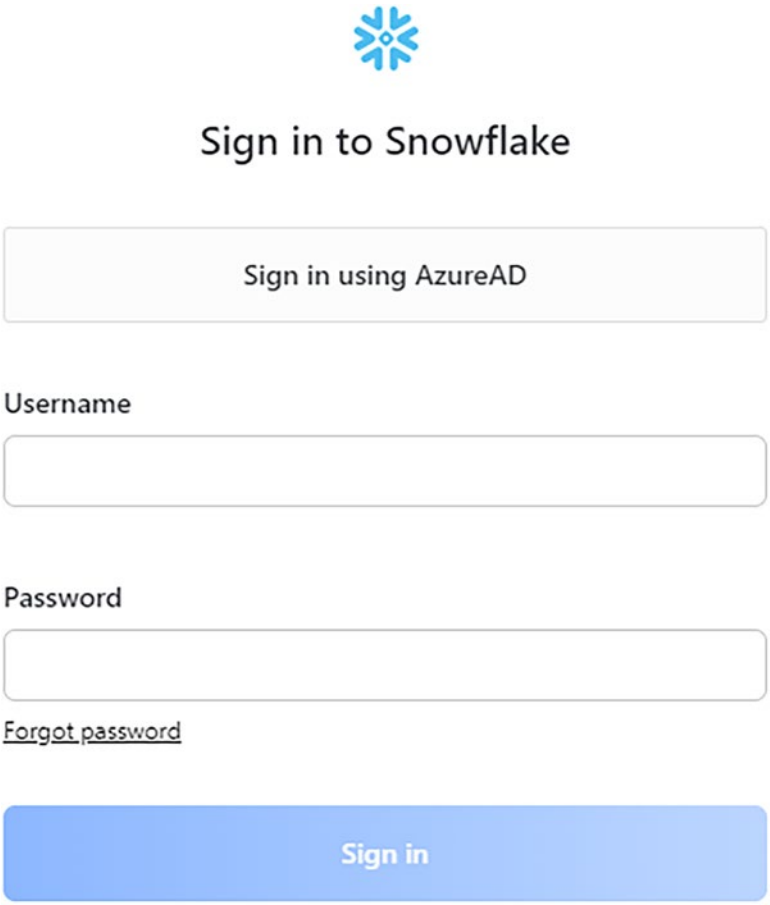

Figure 4-6. *Snowflake login screen using AzureAD for federated authentication*

Instead of typing their Username and Password, a user connects to Snowflake by clicking the identity provider option on the login page. After that, there are two possibilities to connect to Snowflake:

- If the user has already been authenticated by the identity provider, they are immediately granted access to Snowflake.

- If the user has not yet been authenticated by the identity provider, they are taken to the identity provider interface where they authenticate, after which they are granted access to Snowflake.

Instead of going to the Snowflake web interface, the user can choose to initiate their Snowflake session from the interface provided by the identity provider. In the identity provider user interface, the user clicks the Snowflake application icon (when using Okta or Microsoft ADFS) or the custom Snowflake application that has been defined in the identity provider (when using another identity provider). This will initiate a session in Snowflake and display the Snowflake web interface.

Depending on the identity provider, users have two possibilities to log out of Snowflake:

- **Standard logout.** A user must log out of both Snowflake and the identity provider. When a user logs out of just a single Snowflake session, any of their other Snowflake sessions remain connected. They can still initiate new Snowflake sessions if they remain logged in the identity provider.
- **Global logout.** When a user logs out of the identity provider, all of their Snowflake sessions log out as well. With a web-based identity provider such as Okta, closing the browser does not necessarily end the identity provider session. As long as the identity provider session is active, the user can still access Snowflake.

With respect to session timeout, there are two scenarios depending on which session timed out:

- **Snowflake timeout.** When a user is logged into Snowflake via an identity provider and their Snowflake

session expires due to inactivity, they will be asked to authenticate via the identity provider again to continue to work in the expired session.

- **Identity provider timeout.** When a user's session in the identity provider reaches the defined timeout period, their current Snowflake sessions will not be affected. However, if the user wants to initiate additional Snowflake sessions, they will have to log into the identity provider again.

Session Policies

The default timeout period of a Snowflake session is four hours. This timeout period can be overridden by creating a session policy that defines the idle session timeout period in minutes. Similarly, as with network policies, session policies can be assigned to an account or to a user. A session policy assigned to a user takes precedence over a session policy assigned to an account.

To create a session policy, use the `CREATE SESSION POLICY` command. This command takes two optional parameters that can be set depending on whether the session policy will apply to the web user interface or to any of the Snowflake clients:

- `SESSION_UI_IDLE_TIMEOUT_MINS`. Applies to the web user interface.
- `SESSION_IDLE_TIMEOUT_MINS`. Applies to Snowflake clients.

Both parameters can be set to a value between 5 and 240 minutes. Whenever a session policy has not been set, Snowflake will use the default value of 240 minutes (4 hours). Once the session expires, the user will have to authenticate again.

Access Control

Snowflake administrators can implement detailed access control that allows or restricts access to objects for specific users or roles. They can also define what operations can be performed on objects to which users have access and they can define which users are allowed to create, modify, and grant access control policies.

Access control in Snowflake can be defined as either discretionary or role-based:

- **Discretionary Access Control (DAC).** In this model, each object has an owner, and that owner can grant or revoke access to the object to other users or roles.
- **Role-based Access Control (RBAC).** In this model, each object is owned by a role, and access is granted to users by assigning the respective role.

Before delving into Snowflake access control in more detail, it's important to understand a few key components:

- **Securable object.** Any Snowflake object, such as a database, schema, table, view, stored procedure, and so on, to which access can be granted.
- **User.** Either a person or a client program that can log into Snowflake and to whom access to securable objects can be granted.
- **Role.** A set of privileges can be granted to a role and, in turn, a role can be assigned to a user or to another role.
- **Privilege.** One or more allowed types of access to a securable objects, such as the privilege to read or the privilege to modify and so on.

Securable Objects

Snowflake securable objects are organized as a hierarchy of containers. The topmost container is the organization. The next level container is one or more accounts within the organization. Each account contains objects such as databases, users, roles, warehouses, and more. Databases contain schemas and schemas contain securable objects such as tables, views, stages, stored procedures, user-defined functions, and other objects. This hierarchy of objects and containers is illustrated in Figure 4-7.

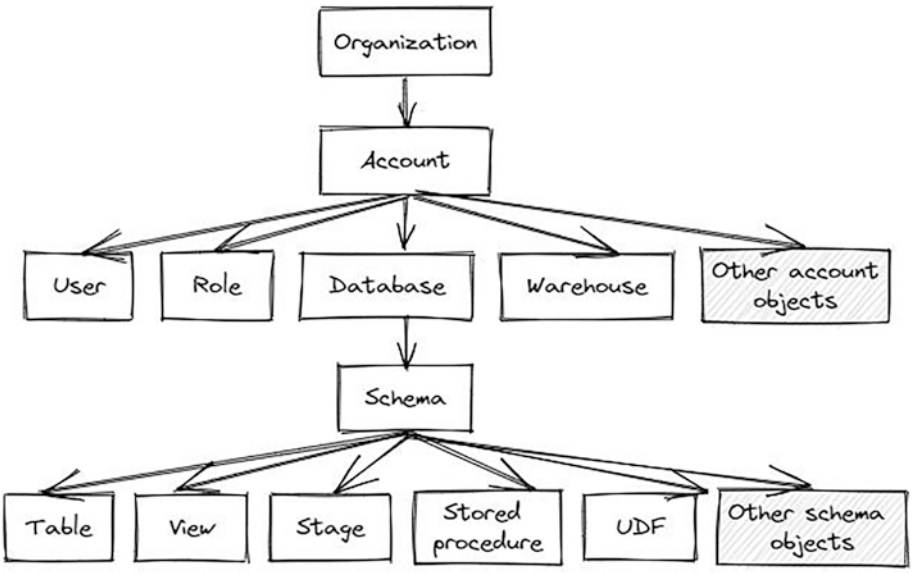

Figure 4-7. *Hierarchy of Snowflake objects and containers*

A securable object is owned by the role that was used to create the object. This role has the `OWNERSHIP` privilege on the object. The same role can be granted to other users, giving them the same level of control over the object as if they had created the object.

In Snowflake, object ownership can be implemented in two ways, by using different types of schemas:

- **Regular schema.** The role that owns the objects has all privileges related to the object, including the ability to grant or revoke privileges on the object to other users or roles as well as to transfer ownership.
- **Managed access schema.** All objects in a managed access schema are owned by the schema itself. There are no individual object owners. Privileges on objects in a managed access schema can be granted either by the schema owner or a role that has the `MANAGE GRANTS` privilege.

Users

User administrators can create and manage Snowflake users, grant and revoke privileges, set default parameters, and reset passwords if needed.

A user administrator is a user who has one of the following privileges:

- A `USERADMIN` role
- A role higher than `USERADMIN`
- A role with the global `CREATE USER` privilege

To create a new user, a user administrator can use the web user interface in Snowsight or the `CREATE USER` command in SQL.

While creating a new user, the Snowflake administrator must also consider the role that will be assigned to the user. Roles can be assigned when the user is created, or they can be granted at a later time. If a user has a default role assigned, this role is used when the user logs into Snowflake and initiates a session.

Note When you assign a default role to a user, this does not immediately grant them the role. You must also explicitly grant the role to the user.

Since you haven't created any roles yet (roles are described in the next section), you will create a new user without assigning a role at this time. To create a new user named SECONDUSER, execute the following SQL command:

```
create user SecondUser
  first_name = 'Second'
  last_name = 'User'
  display_name = 'Second User'
  password = 'Password456!'
  email = 'seconduser@email.com';
```

You can modify user properties using the ALTER USER command. For example, to change the email address of the user, use this command:

```
alter user seconduser set email = 'user2@email.com';
```

You can view information about a user using the DESCRIBE USER command. For example, to view information about the user named SECONDUSER, use this command:

```
describe user seconduser;
```

Partial output of the DESCRIBE USER command is shown in Table 4-3.

Table 4-3. *Partial Output of the DESCRIBE USER Command*

Property	Value	Default	Description
NAME	SECONDUSER	null	Name
DISPLAY_NAME	Second User	null	Display name of the object
LOGIN_NAME	SECONDUSER	null	Login name of the user
FIRST_NAME	Second	null	First name of the user
LAST_NAME	User	null	Last name of the user
EMAIL	user2@email.com	null	Email address of the user
PASSWORD	********	null	Password of the user
DISABLED	FALSE	FALSE	Whether the user is disabled

User administrators can view a list of all users in the account using the `SHOW USERS` command. They can also drop users by executing the `DROP USER` command. For example, to drop the user called `FIRSTUSER` that you created earlier for testing MFA, an administrator would execute the following statement:

```
drop user firstuser;
```

Roles

Privileges on securable objects can be granted to and revoked from roles. Roles are then assigned to users to control what actions they may perform on the securable objects.

More than one role can be granted to a user. In this case, the user will be able to switch roles depending on which set of privileges they need to perform an action.

A hierarchy of roles can be created by granting roles to other roles. A role that is at a higher level of the hierarchy inherits all privileges granted to roles at a lower level of the hierarchy.

Snowflake provides a set of system-defined roles as well as functionality for defining a hierarchy of custom roles.

System-Defined Roles

Every Snowflake account includes system-defined roles with privileges required for account management. These built-in roles cannot be dropped and the privileges that are granted to these roles by Snowflake cannot be revoked.

With the exception of `ORGADMIN`, the system-defined roles are structured according to a hierarchy, in which each role inherits the privileges of all the roles below the current role. `ORGADMIN` is a separate system role that is required to perform tasks at the organization level.

The system-defined roles and their hierarchy are depicted in Figure 4-8 and described in more detail in the following paragraphs.

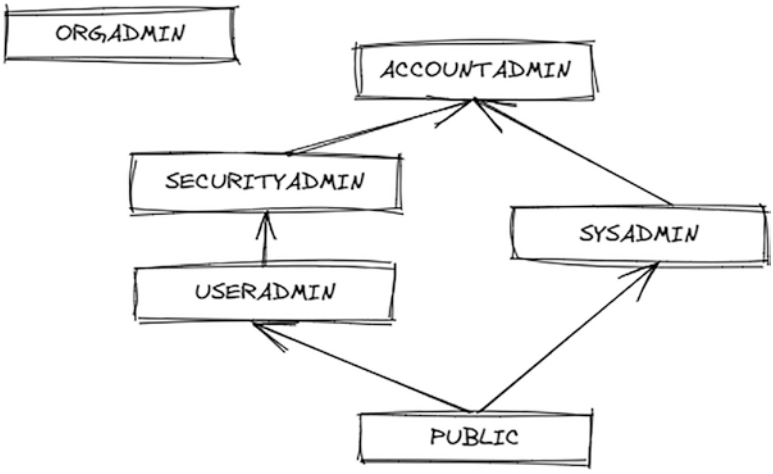

Figure 4-8. *System-defined roles and their hierarchy*

The ORGADMIN or the Organization Administrator role is used to perform tasks at the organization level. This role can create and manage accounts within the organization, view usage information across the organization, and other related tasks.

The ACCOUNTADMIN or the Account Administrator role is the most powerful role in a Snowflake account. It is used to perform account administration tasks, set account parameters, view account usage information, and other related tasks.

The SECURITYADMIN or the Security Administrator role is used to create, monitor, and manage users and roles. This role has the MANAGE GRANTS privilege, which allows it to grant or revoke any privilege.

The USERADMIN or the User and Role Administrator role is used for managing users and roles. This role has the CREATE USER and CREATE ROLE privileges, which allow it to create users and roles in the account.

The SYSADMIN or the System Administrator role is used to create virtual warehouses, databases, schemas, tables, and other objects in an account.

The PUBLIC role is granted to every user and every role in an account. Any securable objects owned by the PUBLIC role are accessible to every other user and role in the account.

Custom Roles

The USERADMIN role 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 to another role in order to take effect.

Snowflake recommends creating a hierarchy of custom roles and granting the topmost custom role to the SYSADMIN system-defined role. This will allow system administrators to access and manage all objects in the account.

Granting and Revoking Privileges

In addition to granting and revoking privileges on existing objects in a schema, Snowflake supports defining privileges on future objects that will be created in a schema.

Privileges are managed using the GRANT and REVOKE commands. Let's walk through an example. You must be logged into Snowflake with a user who has sufficient privileges to create users and roles, as well as the ability to grant roles. If you are using a 30-day free Snowflake trial account, you should have these privileges.

First, use the USERADMIN role to create a custom role named ANALYST:

```
use role USERADMIN;
create role ANALYST;
```

Then switch to the SYSADMIN role, which you will use to create a database named REPORTING, a schema named MARKETDATA, and a table named CUSTOMER:

```
use role SYSADMIN;
create database REPORTING;
create schema MARKETDATA;
create table CUSTOMER (
    c_custkey NUMBER(38,0) NOT NULL,
    c_name VARCHAR(40),
    c_address VARCHAR(255),
    c_country VARCHAR(3),
    c_mktsegment VARCHAR(255)
);
```

Now you will grant read-only access on the CUSTOMER table to the ANALYST role. In addition to granting access on the table itself, access must also be granted to securable objects in the hierarchy above the current

object. This means that privileges that allow the role to use the schema and the database where the table is contained must also be granted to the role.

You are still using the SYSADMIN role because this role owns the securable objects and is allowed to grant privileges:

```
grant select on table CUSTOMER to role ANALYST;
grant usage on schema MARKETDATA to role ANALYST;
grant usage on database REPORTING to role ANALYST;
```

Finally, switching back to the USERADMIN role, you can grant the newly created custom role ANALYST to the SECONDUSER user that you created previously:

```
use role USERADMIN;
grant role ANALYST to user SECONDUSER;
```

To verify that the scenario is working as expected, log out of Snowflake and log back in using the SECONDUSER credentials. You must check whether the ANALYST role is chosen as the current role in the user session; if not, you have to choose it in the Role and Warehouse drop-down list at the top right in the Snowsight user interface. Then you can expand the Databases tab in the left pane to verify that you can see the REPORTING database, the MARKETDATA schema, and the CUSTOMER table, as depicted in Figure 4-9. You can also execute a query to select data from the CUSTOMER table, on which the user should have read-only access.

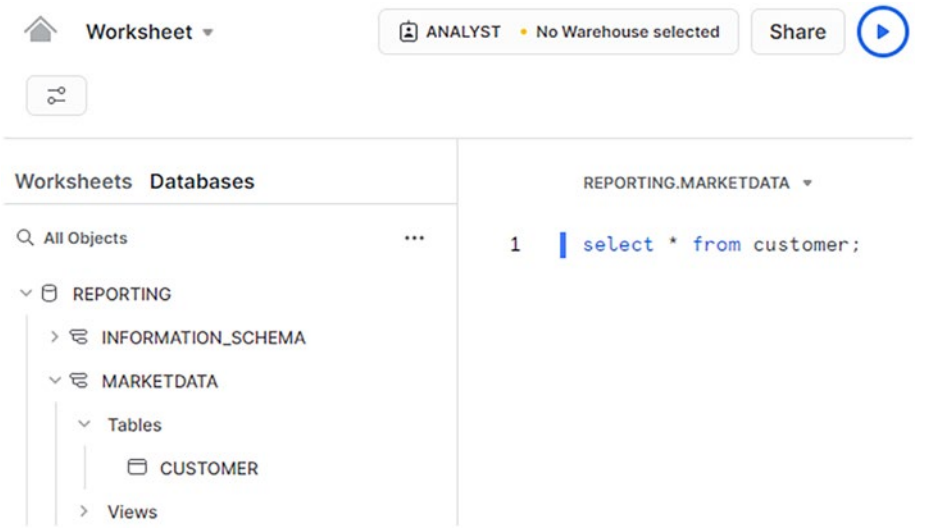

Figure 4-9. *Expanding the Databases tab using the ANALYST role*

Let's create a second role named DATAENGINEER that will have full access to table CUSTOMER and grant this role to SECONDUSER as well. Log into your Snowflake account using your initial credentials and execute these commands:

```
use role USERADMIN;
create role DATAENGINEER;

use role SYSADMIN;
grant all on table CUSTOMER to role DATAENGINEER;
grant usage on schema MARKETDATA to role DATAENGINEER;
grant usage on database REPORTING to role DATAENGINEER;

use role USERADMIN;
grant role DATAENGINEER to user SECONDUSER;
```

Role Hierarchy and Privilege Inheritance

When a user logs into the Snowflake web interface or a program connects to Snowflake via a connector, a session is initiated. The current role in the session is determined by Snowflake according to the following order of precedence:

- If the connection information included the name of a role and that role has been granted to the user, this role is used in the session as the current role.
- If the connection information did not include the name of a role and the user has a default role assigned, the default role is used.
- If the connection information did not include the name of a role and the user has no default role assigned, the `PUBLIC` role is used.

The current role in a session is also referred to as the *primary role*. In addition to the primary role, a user can also activate one or more secondary roles in the session by executing the `USE SECONDARY ROLES` command. A default set of secondary roles can also be defined for a user, similarly as their default primary role using the `DEFAULT_SECONDARY_ROLES` keyword with the `CREATE USER` or `ALTER USER` commands.

When a user has more than one role activated in a session, they can execute SQL statements using the combined privileges of the primary and all secondary roles. These are applied differently whether the user is creating objects or performing other actions on objects:

- The primary role is used to determine whether the user has privileges to execute `CREATE` statements for Snowflake objects. This role will also be set as the owner of any object that the user creates.

- To execute SQL statements other than `CREATE`, Snowflake will allow execution if the user has sufficient privileges granted to the primary or to any of their secondary roles.

Secondary roles are useful in large organizations that have a large number of defined custom roles, such as separate sets of roles for each database. When a developer needs privileges to execute SQL statements that access more than one database in the same query, for example to copy data from one database to another, they can activate a primary and a secondary role, one that grants them access to each of the databases in their current session.

Setting Primary and Secondary Roles

A user can set their primary role in a session using the Roles and Warehouses drop-down list at the top right in Snowsight, as shown in Figure 4-10.

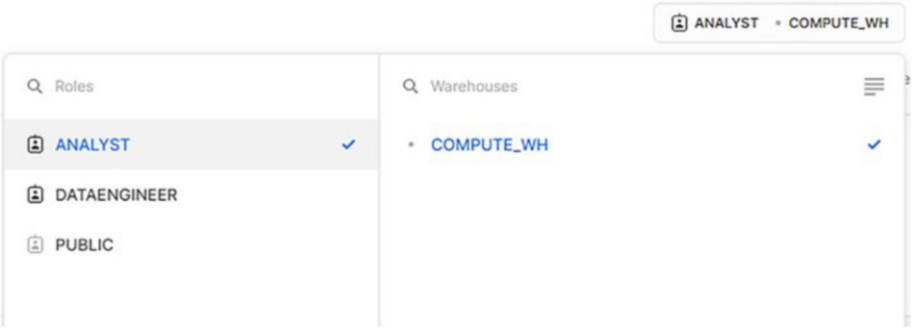

Figure 4-10. *Roles and Warehouses drop-down list in Snowsight*

Alternatively, a user can set their primary role by executing the `USE ROLE` command. For example, when logged into Snowflake as `SECONDUSER`, you use the following command to set the primary role to `ANALYST`:

```
use role ANALYST;
```

To set one or more secondary roles, the `USE SECONDARY ROLES` command can be executed. For example, to set the secondary role to `DATAENGINEER`, use this command:

```
use secondary roles DATAENGINEER;
```

A user can view their current secondary roles using the `CURRENT_SECONDARY_ROLES` function:

```
select current_secondary_roles();
```

The result of this statement is shown in Figure 4-11.

CURRENT_SECONDARY_ROLES()
{"roles":"DATAENGINEER","value":"DATAENGINEER"}

Figure 4-11. *Current secondary roles in a user session*

Summary

This chapter addressed various Snowflake security topics. It described account identifiers and account parameters. It introduced network policies that allow or deny access to Snowflake from specific IP addresses. It also provided an overview of user authentication methods. Finally, it discussed granting privileges to users through roles.

The next chapter talks about Snowflake storage and performance concepts.

CHAPTER 5

Storage and Performance Concepts

Snowflake provides unique data storage features such as micro-partitions, clustering, and search optimization that enable fast query execution on tables with large volumes of data. Much of this optimization is taken care of by Snowflake behind the scenes, away from the users. However, as a Snowflake user, you must understand how this optimization works so that you can apply the most efficient query techniques in the right circumstances.

This chapter describes micro-partitions as the physical storage units in Snowflake. You will then learn about data clustering and creating clustering keys on tables. Understanding how data is stored physically in Snowflake and how and when storage costs are incurred is an important topic that the chapter addresses by reviewing the storage monitoring functionality. Then it discusses the search optimization service and how it can be useful in optimizing queries.

Finally, you learn how to use the Query Profile tool to view query execution information such as data caching, data spilling, micro-partition pruning, and other statistics. The chapter also describes query history, which allows you to find details about queries that executed in the past.

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_5

Micro-partitions

All data in Snowflake tables is stored in contiguous units of storage called micro-partitions. Each micro-partition contains anywhere between 50MB and 500MB of uncompressed data. The actual size of the stored data is smaller because data is always compressed in Snowflake.

Snowflake automatically creates micro-partitions while data is inserted into tables. Data is often ordered along natural dimensions such as date, time, or geographic origin. This natural ordering is useful when micro-partitions are created so that rows that share the same value of a dimension are co-located in micro-partitions whenever possible. Individual micro-partitions contain data that is organized in columnar fashion.

The benefits of Snowflake micro-partitions include:

- Micro-partitions are created automatically, and users don't need to maintain them.
- Micro-partitions are relatively small as compared to the full table, which enables pruning for better query performance.
- Values can overlap between micro-partitions, which reduces skew in the amount of data stored in each micro-partition.
- Micro-partitions are organized in a columnar fashion, which allows only the columns that are used in a query to be accessed.
- Storage in micro-partitions is optimized by compressing columns individually.
- Before queries are executed, Snowflake looks at micro-partition metadata to determine how the metadata can be utilized. For example, to delete all rows from a table, only the metadata is updated.

To see how micro-partitioning works, this chapter will work with a small-scale conceptual representation utilized by Snowflake. It will use a table named CUST with four columns that are sorted by date, as shown in Figure 5-1.

CUST_KEY	CUST_NAME	DEPT_KEY	DATE
116	680e	18	Sep 7
277	g187	20	Sep 7
595	f01d	21	Sep 8
239	f372	0	Sep 8
675	140a	23	Sep 8
129	1e2c	21	Sep 9
643	91fe	22	Sep 11
149	db29	12	Sep 11
367	753c	23	Sep 12

Figure 5-1. *Sample CUST table*

The sample CUST table consists of nine rows, ordered by date, that are physically stored across three micro-partitions, with the rows divided equally between each micro-partition. The first three rows are stored in the first micro-partition, the next three rows in the second micro-partition, and the last three rows in the third micro-partition, as represented in Figure 5-2.

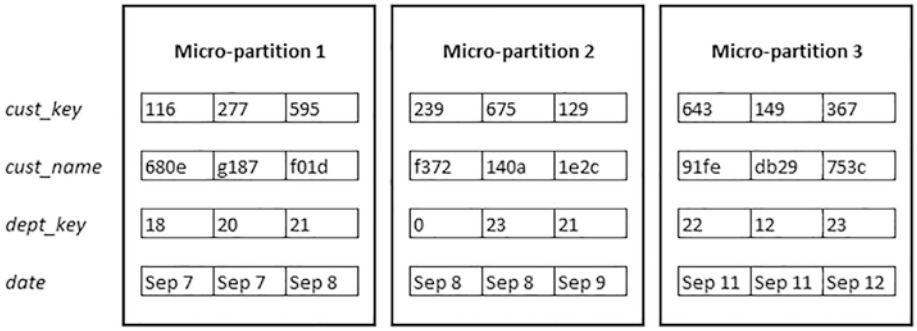

Figure 5-2. *Sample data stored in micro-partitions, ordered by date*

Snowflake keeps track of micro-partition metadata, including:

- The range of values for each of the columns in the micro-partition
- The number of distinct values
- Additional properties used for optimization and query execution

Because data in micro-partitions is sorted and stored by column, Snowflake can optimize query performance by pruning micro-partitions in different ways:

- Prune micro-partitions that are not used
- Prune columns that are not used in the remaining micro-partitions

For example, you might execute a query on the sample `CUST` table and filter for a specific date using this command:

```
select CUST_NAME from CUST where DATE = 'Sep 8';
```

Snowflake uses micro-partition metadata to know that it needs to read data only from micro-partitions 1 and 2 because they contain records for the chosen date (`'Sep 8'`), but not micro-partition 3. Additionally, it only

needs to read the columns that participate in the query, in this case only the DATE and CUST_NAME columns, not any other columns. This approach is the basis for highly selective pruning when querying very large tables, which can be comprised of millions of micro-partitions.

Data 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.

With respect to clustering information, Snowflake tracks the following metadata:

- The total number of micro-partitions that are used to store data from the table
- The number of micro-partitions that contain overlapping values
- The depth of the overlapping micro-partitions

In the previous example from Figure 5-2, you see that rows where the DATE column contains the value 'Sep 8' appear in micro-partitions 1 and 2, indicating that these two micro-partitions overlap with respect to the DATE column.

The *clustering depth* for a set of columns in a table is the average depth of the overlapping micro-partitions. Since fewer overlaps mean that clustering is better, a lower clustering depth indicates that the table is clustered well. On the other hand, when the clustering depth is high, and if the table is large, the table might benefit from adding clustering explicitly.

To view the clustering metadata for a table, Snowflake provides the SYSTEM$CLUSTERING_DEPTH and SYSTEM$CLUSTERING_INFORMATION system

functions. To view the clustering depth of the CUST table, execute the following command:

```
select SYSTEM$CLUSTERING_DEPTH ('cust', '(cust_key)');
```

The first parameter is the name of the table, and the second parameter is a list of columns for which the command will return the clustering depth.

Note The second parameter of the SYSTEM$CLUSTERING_DEPTH function is a list of columns; therefore, you must list the columns in parentheses, even if you are listing only one column.

To view the clustering information of the CUST table, execute the following command:

```
select SYSTEM$CLUSTERING_INFORMATION ('cust', '(cust_key)');
```

The result of this command is a JSON string that contains key/value pairs, as described in Table 5-1.

Table 5-1. *Clustering Information JSON Keys and Descriptions*

Key	Description
`cluster_by_keys`	Columns for which clustering information is provided.
`Notes`	Suggestions on making clustering more efficient, if any.
`total_partition_count`	Total number of micro-partitions in the table.
`total_constant_partition_count`	Total number of micro-partitions for which the values of the specified columns don't overlap significantly.
`average_overlaps`	Average number of overlaps for each micro-partition in the table.
`average_depth`	Same value as `SYSTEM$CLUSTERING_DEPTH`.
`partition_depth_histogram`	A histogram with buckets from 0 to 16 that represent the distribution of clustering depth for each micro-partition in the table. Additional buckets are also displayed, but with larger increments.

Clustered Tables

When data in tables is not clustered well, but users need better query performance that would benefit from micro-partition pruning, data in tables can be clustered. One way to do this is to select data from the original table, perform a sort on the columns that would benefit from clustering, and insert the resulting data into a new table. This manual process would have to be repeated over time, as the clustering degrades due to DML operations on table data.

To avoid having to do this process manually, Snowflake supports designating one or more table columns or expressions as a *clustering key* for the table. When you do this, Snowflake will perform the sorting and reorganizing process behind the scenes without user intervention.

Before creating a clustering key on a table, consider that clustering incurs additional cost due to initial clustering and periodic maintenance of the clustering. Therefore, you should use clustering only when:

- Query performance is more important than clustering cost
- Savings due to improved query performance are greater than clustering costs

Some additional criteria that can help you decide whether a table should be clustered include:

- Queries using data from the table have degraded over time.
- The table has a large clustering depth.
- The table is queried frequently and does not change frequently, which means that a large number of queries will benefit from improved performance while clustering maintenance will be low.

In addition to clustering tables, you can also cluster materialized views.

Clustering Keys

A *clustering key* is a subset of columns in a table or an expression on a table that is designated to co-locate the data in the table in the same micro-partitions according to the clustering key.

To continue with the example using the CUST table in Figure 5-1, where the data is ordered by date, say that users typically query this table by filtering on department first, and then by date. Clustering by department would help improve query performance in this situation.

Figure 5-3 illustrates the same data as Figure 5-2, only this time the data in micro-partitions are ordered by department (the DEPT_KEY column). You can see that the micro-partitions are organized so that rows with different values of DEPT_KEY never overlap across micro-partitions, allowing highly selective pruning.

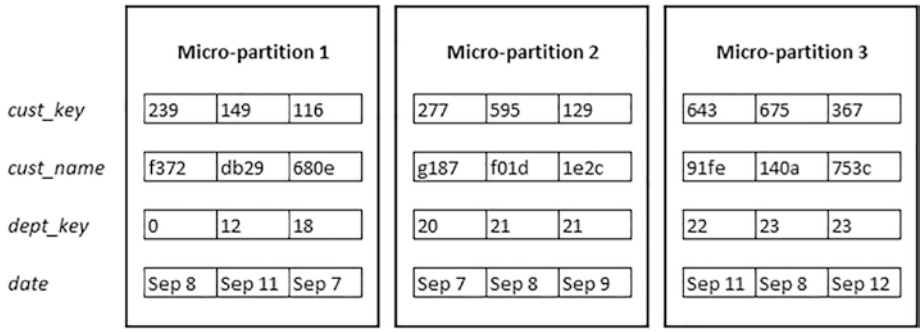

Figure 5-3. *Sample data stored in micro-partitions, ordered by DEPT_KEY*

A clustering key can be defined when you create the table, using the CREATE TABLE command. For example, to create table named CUST_CLUSTERED with the DEPT_KEY column as the clustering column, you would execute the following command:

```
create table CUST_CLUSTERED (
    cust_key NUMBER(38,0) NOT NULL,
    cust_name VARCHAR(100),
    dept_key NUMBER,
    date DATE
)
cluster by (dept_key);
```

A clustering key can also be added to a table at a later time using the `ALTER TABLE` command. For example:

```
alter table CUST cluster by (dept_key);
```

You can also alter or drop the clustering key. After a clustering key has been defined on a table, all future maintenance with respect to clustering is performed automatically by Snowflake.

After you define a clustering key for a table, micro-partitions are usually not reorganized immediately. Snowflake performs clustering in the background and this may take some time to complete. When only a small amount of data in a table has changed, Snowflake might decide not to start reclustering immediately if it estimates that the benefit of reclustering does not justify the cost.

To decide whether to apply clustering on a table, use the following criteria:

- The table contains a large number of micro-partitions so that pruning will be significant.
- Typical queries that are executed on the table select a small portion of rows, which in turn means that a small number of micro-partitions will be retrieved. Additionally, queries that sort data on the clustering key will also perform well if the data is already clustered.
- Many queries filter by the same clustering key—for example, by a date or a time period.

A clustering key can be made up of one or more columns or expressions, such as a substring of a column. Snowflake recommends that you prioritize keys according to how frequently they are used in

filters. If queries typically filter by two dimensions—for example, first by department and then by date—clustering on both columns will most likely result in better performance.

The number of distinct values in a column is also an important factor when selecting a clustering key. A good candidate for a clustering key is a column that has:

- A large enough number of distinct values to enable micro-partition pruning.
- A small enough number of distinct values to allow Snowflake to co-locate rows with the same key in the same micro-partitions.

Note When clustering on a character column, only the first five or six bytes of data are used for clustering data in micro-partitions.

Reclustering

Once a table has a clustering key defined, Snowflake performs the initial clustering as well as periodic reclustering to ensure that the table is well clustered, even after DML operations on table data would cause the clustering to degrade. Reclustering is performed by a Snowflake service on a regular basis.

You can suspend and resume automatic clustering for a clustered table using the `SUSPEND RECLUSTER` and `RESUME RECLUSTER` keywords with the `ALTER TABLE` command. For example, to suspend reclustering, use this command:

```
alter table CUST suspend recluster;
```

When you suspend automatic clustering for a table, Snowflake will not automatically recluster the table, even if clustering has degraded as a result of DML operations.

You can drop the clustering key on a clustered table. When you drop the clustering key, the table will not be reclustered in the future. To drop the clustering key, use the `ALTER TABLE` command with the `DROP CLUSTERING KEY` keywords, like this:

```
alter table CUST drop clustering key;
```

Data Storage Monitoring

Snowflake's consumption-based pricing model is made up of two basic metrics: usage of compute and cost of data storage. The cost of data storage depends on the number of bytes stored per month, as well as the cost of moving data across regions or clouds. Snowflake uses the compressed data size to calculate storage cost since all data that is stored in Snowflake is compressed.

The most obvious source of data storage cost comes from data that is stored in tables, also referred to as *active data*. Storage is also consumed in other Snowflake features, such as by data that is in time travel and failsafe. Both of these features are part of the continuous data protection that is available in all Snowflake accounts and is described in more detail in Chapter 9. For now, you just need to understand that there is additional storage cost associated with continuous data protection.

Note Remember that even when data is deleted from a table, it will continue to incur storage cost as long as it resides in time travel or failsafe.

To monitor data storage for the account, you must have the `ACCOUNTADMIN` role assigned to you. If you are using a 30-day free Snowflake trial account, you will have this role and can view data storage across your account.

One way to monitor data storage is via the Snowsight user interface. First ensure that `ACCOUNTADMIN` is selected as your current role. Then click the Admin option in the left menu, as shown in Figure 5-4. Choose Usage.

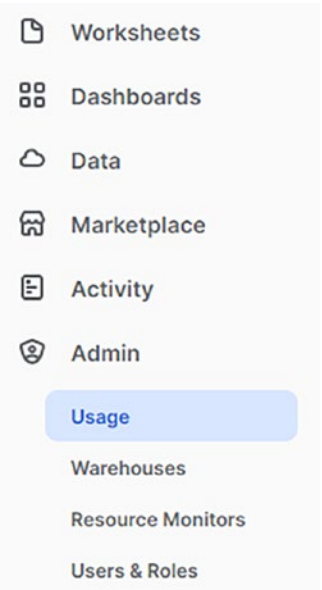

Figure 5-4. *Admin and Usage options in Snowsight*

In the window that appears, click the All Usage Types drop-down list and choose Storage. You will see a chart similar to the one in Figure 5-5.

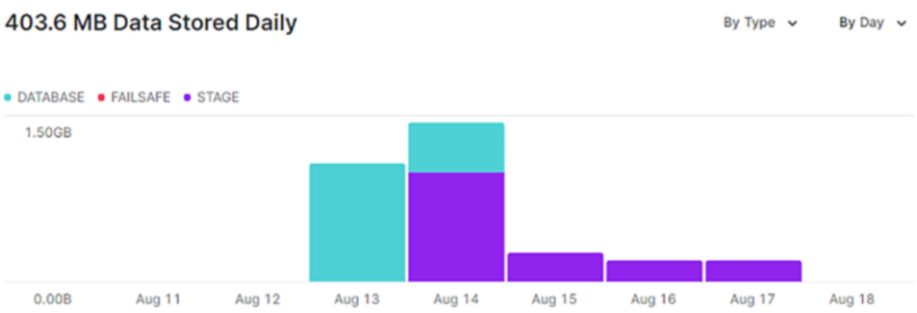

Figure 5-5. *Sample data storage chart in Snowsight*

You may configure the contents of the chart by setting options such as the time period that is included in the chart, or the types of data storage that you are interested in.

Another way to monitor storage is to query the `TABLE_STORAGE_METRICS` view in Information Schema or the `TABLE_STORAGE_METRICS` view in Account Usage.

For example, you can query the `TABLE_STORAGE_METRICS` view in Information Schema using this command:

```
select * from INFORMATION_SCHEMA.TABLE_STORAGE_METRICS;
```

You will get results similar to Figure 5-6.

TABLE_SCHEMA	TABLE_NAME	ID	ACTIVE_BYTES	TIME_TRAVEL_BYTES	FAILSAFE_BYTES
SALESDATA	CUSTOMER	2050	0	0	1,536
SALESDATA	CUSTOMER	4098	2,048	0	0
SALESDATA	ORDERS	7170	10,372	0	0

Figure 5-6. *Sample result from the TABLE_STORAGE_METRICS view*

The `TABLE_STORAGE_METRICS` view provides storage information for each table broken down by the physical storage in bytes for all three states of the Continuous Data Protection lifecycle: `ACTIVE_BYTES`, `TIME_TRAVEL_BYTES`, and `FAILSAFE_BYTES`.

As a Snowflake user without administrative privileges, you will not be able to monitor data storage across your account, but you must still understand what contributes to storage costs so that you use Snowflake cost-efficiently. This means that you should be familiar with how the following types of objects contribute to storage costs:

- Snowflake internal stages
- Cloned objects
- High churn tables
- Temporary and transient tables

Snowflake stages that are used for loading data from external storage locations into tables, incur storage costs, similarly as active data in tables. Working with stages is described in more detail in Chapter 7.

Snowflake's *zero-copy cloning* feature allows you to create a copy of an object without actually creating a physical copy of the object because both objects initially share the same underlying storage.

When a table is cloned, the cloned table shares all micro-partitions with the original table. This means that the cloned table will not incur additional storage costs. After the initial cloning, both the original and the cloned table follow different data lifecycles. Data can be inserted, deleted, or updated in the cloned table or the original table independently from each other. Whenever data is changed in the cloned table, new micro-partitions that are owned only by the clone are created. These new micro-partitions will incur additional cost because they are not shared with the original table.

For example, using the sample `CUST` table from Figure 5-1, you can clone the table into `CUST_CLONED` using this command:

```
create table CUST_CLONED clone CUST;
```

Both tables will share the same micro-partitions and the cloned table will incur no additional storage costs. When you update one of the rows in the cloned table, for example:

```
update CUST_CLONED
set CUST_NAME = 'xyzw'
where CUST_KEY = 149;
```

Snowflake will create a new micro-partition where it will store the updated data. The original CUST table will continue to store data in micro-partitions 1, 2, and 3, as shown in Figure 5-2, while the cloned table will continue to share micro-partitions 1 and 2 with the original table, but will incur additional cost for the newly created micro-partition 4, as shown in Figure 5-7.

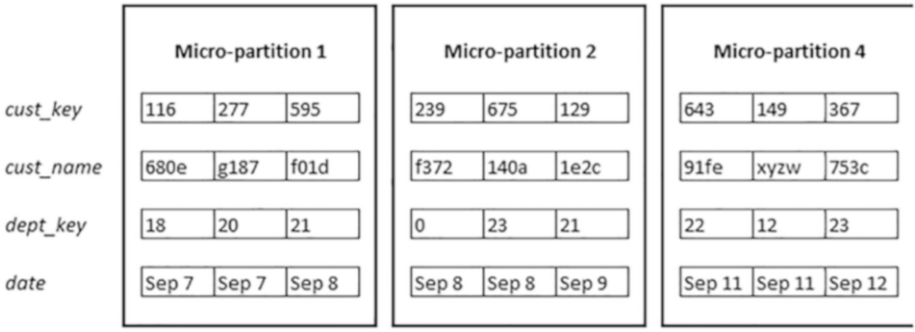

Figure 5-7. *Cloned table micro-partitions after updating*

Cloning is described in more detail in Chapter 9.

When a table is updated frequently, also referred to as a *high churn table,* many new micro-partitions will be created to keep track of all the changed data. Each micro-partition adds to the storage cost of the table, and these storage costs may add up, especially when the table is subject to continuous data protection in the form of time travel and failsafe.

Continuous data protection is described in more detail in Chapter 9.

Storage costs differ between permanent, transient, and temporary tables. Permanent tables incur storage costs associated with time travel and failsafe. In case this level of data protection is not needed, users may create transient or temporary tables, as described in Chapter 3. Storage costs associated with temporary and transient tables are as follows:

- **Temporary tables** have no failsafe and have a time travel retention period of 0 or 1 day. The maximum continuous data protection storage cost for a temporary table is 1 day.
- **Transient tables** have no failsafe and have a time travel retention period of 0 or 1 day. The maximum continuous data protection storage cost for a transient table is 1 day.

Search Optimization Service

The search optimization service helps improve query performance when queries filter data so that a small number of rows from a table is returned. Search optimization can be added to a table for columns with the following data types:

- Fixed-point numbers such as `INTEGER` and `NUMERIC`
- `DATE`, `TIME`, and `TIMESTAMP`
- `VARCHAR`
- `BINARY`

At this time, Snowflake doesn't support adding search optimization on columns with floating-point data types, semi-structured data types, or collations.

When you add search optimization to a table, a search access path is created by a Snowflake service that runs in the background. You don't specify a virtual warehouse for this service because Snowflake takes care of it behind the scenes.

Whenever data in the table is added, updated, or deleted, the search access path is automatically updated by Snowflake to ensure it is up to date. Although it may take some time to update the search access path in the background, you can still execute queries even when the search access path update is in progress. You will still get the correct results, only the queries might run slower until the background process completes.

Before you decide to add search optimization for a table, remember that this service incurs both storage and compute costs.

To add or remove search optimization for a table, the following privileges are required:

- `OWNERSHIP` privilege on the table
- `ADD SEARCH OPTIMIZATION` privilege on the schema that contains the table

You don't need any special privileges to use the search optimization service in a query. Because search optimization is a table property, you only need `SELECT` privileges on the table and the search optimization will be automatically detected and used as needed.

To add search optimization on a table, use the `ADD SEARCH OPTIMIZATION` keywords in the `ALTER TABLE` command, like this:

```
alter table CUST add search optimization;
```

After search optimization has been added, you can verify whether and how it has been applied to the table by using the `SHOW TABLES` command:

```
show tables like 'CUST';
```

A subset of the output of this command might look something like Figure 5-8.

search_optimization	search_optimization_progress	search_optimization_bytes
ON	100	131,072

Figure 5-8. *Selected columns from the output of the SHOW TABLES command*

The SEARCH_OPTIMIZATION column shows that the search optimization is ON for the table. The SEARCH_OPTIMIZATION_PROGRESS column shows the service percentage complete. For large tables, creating the initial search access path may take some time. This column will tell you how much of the service activity has been completed. The SEARCH_OPTIMIZATION_BYTES column tells you how much storage is consumed by the search access path.

To drop search optimization from a table, you can use the DROP SEARCH OPTIMIZATION keywords in the ALTER TABLE command, like this:

```
alter table CUST drop search optimization;
```

Choosing Tables for Search Optimization

To ensure that search optimization service is beneficial, Snowflake offers some criteria to help you identify which types of tables perform best, as well as which types of queries perform best.

Search optimization works best when the following criteria is met for the table:

- The table is not clustered.
- If the table is clustered, queries typically don't use the clustering key.

If the table is clustered, the clustering itself would improve query performance when using the clustering key; therefore, search optimization would not be needed in addition to clustering.

Choosing Queries for Search Optimization

Search optimization works best when the following criteria are met for the query:

- The query runs for a relatively long time, at least a minute or more.
- At least one of the columns that is used in the query filter has somewhere in the range of 100,000 and 200,000 distinct values.
- The query uses an equality predicate (for example, `where CUST_KEY = 149`) or the `IN` predicate (for example `where CUST_KEY in (149, 277)`)

For queries that use conjunctions (`AND`) or disjunctions (`OR`) of predicates, query performance can be improved by search optimization under the following conditions:

- **Conjunction (`AND`).** Any predicate in the query meets the query conditions listed previously.
- **Disjunction (`OR`).** All predicates in the query meet the query conditions listed previously.
- **Both conjunction (`AND`) and disjunction (`OR`).** The `AND` predicate is at the top level.

When a query joins tables, the search optimization service does not affect query performance directly. However, it can aid in filtering rows from tables that participate in the join, as long as the tables in the join condition have search optimization enabled.

Search optimization also works with views, for example when the query for a view uses tables that have search optimization enabled. Just like with joins, search optimization can aid in filtering rows from tables that are used in the query for the view.

Snowflake decides internally whether it will use search optimization for each query. Users don't have any control over which queries will use search optimization and to what extent.

To verify whether a query has used the search optimization service, you can examine the Query Profile tool and check whether the Search Optimization Access node is included in the query plan. Query Profile is described in more detail later in this chapter.

Using Tables with Search Optimization

Actions such as modifying, cloning, replicating, or sharing tables that use search optimization have various consequences, depending on what type of modification was made. These are summarized in Table 5-2.

Table 5-2. *Consequences of Modifying, Cloning, Replicating, or Sharing a Table with Search Optimization*

Table Modification Action	Search Access Path State	Search Optimization Consequence
Add a column to a table	Valid	The column is added to the search access path automatically
Drop a column from a table	Valid	The column is removed from the search access path automatically
Rename a column	Valid	There is no change in the search access path
The default value of a column is changed	Invalid	Search optimization must be dropped and added back
Drop the table	-	The search access path is also dropped

(*continued*)

Table 5-2. (*continued*)

Table Modification Action	Search Access Path State	Search Optimization Consequence
Undrop the table	Valid	Search optimization is reestablished on the table
Drop search optimization from the table	-	The search access path is dropped and cannot be undropped
Add search optimization to a table	Valid	The maintenance service builds the search access path
Clone the table	Valid	The search optimization property is also cloned
`CREATE TABLE ... LIKE`	-	The search optimization property is not copied to the new table
Replicate the table	Valid	The search access path is rebuilt in the secondary database
Share the table	Valid	Data consumer queries can take advantage of search optimization

Cost of Search Optimization

The search optimization service incurs costs for both storage and compute resources.

The search access path data structure that is created by the search optimization service consumes storage. The storage cost of the search access path depends mainly on the number of distinct values in the table.

Adding search optimization to a table and maintaining it also consumes compute resources. Resource consumption is higher when the table is updated frequently. Deleting data also incurs some cost because the search access path must be maintained.

Before adding search optimization to a table, you can estimate the cost using the SYSTEM$ESTIMATE_SEARCH_OPTIMIZATION_COSTS function.

For example, using the sample CUST table from Figure 5-1, you could estimate the cost of adding search optimization using this command:

```
select SYSTEM$ESTIMATE_SEARCH_OPTIMIZATION_COSTS('cust');
```

The result of this command is a JSON structure, similar to what's shown in Figure 5-9.


```
{
  "tableName" :  "CUST",
  "searchOptimizationEnabled" : false,
  "costPositions" : [ {
    "name" : "BuildCosts",
    "costs" : {
      "value" : 0.000742,
      "unit" : "Credits"
    },
    "computationMethod" : "Estimated",
    "comment" : "estimated via sampling"
  }, {
    "name" : "StorageCosts",
    "costs" : {
      "value" : 0.000001,
      "unit" : "TB",
      "perTimeUnit" : "MONTH"
    },
    "computationMethod" : "Estimated",
    "comment" : "estimated via sampling"
  }, {
    "name" : "MaintenanceCosts",
    "costs" : {
      "value" : 0.000000,
      "unit" : "Credits",
      "perTimeUnit" : "MONTH"
    },
    "computationMethod" : "Estimated",
    "comment" : "Estimated from historic change rate over last ~7 day(s)."
  } ]
}
```


Figure 5-9. *Estimated search optimization costs for the sample CUST table*

In the resulting JSON structure, Snowflake provides an estimated cost (in credits) for the initial build of the search access path, an estimated storage size (in TB per month), and an estimated maintenance cost (in credits per month) based on the previous seven-day historic change rate.

The search optimization service can be used with most types of tables in Snowflake. However, you cannot add search optimization to external tables or materialized views. Additionally, you can't add search optimization to concatenated columns, columns that are calculated using analytical expressions, or columns that use data casting.

Search optimization can be used only on active data. This means that it will not benefit queries that use time travel data.

The Query Profile Tool

Query Profile is a Snowflake tool that helps you understand execution details of queries. For example, when you want to know more about the execution plan or performance of a chosen query, you can use Query Profile. The tool can also help you identify performance issues and less efficient usage of SQL query expressions.

The following example illustrates how to use and interpret Query Profile. First create two tables using the Snowflake sample database:

```
create table CUSTOMER as
select * from SNOWFLAKE_SAMPLE_DATA.TPCH_SF100.customer;
create table ORDERS as
select * from SNOWFLAKE_SAMPLE_DATA.TPCH_SF100.orders;
```

You will execute a query that joins these two tables in the Snowsight user interface:

```
select c_custkey, o_orderstatus, o_orderpriority, o_totalprice
from ORDERS O
inner join CUSTOMER C on C.c_custkey = O.o_custkey;
```

After executing this query, you will see the results in the usual results pane at the bottom of the window. To the right of the results pane, there is a Query Details pane, as shown in Figure 5-10. To access the Query Profile for the query that you just executed, click the three dots at the top right of this pane and choose View Query Profile.

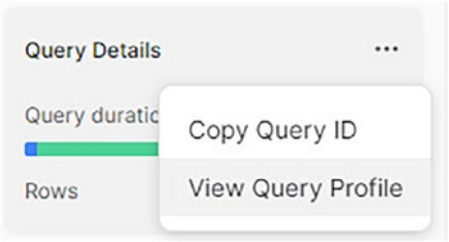

Figure 5-10. *Query Details pane in Snowsight*

A new window will appear with the Query Profile for the query that you just executed. This window provides a graphical representation of the execution plan for the query. Each component of the execution plan also shows several statistics, which are described in more detail in the following section. The graphical representation of the query plan is shown in Figure 5-11.

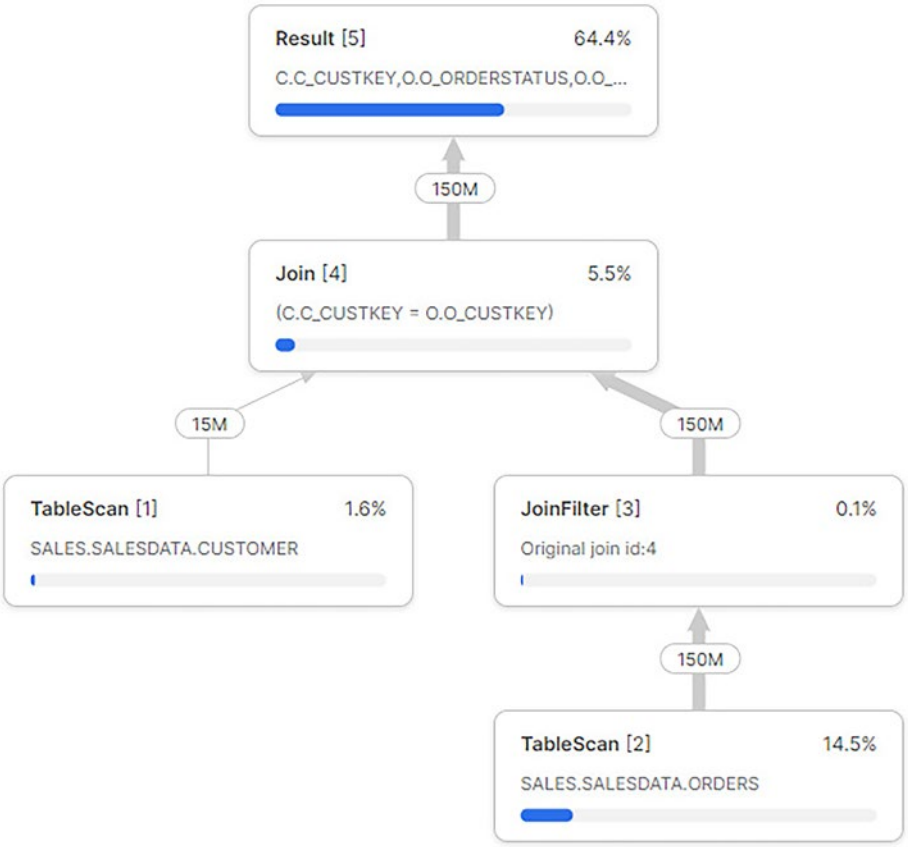

Figure 5-11. *Graphical representation of the query plan*

In addition to the graphical representation of the query plan, there are panes on the right side of the window: Most Expensive Nodes, Profile Overview, and Statistics.

Let's review each of these components in more detail. We will describe all items that can appear in each information pane, although each Query Profile usually contains a subset of these items. Only the statistics that are relevant for the query are shown in the panes.

The first pane is the Most Expensive Nodes pane, as shown in Figure 5-12.

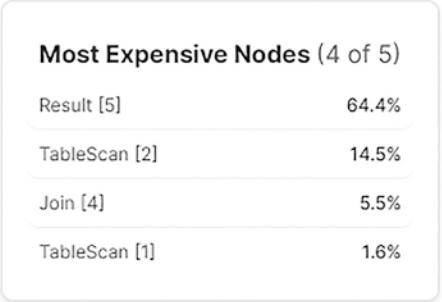

Figure 5-12. *Most Expensive Nodes pane in the Query Profile window*

In this pane you will see a list of all nodes that lasted for at least 1 percent of the total query execution time, sorted in descending order.

The second pane is the Profile Overview pane, as shown in Figure 5-13.

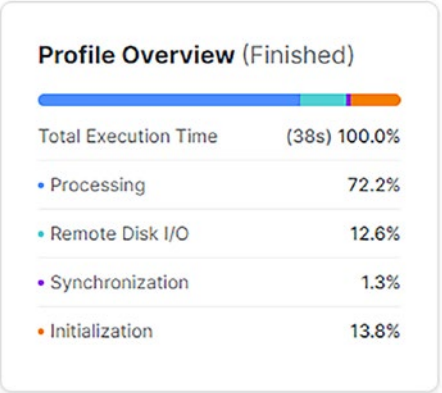

Figure 5-13. *Profile Overview pane in the Query Profile window*

In this pane, you will see which processing tasks consumed query time. Processing tasks are displayed by these categories:

- **Processing.** How much time was spent on data processing.
- **Local Disk IO.** How much time the processing was blocked by local disk.

- **Remote Disk IO.** How much time the processing was blocked by remote disk.
- **Network Communication.** How much time the processing was waiting for network data transfer.
- **Synchronization.** How much time was spent for synchronization activities between processes.
- **Initialization.** How much time was spent setting up query processing.

The third pane is the Statistics pane, as shown in Figure 5-14.

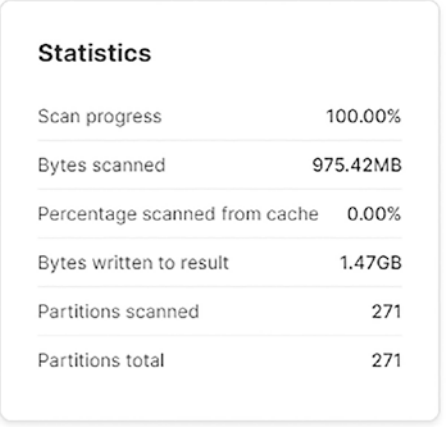

Figure 5-14. *The Statistics pane in the Query Profile window*

This pane shows information about various statistics, grouped by IO, DML, Pruning, Spilling, Network, and External Functions.

Note Not all statistics are applicable to all queries. This means that queries show different information panes of only relevant statistics.

In the IO group, the following information about the input-output operations performed during the query may be provided:

- **Scan progress.** Percentage of data scanned for a table so far (if you are viewing the Query Profile while the query is still running, this value can be less than 100 percent).
- **Bytes scanned.** Number of bytes scanned so far.
- **Percentage scanned from cache.** Percentage of data scanned from the local disk cache.
- **Bytes written.** Bytes written when loading data into a table.
- **Bytes written to result.** Bytes written to the result object, depending on what is produced as a result of the query, such as an intermediate data set.
- **Bytes read from result.** Bytes read from the result object.
- **External bytes scanned.** Bytes read from an external object, such as a stage.

In the DML group, the following statistics may be provided:

- **Number of rows inserted.** Number of rows inserted into tables.
- **Number of rows updated.** Number of rows updated in tables.
- **Number of rows deleted.** Number of rows deleted from tables.
- **Number of rows unloaded.** Number of rows unloaded during data export.

In the Pruning group, the following information may be provided:

- **Partitions scanned.** Number of partitions scanned.
- **Partitions total.** Total number of partitions.

In the Spilling group, the following information about disk usage for operations where intermediate results do not fit in memory may be provided:

- **Bytes spilled to local storage.** Volume of data spilled to local disk.
- **Bytes spilled to remote storage.** Volume of data spilled to remote disk.

In the Network group, the amount of data sent over the network may be provided. In the External Functions group, additional information about calls to external functions may be provided.

Analyzing all these statistics is a major endeavor, left mostly to Snowflake administrators and query tuning experts. As a Snowflake user, it is still beneficial to understand some aspects of Query Profile, especially to be able to spot problematic queries. The most common statistics that should be analyzed are data spilling, data caching, and micro-partition pruning.

Data Spilling

When you execute a complex query, the amount of memory required to hold intermediate results of the query might not be sufficient. In this case, Snowflake will start writing data to local disk. If the local disk space is still not sufficient, the data will be written to remote disks. Writing data that doesn't fit into memory to local or remote disk is also referred to as *spilling*.

Not surprisingly, spilling can cause poor query performance, especially if data has been spilled to remote disk. To reduce the amount of spilling, Snowflake recommends:

- Using a larger virtual warehouse, which will increase available memory and local disk space.
- Processing data in smaller batches that all fit into memory.

Data Caching

Snowflake provides data caching to improve query performance by storing partial query results for reuse by other queries. There are a few different cache layers, including:

- **Result cache.** Holds query results for queries that have been executed within the last 24 hours. Users may access query results from the same queries that were executed by other users, under the condition that the underlying data has not changed since and the role of the user accessing cached results has sufficient privileges.
- **Local disk cache.** Holds data used by SQL queries. Data that is needed for each query is first retrieved from remote disk and stored in the local disk. Other queries that need the same data will benefit by using the locally cached data.
- **Remote disk.** Holds the long-term data storage.

Query Profile will show you whether data was scanned from the result cache, local disk cache, or remote cache. This will help you interpret query execution plans and understand why the same query may sometimes run

faster and sometimes run slower, depending on what level of caching was available. Caching and various caching considerations with respect to virtual warehouses are described in more detail in Chapter 6.

Micro-partition Pruning

Snowflake maintains micro-partition metadata, which enables pruning of micro-partitions during query execution, as explained earlier in this chapter.

When examining performance via Query Profile, you can calculate pruning efficiency by comparing the Partitions Scanned and Partitions Total values. Pruning is efficient when the number of scanned partitions is a small fraction of the total partitions.

Not all queries benefit from 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 partition pruning. If you know that there is a filter in the query that should reduce the number of scanned partitions, but you don't see the reduction in the Query Profile, this indicates that additional actions to reorganize data in the table may be useful.

Query History

The Query History page shows a list of queries that have been executed in your Snowflake account in the last 14 days. You can click each of the queries to view additional details about the query execution. If you are not a Snowflake administrator, you cannot view queries across your account, but you will still see a history of your own queries as well as a history of all queries executed by any of the roles that have been granted to you.

To view Query History in the Snowsight interface, click the Activity option in the left menu, as shown in Figure 5-15, and choose Query History.

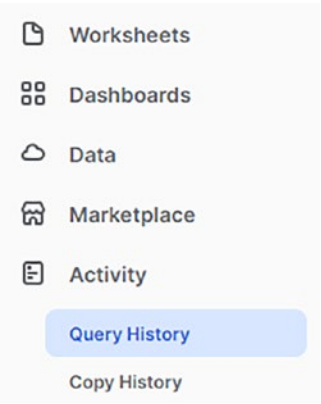

Figure 5-15. *Activity and Query History options in Snowsight*

The Query History page will appear with a list of queries that have been executed in the past; something like Figure 5-16.

SQL TEXT	QUERY ID	STATUS	WAREHOUSE	DURATION
select c_custkey, o_orderstatus, o_totalpr…	01a66f04-0000-2867-00…	Failed	COMPUTE_WH	26ms
select * from orders;	01a66f02-0000-286a-00…	Success	COMPUTE_WH	12s
create or replace table customer as select…	01a66f02-0000-286a-00…	Success	COMPUTE_WH	3.0s

Figure 5-16. *Sample query history*

On this page, you can add filters if you are looking for a specific query by clicking the Filters button. Once you have found the query that you want to analyze, click the line containing the query. A new window will appear where you can view the Query Details tab or the Query Profile tab, as explained earlier in this chapter.

Explain Execution Plan

Another way to view the execution plan for a query, other than using the graphical representation from the Query Profile, is using the `EXPLAIN` command provided by Snowflake.

For example, to view the execution plan for a query, add the `EXPLAIN` keyword in front of the query, like the following:

```
explain
select c_custkey, o_orderstatus, o_orderpriority, o_totalprice
from ORDERS O
inner join CUSTOMER C on C.c_custkey = O.o_custkey;
```

The output of this command will show the operations, such as table scans and joins, that Snowflake performs as part of query execution, similar to Figure 5-17.

step	id	parent	operation	objects	alias	expressions
null	null	null	GlobalStats	null	null	null
1	0	null	Result	null	null	C.C_CUSTKEY, O.O_ORDERSTATUS, O.O_ORDERPRIORITY, O.O_TOT/
1	1	0	InnerJoin	null	null	joinKey: (C.C_CUSTKEY = O.O_CUSTKEY)
1	2	1	TableScan	CUSTOMER	C	C_CUSTKEY
1	3	1	JoinFilter	null	null	joinKey: (C.C_CUSTKEY = O.O_CUSTKEY)
1	4	3	TableScan	ORDERS	O	O_CUSTKEY, O_ORDERSTATUS, O_TOTALPRICE, O_ORDERPRIORITY

Figure 5-17. *Sample output of the EXPLAIN command*

The output of the `EXPLAIN` plan command lists the steps that will be performed when executing the query and their relationships.

You can compare the output of the `EXPLAIN` command in tabular format, as shown in Figure 5-17, to the output of the Query Profile graphical representation, as shown in Figure 5-11. Since you are using the same query, the operations used in the query execution plan are the same in the tabular and graphical representations.

Note The `EXPLAIN` command does not execute the statement; it just compiles the statement to derive the execution plan.

Summary

This chapter discussed Snowflake features—including micro-partitions, clustering, and search optimization—that enable fast query execution on tables that contain large volumes of data. It reviewed the storage monitoring functionality that helps you understand how and when storage costs are incurred. You learned how to use the Query Profile to view query execution parameters such as data caching, data spilling, and micro-partition pruning. The chapter also described query history, which enables you to find details about queries that executed in the past.

The next chapter covers Snowflake virtual warehouses and other query performance optimization topics.

CHAPTER 6

Virtual Warehouses

A *virtual warehouse* in Snowflake is a cluster of compute resources that provides the memory, temporary storage, and CPU required to execute SQL statements and other DML operations such as loading data into tables.

This chapter describes virtual warehouses, starting with an overview of the warehouse sizes that are available in Snowflake. It then reviews multi-cluster warehouses and their properties. It compares scaling warehouses up and scaling warehouses out. Next, you will learn the most commonly used commands to work with warehouses. Finally, you will learn how to monitor warehouse loads and how warehouse usage impacts account billing.

To prevent users from running warehouses excessively, Snowflake administrators can set up resource monitors that monitor warehouse usage and send an alert or suspend user activity when limits are reached. You will review resource monitors as well in this chapter.

Warehouse Size

A Snowflake virtual warehouse is defined by its size and additional properties that control and automate warehouse activity. Warehouse size determines the amount of compute resources that are available to execute SQL statements and other DML operations.

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_6

Warehouses consume compute resources that are billed according to credits associated with their size. A list of supported Snowflake warehouse sizes along with their credit consumption per hour is provided in Table 6-1.

Table 6-1. *Snowflake Virtual Warehouse Sizes*

Warehouse Size	Credits per Hour
X-Small	1
Small	2
Medium	4
Large	8
X-Large	16
2X-Large	32
3X-Large	64
4X-Large	128
5X-Large	256
6X-Large	512

As seen in Table 6-1, each warehouse size consumes double the number of credits per hour as the preceding warehouse size.

A warehouse can execute many queries concurrently. The number of queries that can be accommodated depends on the complexity of each query. Before queries are executed, Snowflake estimates the resources that will be needed for execution. If the warehouse doesn't have sufficient resources available to execute the query, it will let the query wait in a queue until resources are released by prior queries that have finished executing.

To provide additional compute resources so that more queries can be accommodated, a Snowflake administrator can make a warehouse larger. Another way that the administrator can provide more resources is to create additional warehouses and submit subsequent queries to the newly created warehouses.

Adding warehouses can be automated by using multi-cluster warehouses. This is a Snowflake feature available in Enterprise Edition or higher. Multi-cluster warehouses are particularly useful when the number of users and queries is increasing so that resources are adjusted as needed.

Multi-cluster Warehouses

A virtual warehouse in Snowflake consists of a single cluster of compute resources that execute queries. When you need more compute resources than a single cluster can provide, Snowflake offers multi-cluster warehouses. A multi-cluster virtual warehouse is made up of a set of compute clusters with the following properties:

- Maximum number of clusters, at least 1 and up to 10
- Minimum number of clusters, a number up to the maximum number of clusters

Just like with single-cluster virtual warehouses, you can perform the following actions on multi-cluster warehouses:

- Choose a warehouse size
- Resize a warehouse
- Auto-suspend all clusters of a running warehouse due to inactivity
- Auto-resume a suspended warehouse when new queries are submitted

The number of clusters in a multi-cluster warehouse can be increased or decreased at any time, including while the warehouse is running and executing queries.

Additional properties of multi-cluster warehouses are the multi-cluster warehouse mode and scaling policy. These are represented graphically in Figure 6-1 and described in more detail in the following sections.

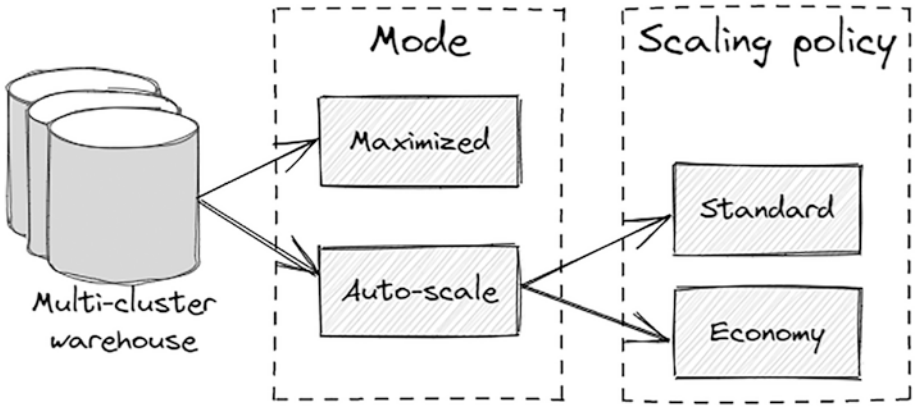

Figure 6-1. *Multi-cluster warehouse modes and scaling policies*

Multi-cluster Warehouse Modes

A multi-cluster warehouse can run in one of two modes:

- Maximized
- Auto-scale

A multi-cluster warehouse will run in *maximized* mode when the values for the minimum and maximum number of clusters are the same. When the warehouse is started, all the clusters are started to ensure that maximum resources are available to users. Administrators can increase or decrease the number of clusters as needed by the workload.

A multi-cluster warehouse will run in *auto-scale* mode when the values for the minimum and maximum number of clusters are different. Clusters are started and stopped dynamically, depending on warehouse load. Snowflake will start additional clusters, up to the maximum number of clusters, when the number of user sessions or number of queries increases. By the same token, Snowflake will shut down clusters when the workload decreases.

In auto-scale mode, users have some control over the approach that Snowflake uses when starting or shutting down clusters so that credits consumed by running warehouses are used economically.

Multi-cluster Warehouse Scaling Policies

Multi-cluster warehouses that are running in auto-scale mode adhere to one of two scaling policies:

- Standard (default)
- Economy

The *standard* scaling policy starts additional clusters whenever a query is queued or when a new query is submitted that can't be processed by the currently running warehouses. Using this policy, more compute credits may be consumed because clusters are started, but users experience better performance because queries are not queued and waiting for executing.

When a new cluster is started, it doesn't start sooner than 20 seconds after the previous cluster has started.

To determine whether a cluster can be shut down, Snowflake performs checks at one-minute intervals to find the least loaded cluster and determine whether the load from that cluster could be distributed to the remaining running clusters. After two to three consecutive such checks, the cluster is shut down.

The *economy* scaling policy starts an additional cluster only if there are enough queries waiting for execution that will keep the cluster running for at least six minutes. Using this policy, fewer compute credits are consumed, but some queries may take longer to execute because they have been queued.

To determine whether a cluster can be shut down, Snowflake performs checks at one-minute intervals to find the least loaded cluster and determine whether the load from that cluster could be distributed to the remaining running clusters. After five to six consecutive such checks, the cluster is shut down.

Examples of Multi-cluster Credit Usage

The following examples illustrate how to calculate credit usage for a multi-cluster warehouse.

Note There may be questions on the exam that ask you to calculate credit usage for various scenarios of multi-cluster warehouses, so be sure that you understand these calculations.

For these examples, to simplify calculations, the time intervals are rounded to 1 hour or 30 minutes.

First example. Calculate how many credits are consumed by a medium-size warehouse with three clusters that runs in maximized mode for two hours. For better understanding, this scenario is illustrated in Figure 6-2.

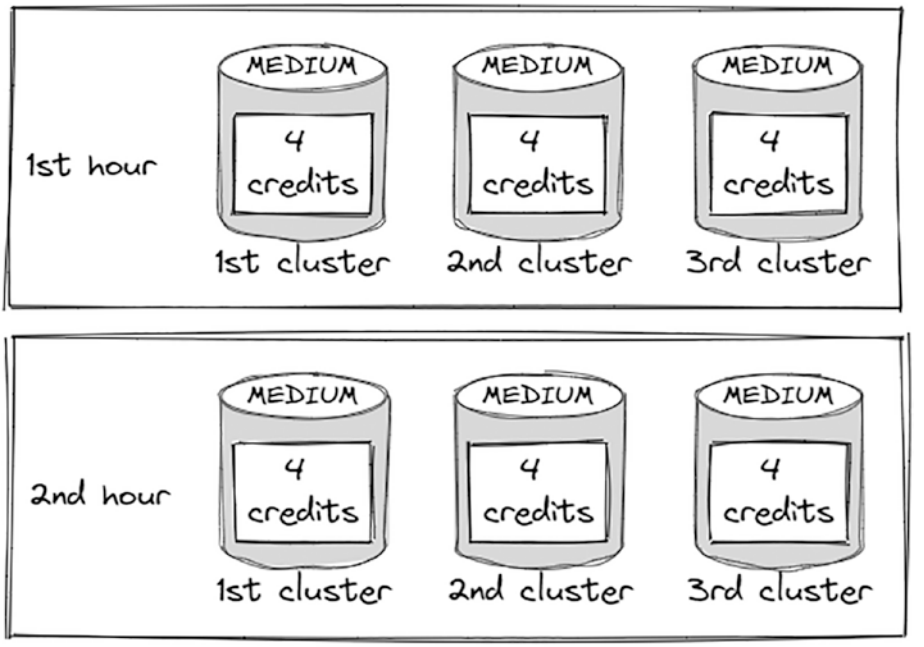

Figure 6-2. *Medium-size warehouse with three clusters in maximized mode*

This is what you need to know:

- According to Table 6-1, a medium-size warehouse consumes four credits per hour.
- This warehouse runs in maximized mode, which means that all clusters are running all the time.

The total number of consumed credits per hour is 3 clusters x 4 credits each = 12 credits per hour.

If the cluster runs for two hours, the total number of credits is 12 credits per hour x 2 hours = 24 total credits.

Second Example. This is another example using a warehouse running in auto-scale mode. You want to calculate how many credits are consumed

by a large-size warehouse with three clusters that run for two hours and auto-scale as follows:

- The first cluster runs continuously
- The second cluster runs continuously for the second hour only
- The third cluster runs for 30 minutes during the second hour only

The scenario is illustrated in Figure 6-3.

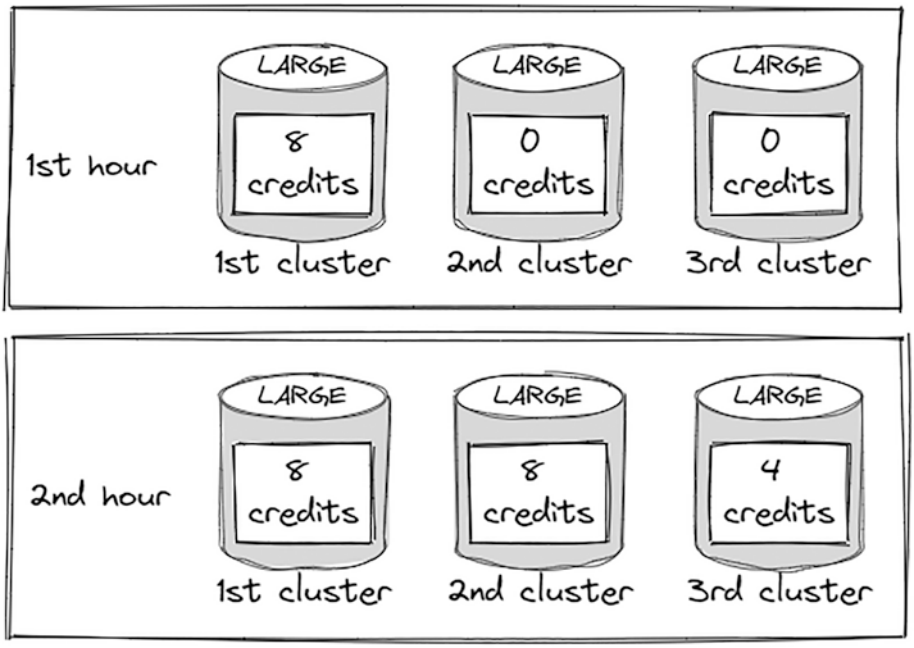

Figure 6-3. *Large size warehouse with three clusters in auto-scale mode*

Taking into consideration that a large-size warehouse consumes 8 credits per hour according to Table 6-1, you can calculate that the total number of consumed credits for this scenario is 16 for the first warehouse

(2 hours x 8 credits per hour), 8 for the second warehouse (8 credits during the second hour), and 4 for the third warehouse (half of 8 credits during the second hour), giving a grand total of 16 + 8 + 4 = 28 credits.

Warehouse Scaling

Snowflake supports two ways to scale warehouses:

- Scale up by resizing a warehouse.
- Scale out by adding clusters to a multi-cluster warehouse.

Scaling Up

Resizing a warehouse, also referred to as scaling up, results in better query performance. This is especially evident in complex queries that require a large amount of resources to execute. Small and simple queries that execute efficiently on a smaller warehouse will usually not benefit from a larger warehouse.

With a larger warehouse, more queries can be accommodated at the same time, which helps reduce the number of queries that have to wait in queue until sufficient resources are released by completing previous queries.

Snowflake supports resizing a warehouse at any time, even while it is running. Resizing a running warehouse does not impact queries that are already being processed by the warehouse. The additional compute resources are used for queued and new queries.

Decreasing the size of a running warehouse removes compute resources from the warehouse. When the compute resources are removed, the cache associated with those resources is dropped, which can impact performance in the same way that suspending the warehouse can impact performance after it is resumed.

Scaling Out

Adding clusters to a multi-cluster warehouse will increase concurrency when new users and workloads are added. Snowflake provides some recommendations with respect to configuring multi-cluster warehouses:

- With Snowflake Enterprise Edition or higher, configure all your warehouses as multi-cluster warehouses, even if the maximum number of clusters is set to 1. This will allow you to scale out as needed even if you are starting with just one cluster.
- Run your multi-cluster warehouses in auto-scale mode, so that Snowflake can add and remove clusters as needed, based on the workload.

To configure the number of clusters for a multi-cluster warehouse, use these guidelines:

- For the minimum number of clusters, the recommended value is 1 so that the least number of clusters will be started.
- For the maximum number of clusters, there is no recommended value. On the one hand, you should set this number as high as possible. On the other hand, depending on the size of the warehouse, credit usage could add up quickly so you should monitor usage and adjust as necessary.

Working with Warehouses

Before you create a warehouse, consider the following factors with respect to anticipated cost and performance requirements:

- Warehouse size
- Manual or automated resuming and suspending
- Number of clusters in case of a multi-cluster warehouse

The warehouse size that you choose depends on how it will be used and the workload. Some examples of typical workloads along with the recommended warehouse size are shown in Table 6-2.

Table 6-2. *Typical Workloads and Recommended Warehouse Size*

Workload	Recommended Warehouse Size
Loading data from files that are sized between 100MB and 250MB, compressed	Small, Medium, Large
Running queries in small-scale testing environments	X-Small, Small, Medium
Running queries in large-scale production environments	Large, X-Large, 2X-Large

Due to per-second credit billing and auto-suspend functionality that will limit overspending, you have the flexibility to start with a warehouse size based on your best guess and then adjust the size over time to match your workloads.

Creating a Warehouse

A warehouse can be created through the web interface or by using SQL.

To create a warehouse in the Snowsight user interface, click the Admin option in the left menu, as shown in Figure 6-4, and then choose Warehouses.

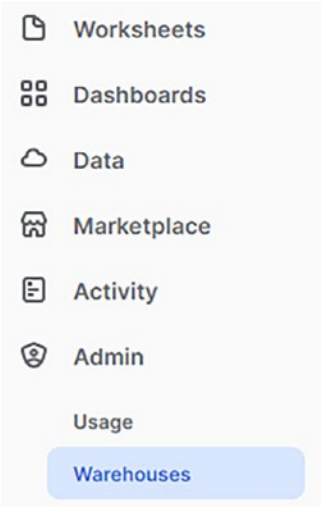

Figure 6-4. *Admin and Warehouses options in Snowsight*

You will see a list of all warehouses in your account, along with their status, size, number of clusters, and additional information.

To create a new warehouse, click the +Warehouses button at the top right. The New Warehouse window will appear, as shown in Figure 6-5.

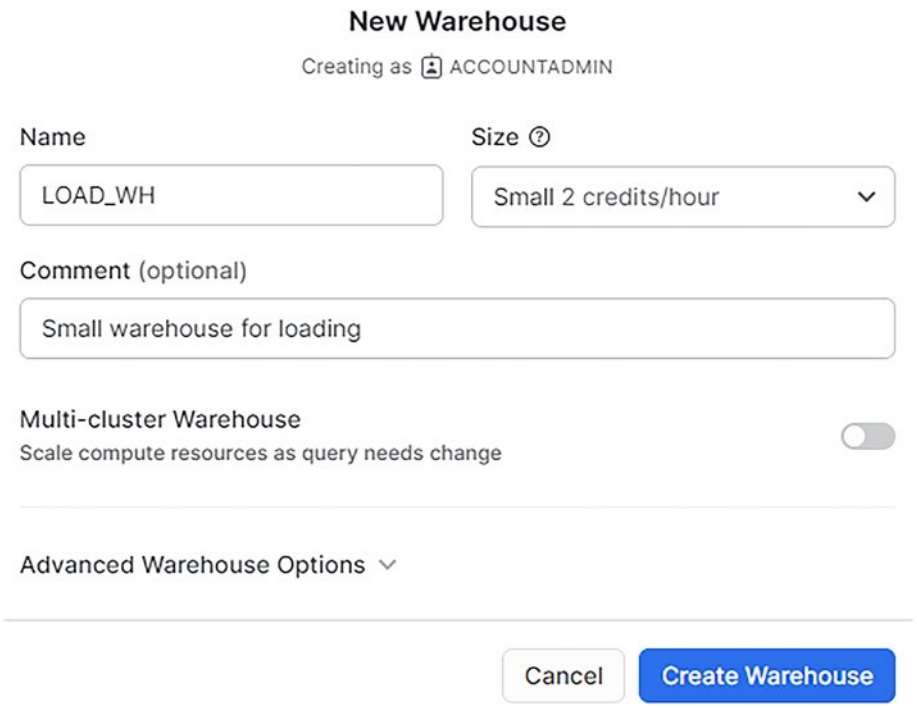

Figure 6-5. *New Warehouse window*

Fill in the name of the virtual warehouse; this is the only required parameter in this window. The default warehouse size is X-Large, but you should choose the size that suits your use case. You may add an optional comment. Then click the Create Warehouse button.

You will see the new warehouse added to the list of warehouses.

Alternatively, you can create the same warehouse using the following SQL command:

```
create warehouse LOAD_WH with warehouse_size = 'Small';
```

Note The default size for a warehouse created using the CREATE WAREHOUSE statement is X-Small. The default warehouse size for a warehouse created using the web interface is X-Large.

To create a multi-cluster warehouse, click the button next to the Multi-Cluster Warehouse option. Additional options will appear, as shown in Figure 6-6.

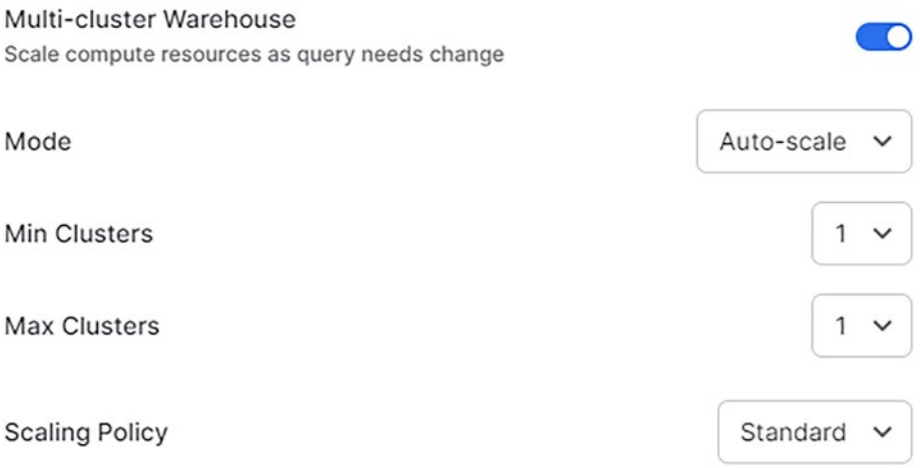

Figure 6-6. *Multi-cluster warehouse options in auto-scale mode*

First use the drop-down list box to choose the warehouse mode. If you choose auto-scale mode, you will see drop-down list boxes, as shown in Figure 6-6, which allow you to choose the minimum and maximum number of clusters and the scaling policy.

Alternatively, you can create a multi-cluster warehouse named PROD_WH that runs in auto-scale mode with a minimum of one and a maximum of two clusters using the following SQL command:

```
create warehouse PROD_WH
with warehouse_size = 'X-Large'
```

```
min_cluster_count = 1
max_cluster_count = 2;
```

If you choose maximized mode, you will see drop-down list boxes, as shown in Figure 6-7. They allow you to choose the number of clusters.

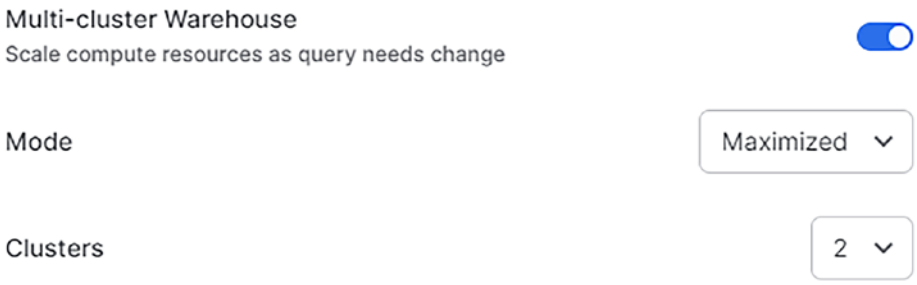

Figure 6-7. *Multi-cluster warehouse options in maximized mode*

Alternatively, you can create a multi-cluster warehouse named `PROD_WH` that runs in maximized mode with two clusters using the following SQL command:

```
create warehouse PROD_WH
with warehouse_size = 'X-Large'
max_cluster_count = 2;
```

The final group of options that can be set for a warehouse are available if you click the down arrow next to Advanced Warehouse Options. You will see the options in Figure 6-8.

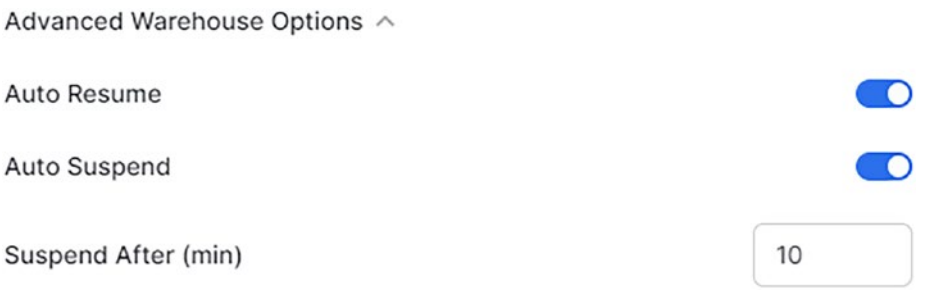

Figure 6-8. *Auto resume and auto suspend options*

By default, the *auto-suspend* option is enabled. That way, the warehouse will be automatically suspended after a defined period of time if it becomes inactive. Remember that a running warehouse consumes credits even when there are no queries running, which is why it is better to suspend the warehouse if it is not used. You also want the warehouse to resume as soon as a user wants to run a query. This behavior is ensured by setting the *auto-resume* option, which is enabled by default.

Snowflake offers the following recommendations with respect to setting auto-suspend:

- Auto-suspend should be set at a low value, such as somewhere between 5 and 10 minutes, to ensure that the warehouse does not consume credits when no queries are running.
- If you have many users accessing the warehouse at all times so that the workload is steady over time, you might disable auto-suspend to avoid suspending the warehouse and resuming again in a short period of time.
- When thinking about the auto-suspend and auto-resume parameters, keep in mind that the warehouse cache is dropped each time the warehouse is suspended and must be rebuilt each time the warehouse is resumed.

If you are creating a multi-cluster warehouse using a SQL command rather than in the Snowsight user interface, you can set the auto-suspend and auto-resume properties using `AUTO_SUSPEND` and `AUTO_RESUME` keywords.

In a multi-cluster warehouse, the auto-suspend and auto-resume parameters apply to the warehouse, not to the individual clusters in the warehouse. The following scenarios apply:

- When no queries are running on any of the clusters, and the minimum number of clusters is active, the warehouse will auto-suspend.
- When the warehouse is suspended and no clusters are running, it will auto-resume when a new query is submitted.

While setting the auto-resume option is preferred to ensure high availability whenever a user wants to run a query, you might still want to have better control over when a warehouse is started. In this case, disable auto-resume and resume the warehouse manually as needed.

Starting or Resuming a Warehouse

A warehouse can be started on initial creation or any time after that. It will continue to run while queries are submitted for execution, but will probably become suspended at some point because there will be no queries to execute and the auto-suspend time has expired.

A suspended warehouse can be resumed through the Snowsight user interface or using a SQL command. In the Snowsight user interface, click the three dots next to the suspended warehouse that you want to resume, as shown in Figure 6-9. Then choose the Resume option.

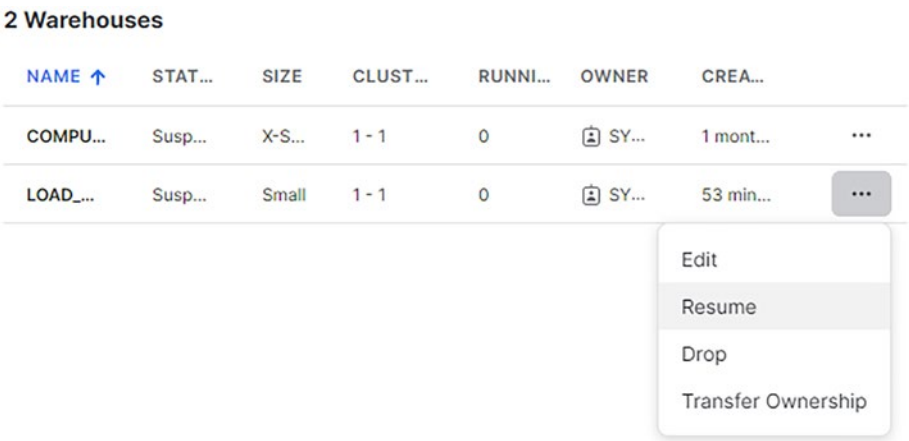

Figure 6-9. *Resume a suspended warehouse*

The same can be achieved using the following SQL command:

```
alter warehouse LOAD_WH resume;
```

When you resume a warehouse, it should become available in a few seconds. It is possible in rare circumstances that it might take longer. As soon as all compute resources are provisioned, the warehouse begins to consume credits.

During the time that the warehouse is resuming, Snowflake will not execute any submitted SQL statements. Rather, it will wait until all of the resources are ready, unless any of the resources fail to provision.

Note While the warehouse is resuming, if any compute resources fail to provision, Snowflake will try to repair the failed resources. To avoid waiting indefinitely, it will allow the warehouse to start executing SQL statements when at least 50 percent of the resources have been successfully provisioned.

Suspending a Warehouse

You can suspend a warehouse at any time, including when it is executing SQL statements. When a warehouse is suspended and all compute resources are shut down, it will not consume any more compute credits.

A warehouse can be suspended through the Snowsight user interface or by using a SQL command. In the Snowsight user interface, click the three dots next to the warehouse that you want to resume, as shown in Figure 6-10. Then choose the Suspend option.

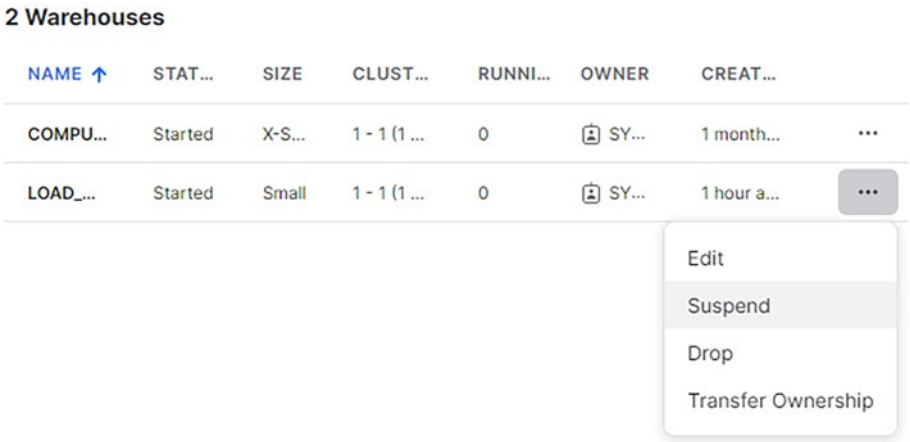

Figure 6-10. *Suspend a warehouse*

The same can be achieved using the following SQL command:

```
alter warehouse LOAD_WH suspend;
```

When you issue the suspend command to a warehouse, all idle compute resources will be shut down immediately. Any compute resources that are still executing statements will be allowed to complete execution. Only then do the remaining resources shut down and the warehouse becomes suspended. While compute resources wait to be shut down, the warehouse is considered to be in *quiesce* mode.

Resizing a Warehouse

You can resize a warehouse up or down at any time, including when it is executing SQL statements.

A warehouse can be resized through the Snowsight user interface or using a SQL command. In the Snowsight user interface, click the three dots next to the warehouse that you want to resize. Then choose the Edit option. The same window that is used to create a warehouse, as shown in Figure 6-5, will appear. Here you can change the warehouse size and any other options that are available.

You can also resize a warehouse using a SQL command. For example, to resize the warehouse called `LOAD_WH` to Medium, use this command:

```
alter warehouse LOAD_WH set warehouse_size = 'Medium';
```

When you resize a running warehouse, compute resources are added to or removed from each cluster in the warehouse. This doesn't impact any SQL statements that are being executed by the warehouse at that time. Existing queries will continue to run on existing capacity. Only statements that are waiting in the queue or newly submitted statements will execute on the newly provisioned resources.

When you resize a suspended warehouse, there is no need to provision new resources. When the warehouse is resumed any time after that, Snowflake will provision the additional compute resources that have been defined.

Using a Warehouse

Queries can be executed in Snowflake only if there is a virtual warehouse available, running, and set as the current warehouse in the session where the query will be executed. Only one virtual warehouse can be used in a session at the same time.

The current warehouse for a session can be selected or changed through the Snowsight user interface in the Roles and Warehouses drop-down list at the top right, as shown in Figure 6-11.

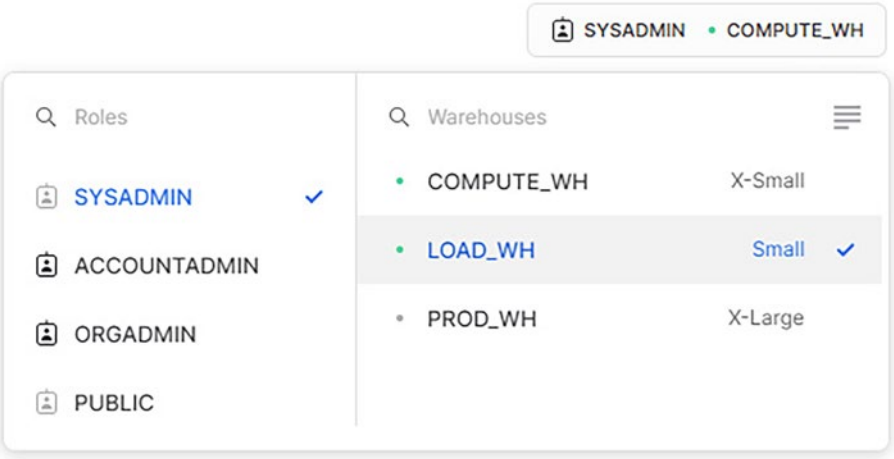

Figure 6-11. *Roles and Warehouses drop-down list*

Alternatively, you can execute a `USE WAREHOUSE` command. For example to use the warehouse `LOAD_WH` in the current session, execute this command:

```
use warehouse LOAD_WH;
```

Once a warehouse has been set as the current warehouse for a session, queries submitted in the session are processed by this warehouse.

To allow querying immediately after a session is initiated in Snowflake without having to explicitly choose the warehouse, you can set a default warehouse for each user. This warehouse will be used whenever the user initiates a new session in Snowflake.

In Snowflake clients, such as SnowSQL, JDBC driver, ODBC driver, Python connector, and others, you can also define a default warehouse.

When a client connects to Snowflake, the warehouse that will be used in the session is determined in the following order:

- If a warehouse was specified on the command line when running the client interactively or by setting parameters in the client tool, this warehouse will be used.
- If the previous is not specified, then the default warehouse that is set in the client configuration file will be used.
- If both of the previous are not specified, then the default warehouse of the user who is executing the client application is used.

Monitoring Warehouse Loads

The Snowsight user interface provides a Warehouse Activity chart that depicts queries processed by a warehouse over a chosen time period of up to two weeks. The measure in the chart is the average number of queries that were running or queued within the time period.

To view the chart, you must be using a role that has the `MONITOR` privilege on the warehouse. If you are using a 30-day free Snowflake trial account, you can use one of the system administration roles (`ACCOUNTADMIN` or `SYSADMIN`) that has this privilege.

In the Snowsight user interface, click the name of the warehouse in the Warehouses list. A page with the warehouse information will appear and the Warehouse Activity chart will be at the top, as shown in Figure 6-12.

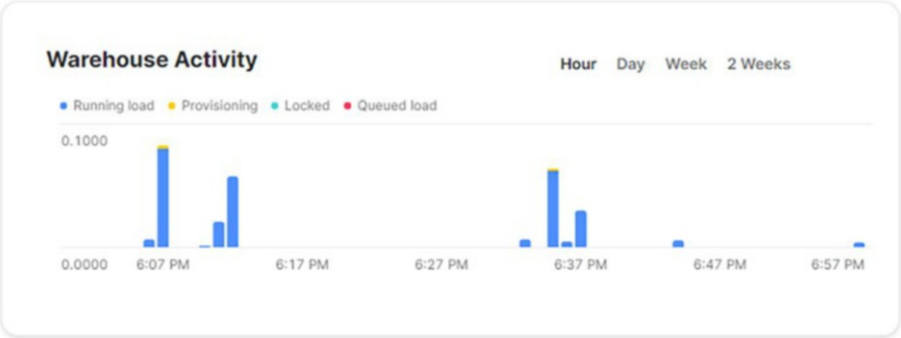

Figure 6-12. *Warehouse Activity chart*

Using the options at the top right, you can choose the time interval. This can be an hour, a day, a week, or two weeks. As you change the time interval, the chart adjusts dynamically to the chosen scale.

You can hover over a bar to view the average number of queries processed by the warehouse during the chosen time interval, as shown in Figure 6-13.

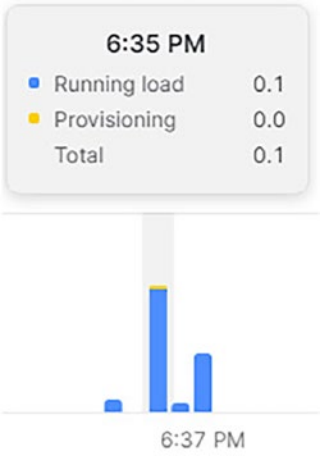

Figure 6-13. *Warehouse load details during the interval*

The bar displays the query load for each status, such as Running, Provisioning, Locked, or Queued, that occurred during the interval. Query load is calculated by dividing the execution time of all queries in an interval by the total time of the interval.

The load monitoring chart shows usage patterns across time and can help you spot patterns such as:

- **Slow query performance.** A query can run slowly due to an overloaded warehouse or it can simply be queued where it waits until resources are available for execution. In such situations, you can increase the available resources by creating additional warehouses or, in the case of multi-cluster warehouses, by adding more clusters. On the other hand, if you see that the warehouse is not overloaded but the query is still running slowly, you can resize the warehouse to provide more compute resources. Don't forget to re-execute the query after the additional resources are available.
- **Peak query performance.** You might find usage spikes when you review daily workload on the warehouse in the past. If you can isolate the queries that are causing the spike, you can create a dedicated warehouse for just those queries. You might even be able to reduce the size of the warehouse that will execute the remaining queries if the workload is not too high. With a multi-cluster warehouse, you can add clusters that will become available only during the peak load.
- **Excessive credit usage.** When reviewing the load monitoring chart, you might also see that query load was less than 1 for longer periods of time. This means

that the warehouse was consuming credits because it was running, but it was not used efficiently because it wasn't fully utilized. In this situation you might reduce the size of the warehouse, but keep in mind that you should continue to monitor the load to make sure that you haven't decreased it too much so that the performance would degrade. With a multi-cluster warehouse, you can just reduce the number of clusters.

Warehouse Billing

In addition to user-managed warehouses that process queries and are described in previous sections of this chapter, Snowflake also uses compute resources for the cloud services layer. Both of these types of compute resources contribute to warehouse billing.

Billing for User-Managed Warehouses

Billing for Snowflake virtual warehouses is calculated as the sum of:

- 60 seconds minimum each time the warehouse is started or resumed, even if it was running for less than 60 seconds
- Per-second billing after the first 60 seconds

When you view your credit usage, you will see that it is presented in hours, so you will see fractional amounts in hours due to per-second billing.

Table 6-3 summarizes a few examples of running warehouse scenarios with an explanation of how they are billed.

Table 6-3. *Examples of Warehouse Billing*

Running Warehouse Scenario	Billing Calculation
A warehouse runs for less than 60 seconds	Billed for 60 seconds (due to 60 second minimum)
A warehouse runs for 61 seconds	Billed for 61 seconds (due to per-second billing after initial 60 seconds)
A warehouse runs for 61 seconds, shuts down, then restarts and runs for less than 60 seconds	Billed for 121 seconds (61 seconds before it shuts down and 60 seconds minimum after it restarts)

When you resize a warehouse to a larger size, it will consume more credits because additional resources will be provisioned. Consumption of more credits results in higher billing.

Billing for multi-cluster warehouses is calculated based on warehouse size and the number of clusters that were running multiplied by the duration of time that the clusters were running. Some examples of multi-cluster warehouse credit usage calculations were provided earlier in this chapter.

Billing for Cloud Services

The Snowflake cloud services layer coordinates activities such as user logins and authentication, infrastructure management, metadata management, query optimization, access control, and a few others. It is described in more detail in Chapter 3.

Similarly to user-managed warehouses, services layer warehouses also consume Snowflake credits. However, the billing model is different. Cloud services will only be billed if their daily consumption exceeds 10 percent of the daily consumption of user-managed warehouses. Billing for cloud services is calculated each day and the amount that is shown on the monthly bill is calculated as the sum of these daily amounts.

Resource Monitors

A resource monitor can be used to monitor credit usage in Snowflake by both user-managed warehouses and warehouses used by cloud services. It can suspend user-managed warehouses when they exceed the credit usage threshold. It can also send email notifications when the limits are reached or are approaching.

Resource monitors can only be created by account administrators who have the `ACCOUNTADMIN` role. Users with other roles can be granted privileges to view and modify resource monitors.

A resource monitor is a first-class object in Snowflake. It can be created through the web interface or using SQL.

To create a resource monitor in the Snowsight user interface, click the Admin option in the left menu, as shown in Figure 6-14, and choose Resource Monitors.

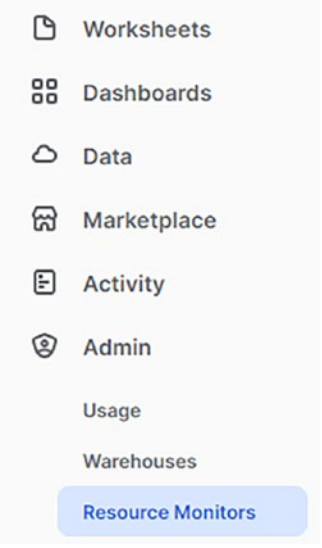

Figure 6-14. *Admin and Resource Monitors options in Snowsight*

You will see a list of your previously created resource monitors, if any. To create a new resource monitor, click the +Resource Monitor button at the top right.

A New Resource Monitor window will appear, as shown in Figure 6-15.

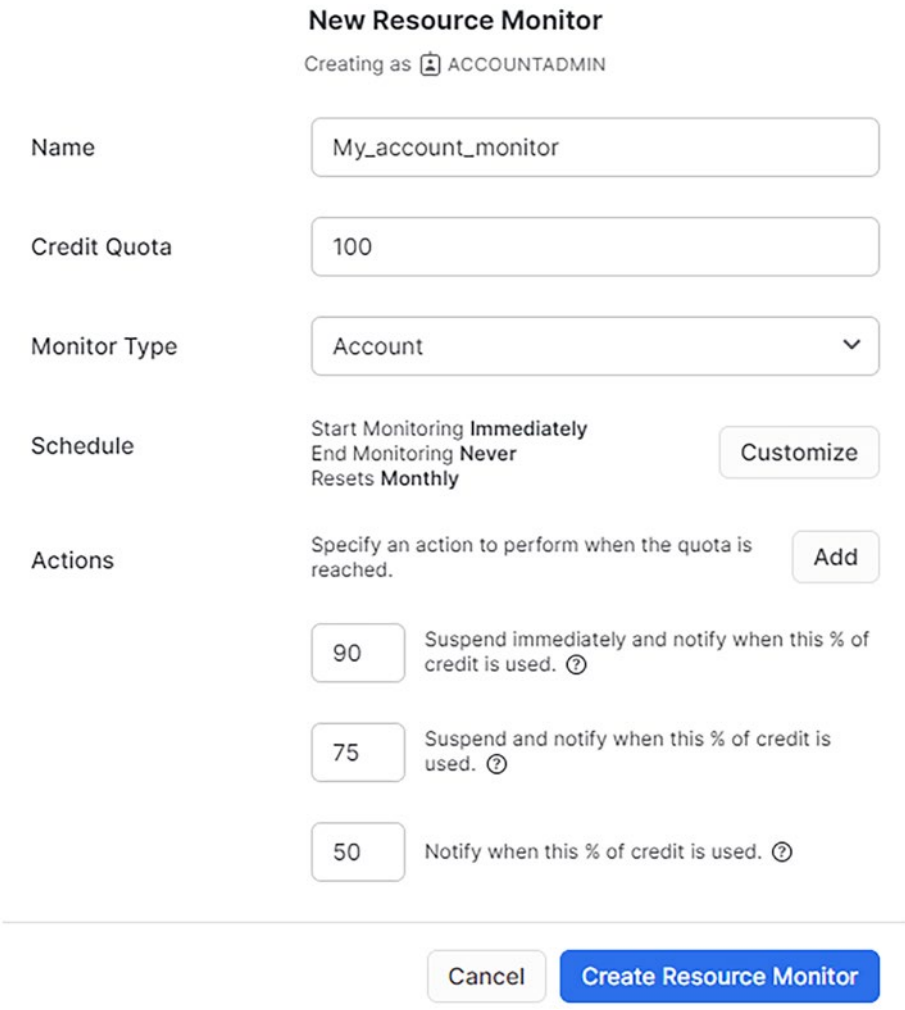

Figure 6-15. *New Resource Monitor window*

In this window, you can enter the resource monitor properties and parameters. First, enter the resource monitor name that you chose. Then fill in the resource monitor properties, described in more detail next.

Credit Quota. Specify the number of Snowflake credits allocated to the monitor for the chosen time interval. If the resource monitor is defined at the account level, this number includes credits consumed by all warehouses, including user-managed warehouses and warehouses used by cloud services.

Monitor Type. Specify whether the resource monitor will monitor credit usage for all warehouses in the account or for a group of individual warehouses. When you choose the monitor type as Warehouse, you will see a drop-down list of warehouses, where you can choose one or more warehouses to which the resource monitor will be assigned.

An account-level resource monitor will control only the user-managed virtual warehouses in the account, but not the services layer warehouses used for serverless features. Similarly, a resource monitor associated with an individual warehouse will not control warehouses used for cloud services.

Schedule. Specify the schedule for a resource monitor. The default settings are as follows:

- The resource monitor starts monitoring credit usage immediately after creation.
- There is no specified end date when to stop monitoring.
- Used credits reset to 0 at the beginning of each calendar month, which is the start of the standard Snowflake billing cycle.

You can customize the schedule by clicking the Customize button. A new window named Customize Resource Monitor Schedule, as shown in Figure 6-16, will appear.

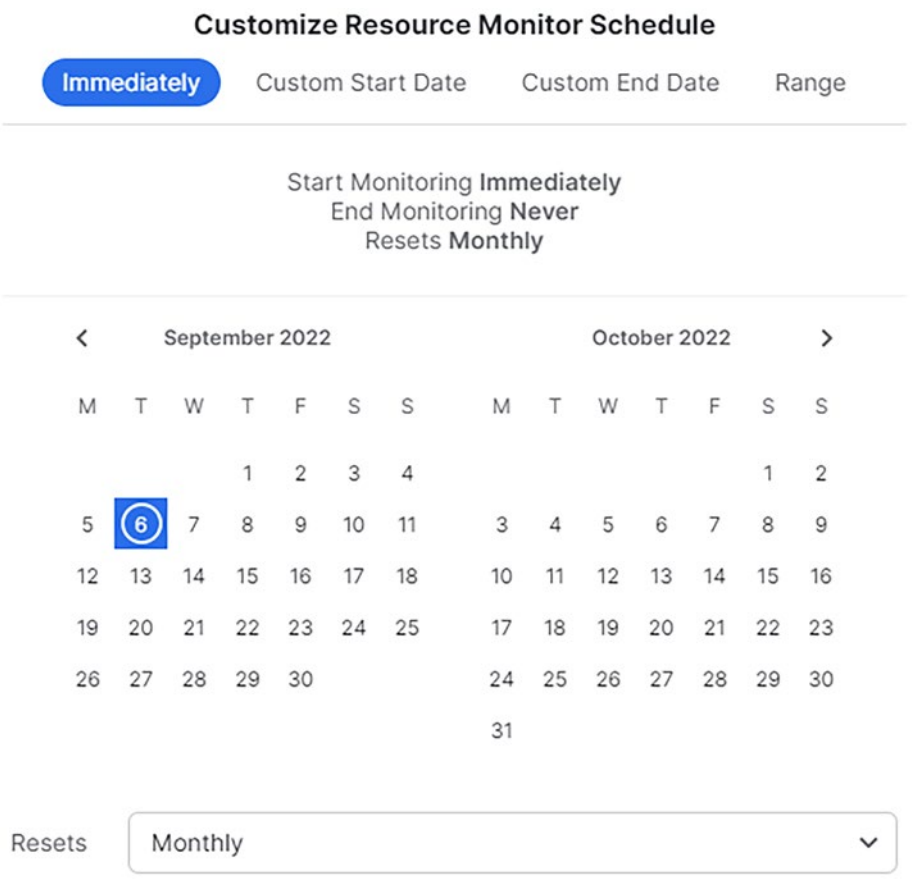

Figure 6-16. *Customize Resource Monitor Schedule window*

In this window, you can customize the schedule for a resource monitor with additional properties, such as:

- **Resets.** Specify the interval after which used credits reset. Possible values are Monthly, Daily, Weekly, Yearly, and Never. If you choose Never, used credits will never reset and the assigned warehouse will just use credits until it reaches the full credit quota.

- **Custom Start Date.** Specify the date and time when the resource monitor will start monitoring the assigned warehouses.
- **Custom End Date.** Specify the date and time when Snowflake suspends the warehouses that are controlled by the resource monitor. This is different from specifying the interval when used credits reset. With a custom end date, the warehouse is suspended regardless of any defined credit quota thresholds.

Actions. Each action specifies what to perform when the percentage of the credit quota is reached. Resource monitors support the following actions:

- **Suspend Immediately and Notify.** This action will suspend all assigned warehouses immediately and by this it will effectively cancel any statements that were executing at the time. It will also send a notification to all account administrators who have enabled their notifications.
- **Suspend and Notify.** This action will suspend all assigned warehouses but will wait until all statements that are still executing at the time have completed. It will also send a notification to all account administrators who have enabled their notifications.
- **Notify.** This action will not suspend any warehouses; it will just send an alert notification to all account administrators who have enabled their notifications.

Each resource monitor can have one Suspend action, one Suspend Immediate action, and up to five Notify actions. To add Notify actions, click the Add button in the New Resource Monitor window.

Suspend and Suspend Immediate actions apply only to user-managed warehouses. Virtual warehouses that provide cloud services cannot be suspended and may continue to run even after user-managed warehouses have been suspended.

Note A resource monitor is useful only when it has at least one defined action; otherwise, it will do nothing when the credit quota threshold is reached.

You can also create the same resource monitor with the following SQL command:

```
create resource monitor My_account_monitor
with credit_quota = 100
triggers on 50 percent do notify
         on 75 percent do suspend
         on 90 percent do suspend_immediate;
```

To allow flexibility in monitoring credit usage, you can define many Resource Monitors in your Snowflake account according to the following rules:

- You can define only one monitor at the account level.
- You can define multiple monitors that are each assigned to one or more warehouses.
- A warehouse can be assigned to no more than one resource monitor.
- When a warehouse is assigned to an account-level resource monitor and a warehouse-level resource monitor, the specified action from either of the resource monitors will be performed when the quota threshold is reached.

- When a single resource monitor is assigned to multiple warehouses, the defined credit quota applies to all the warehouses in summary.

If a resource monitor has a Suspend or Suspend Immediately action defined and its used credits reach the threshold, all warehouses assigned to the monitor are suspended. The warehouses will be resumed when at least one of the following conditions is met:

- At the start of the next time interval, as specified in the start date.
- When the credit quota for the monitor is increased.
- When the credit threshold for the suspend action is increased.
- When the warehouses are no longer assigned to the monitor.
- When the monitor is dropped.

Summary

This chapter described virtual warehouses. It provided an overview of warehouse sizes that are available in Snowflake. It reviewed multi-cluster warehouses and their properties. It compared scaling warehouses up and scaling warehouses out. It practiced the most commonly used commands to work with warehouses. It also discussed how to monitor warehouse loads, how warehouse usage impacts account billing, and how to set up resource monitors that monitor warehouse usage and send an alert or suspend user activity when limits are reached.

The next chapter explains how to load data into Snowflake and unload data from Snowflake.

CHAPTER 7

Data Loading and Unloading

When loading or unloading data into/from Snowflake, you must understand that Snowflake is hosted in the cloud. This means that loading and unloading data is easiest between Snowflake and other cloud storage locations. Loading or unloading data from local files requires an additional step that moves the data to a cloud storage location, where it is accessible to Snowflake.

There are many approaches to loading data into Snowflake, depending on the use case, including:

- Loading small amounts of data via a wizard in the user interface
- Loading data files from a cloud storage location using external stages
- Loading data files from a local file system using internal stages
- Loading continuously using Snowpipe
- Loading using external tables

This chapter first describes the file formats and properties of files that are used to load data into Snowflake. It reviews the different ways to bulk-load data into Snowflake, including the related commands used to

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_7

stage, load, and purge data files. It also discusses continuous loading using Snowpipe. It looks at creating external tables as an alternative approach to loading data. Finally, the chapter reviews unloading Snowflake data into data files.

Preparing Files for Data Loading

Files that contain data for loading into Snowflake can be prepared in a local environment or in cloud storage. If the files are prepared in a local environment, they must be copied to an internal Snowflake stage before they can be loaded into tables. If the files are in cloud storage, they can be loaded into tables directly from these cloud storage providers:

- **Amazon S3.** Files can be loaded from buckets created by users.
- **Google Cloud Storage.** Files can be loaded from buckets created by users.
- **Microsoft Azure.** Files can be loaded from containers created by users in blob storage, data lake storage gen2, general-purpose v1, or general-purpose v2.

File Formats and Encoding

File formats that can be loaded into Snowflake include:

- Delimited files such as CSV and TSV, or files using any valid delimiter
- Semi-structured formats such as JSON, Avro, ORC, Parquet, and XML
- Unstructured formats

The default file encoding for delimited files such as CSV and TSV is UTF-8. You may use other characters sets, but you must then explicitly specify the encoding to be used for data loading. Semi-structured file formats, such as JSON, Avro, ORC, Parquet, and XML, support only UTF-8 encoding.

File Compression and Encryption

Files for loading data may be compressed or uncompressed. If the files are uncompressed, they will be automatically compressed using gzip in the Snowflake stage.

For files that are already compressed, Snowflake can automatically detect the following compression methods:

- gzip
- bzip2
- deflate
- raw_deflate

In addition, Snowflake can also load files that are compressed using the Brotli or Zstandard compression methods, but these must be explicitly specified.

Files for loading data may be unencrypted or encrypted. If the files are unencrypted, they will be automatically encrypted in a Snowflake internal stage using 128-bit keys. For stronger encryption, 256-bit keys can be enabled.

Files that are already encrypted can be loaded into Snowflake from external cloud storage as long as the key that was used to encrypt the files is shared with Snowflake. Encryption is discussed in more detail in Chapter 9.

File Sizing Best Practices

To ensure optimal performance when loading data in batches, Snowflake recommends that data files be somewhere between 100MB and 250MB in size when compressed. Because loading is performed using parallel operations, the number of data files to be loaded should be less than the number of load processes.

Snowflake recommends that you don't attempt to load very large files, for example files that are 100GB or larger. Loading a very large file could result in consuming a large number of credits. Additionally, if the data-loading operation takes more than the maximum supported session duration of 24 hours, it could be aborted without any data from the file being loaded.

Note If you must load a very large file, try to split the file into multiple smaller files before loading.

Snowflake also recommends not loading many very small files because there is an overhead for each file. It is better to combine smaller files into larger files if possible.

Semi-structured data is stored in the `VARIANT` data type in Snowflake after loading. This data type imposes a 16MB size limit, which means that you will not be able to load files of semi-structured data that are larger than 16MB.

With JSON files, there is a way around this limitation if the file contains an array of documents with the same structure. In this case, you can use the `STRIP_OUTER_ARRAY` keyword together with the `COPY` command. This will load each document in the JSON array as a separate row with the `VARIANT` data type. This means that each JSON document will be limited by the 16MB limit, not the entire file.

For example, consider a JSON file with two documents, as shown in Listing 7-1.

Listing 7-1. Sample JSON File with Multiple Documents

```
[
      {
            "fruit": "Apple",
            "color": "Red"
      },
      {
            "fruit": "Lemon",
            "color": "Yellow"
      }
]
```

Without using the `STRIP_OUTER_ARRAY` keyword, the contents of the entire file would be loaded into a single row with a `VARIANT` column in Snowflake, as shown in Figure 7-1. In this case, the file size is limited to 16MB.

ONE_FRUIT
[{ "color": "Red", "fruit": "Apple" }, { "color": "Yellow", "fruit": "Lemon" }]

Figure 7-1. *Table in Snowflake with contents of a JSON file in one row*

With the `STRIP_OUTER_ARRAY` keyword, each document in the JSON file will be loaded into a separate row with a `VARIANT` column in Snowflake, as shown in Figure 7-2. In this case, each document in the file is limited to 16MB.

ONE_FRUIT
{ "color": "Red", "fruit": "Apple" }
{ "color": "Yellow", "fruit": "Lemon" }

Figure 7-2. *Table in Snowflake with JSON documents in separate rows*

Folder Structures

Snowflake recommends using logical paths when preparing files for loading in cloud storage. You might organize your data by date, by geographical location, or by any other identifier that helps split the files into logical groupings.

By organizing data using logical paths, also referred to as *partitioning*, you can load subsets of data in each partition independently from each other. This allows you to use parallel operations during the load process.

In addition to loading a subset of data from a logical partition, you can also select a subset of files by file prefix, if the files are named so that this is possible, or by wildcard matching if desired.

File Formats

When loading files, Snowflake must know the format of the file to be able to interpret the data and load it into structured format in tables and columns. You may supply the format in the `COPY` command when loading data. However, if you load files with the same format frequently, you may also create a named format that is defined once and used whenever you load files with that format.

A named file format is a database object that encapsulates format information. For example, to create a file format named `FRUIT_FORMAT` that will load CSV files with a semicolon as the delimiter and with a header row that will be skipped when loading, you would use the following SQL command:

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

You can specify the file format in different places during the loading process. The load operation applies the options in the following order of precedence, overriding all previous options:

- COPY statement
- Stage definition
- Table definition

The CREATE FILE FORMAT command takes many options that allow you to specify file formatting. For descriptions of all the options, review the Snowflake file format reference documentation here:

https://docs.snowflake.com/en/sql-reference/sql/create-file-format.html

Sample File and Table for Loading

The following sections of this chapter describe each of the different ways that you can load data into Snowflake. The commands used for loading are illustrated using a sample CSV file named fruitbasket.csv. The contents of this CSV file are shown in Listing 7-2.

Listing 7-2. Sample CSV File

```
Fruit_name;Fruit_color
Apple;Red
Lemon;Yellow
Grape;Blue
```

In order to load data from this CSV file into a table in Snowflake, you need a table that will receive the data. Create a table named FRUITS using the following statement:

```
create table FRUITS (
    fruit_name varchar(100),
    fruit_color varchar(100)
);
```

Loading Using the Web Interface

A quick and easy way to load small amounts of data, up to 50MB, from files in your local file system into a Snowflake table is via a wizard in the user interface. Although you have been using the Snowsight user interface until now, this user interface does not (yet) provide a wizard for loading data. You have to visit the classic web interface to use the data loading wizard.

To open the classic web interface, click the Classic Console option in the left pane in the Snowsight user interface, as shown in Figure 7-3.

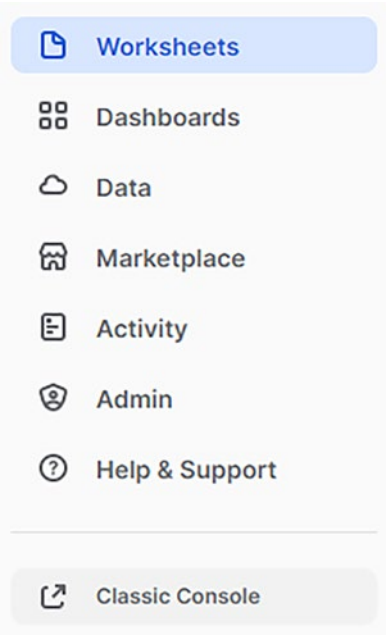

Figure 7-3. *Classic Console option in Snowsight*

A new window will open, where you will have to log in to your Snowflake account again using the same credentials that you use to log in to the Snowsight user interface.

If the Classic Console option is not visible in your account, click your username at the top left, choose Profile, and change your Default Experience to Classic UI. Log out of Snowflake and log in to your Snowflake account again using the same credentials that you use to log in to the Snowsight user interface. When you finish using the classic web interface, log out of Snowflake. The next time you log in to your Snowflake account, a popup window will ask you to confirm that your default experience is changed back to Snowsight.

Once the classic web interface window opens, click the Databases icon on the top row, as shown in Figure 7-4.

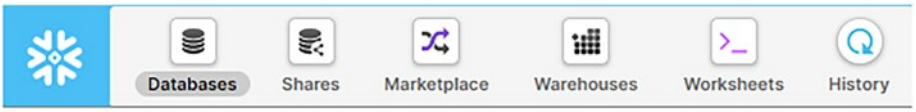

Figure 7-4. *Databases icon in the classic web interface*

A list of databases in your account will appear. Click the name of the database where you created your `FRUITS` table. A list of schemas will appear; click the name of the schema where you created your `FRUITS` table. Finally, a list of tables will appear; click the `FRUITS` table.

You will then see the Load Table option, as shown in Figure 7-5.

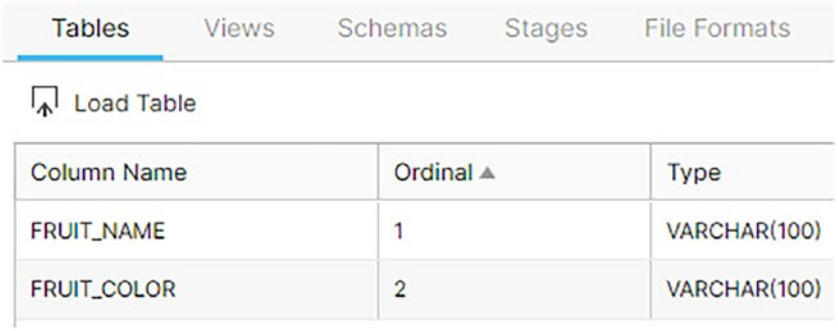

Figure 7-5. *Load Table option in the classic web interface*

Once you click the Load Table option, a wizard will appear that will guide you through the next steps.

The first step is to choose the warehouse that you want to use to load the data. If you are using a 30-day free Snowflake trial account, you should have the default `COMPUTE_WH` warehouse available in the drop-down list. Choose this warehouse and click Next.

The Source Files window will appear, as shown in Figure 7-6.

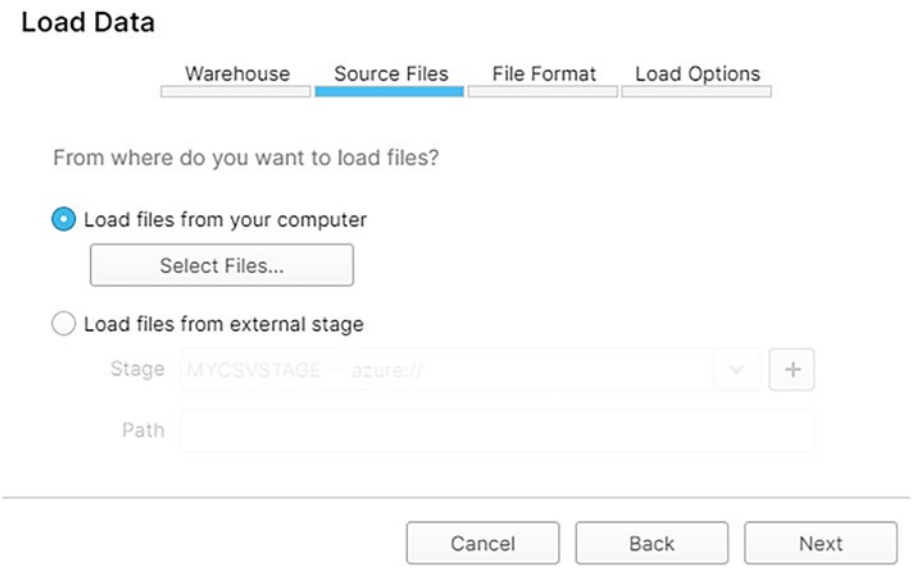

Figure 7-6. *Load Data wizard in the classic web interface*

From the Load Files From Your Computer radio button, click the Select Files… button. Navigate to the `fruitbasket.csv` file in your local file system and click Open to upload it. Then click Next in the wizard.

The File Format window will appear, as shown in Figure 7-7.

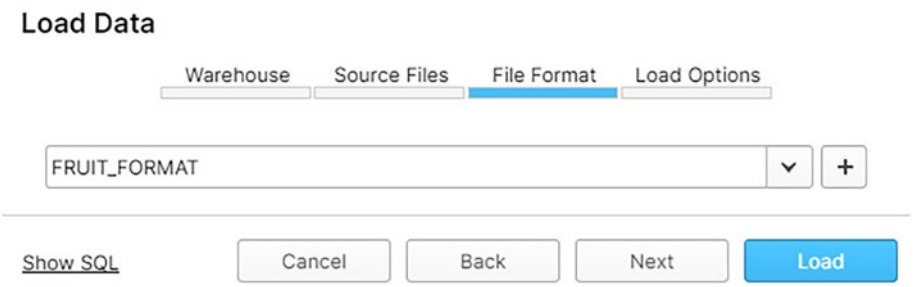

Figure 7-7. *Load Data wizard in the classic web interface*

Since you created the FRUIT_FORMAT file format earlier, it should appear in the drop-down list where you can choose it. Alternatively, you can click the + button, after which another window will appear, where you can define a new file format if desired.

After choosing the file format, click Next. The Load Options window will appear, as shown in Figure 7-8.

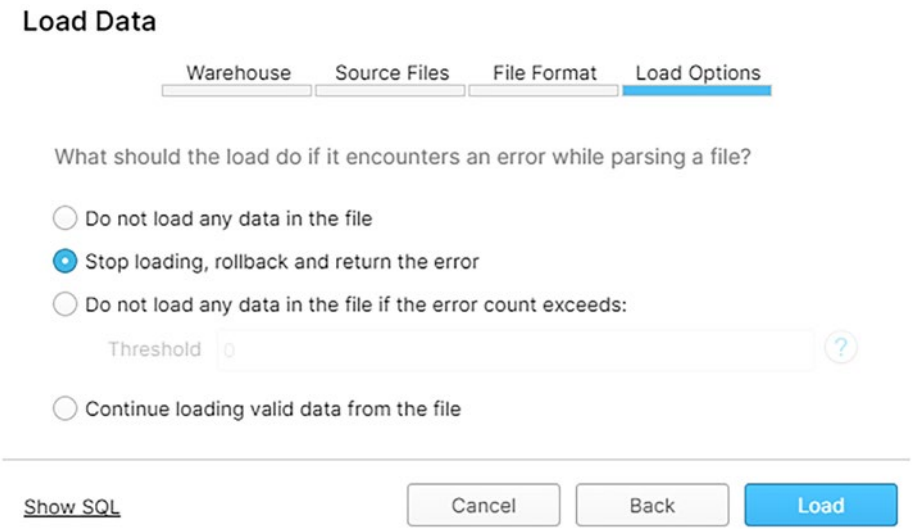

Figure 7-8. *Load Data wizard in the classic web interface*

Choose your preferred load options based on how you want any potential errors to be handled. Finally, click the Load button. The data from the CSV file will be loaded into the table. You will also see a message informing you that the data is being encrypted, which Snowflake will always do when loading data. When loading is complete, a window will appear that will show how many rows were loaded.

In any of the windows of the Load Data wizard, you can click the Show SQL hyperlink at bottom left. This shows you the SQL statements that the wizard generates behind the scenes that are executed during the load process. The SQL may look something like this:

```
PUT file://mycsvfiles/fruitbasket.csv @FRUITS/ui1663266395049

COPY INTO FRUITS FROM @/ui1663266395049
    FILE_FORMAT = 'FRUIT_FORMAT'
    ON_ERROR = 'ABORT_STATEMENT'
    PURGE = TRUE;
```

As you can see, the generated SQL includes the PUT and COPY commands using options that you chose in the wizard. The PUT and COPY commands are described in more detail later in this chapter.

Loading from Cloud Storage

To load data from files that are hosted in cloud storage providers, you can use any of the supported providers, including Amazon S3, Google Cloud Storage, or Microsoft Azure, regardless of the cloud platform that hosts your Snowflake account.

Note To follow along with the examples in this section, you must have access to one of the supported cloud storage providers. You can sign up for a free trial account with any of the providers if you don't have an account already.

The following examples use Microsoft Azure, but the process is similar when using Amazon S3 or Google Cloud Storage.

Creating an External Stage

A named external stage is a database object created in a schema. This object stores the URL that points to files in cloud storage, the settings or credentials used to access the cloud storage account, and optional settings

such as options that describe the format of staged files. For this example, create a stage named MY_CSV_STAGE using the CREATE STAGE command, like this:

```
create stage MY_CSV_STAGE
URL = 'azure://csvcontainer.blob.core.windows.net/csvfiles/'
CREDENTIALS = (AZURE_SAS_TOKEN = '?sv=...%3D')
file_format = (format_name = 'fruit_format');
```

With this command, you have created an external stage named MY_CSV_STAGE with a URL that points to a storage container named CSVCONTAINER on Microsoft Azure blob storage and to a path named CSVFILES, where the files are hosted. You also provided a SAS token, which is used as the credential that will allow Snowflake access to the Microsoft Azure storage container.

For instructions on how to configure the access credentials to your chosen storage provider, refer to the cloud provider's documentation. For specific syntax of the CREATE STAGE command for each of the supported cloud storage providers, refer to this Snowflake documentation:

https://docs.snowflake.com/en/sql-reference/sql/create-stage.html

---

Caution Although you may load data from any cloud storage provider into any Snowflake account regardless of the cloud platform that hosts the Snowflake account, keep in mind that some data transfer charges may apply when loading data from files in a cloud storage service in a different region or cloud platform from your Snowflake account.

To load data into Snowflake, data files must be uploaded to the cloud storage provider location. This can be done using tools that are supported by the cloud storage service. For this example, you will

manually upload the sample CSV file `fruitbasket.csv` to the Microsoft Azure blob storage container named `CSVCONTAINER` into the path named `CSVFILES`.

After you upload the file, you can view the contents of the external stage `MY_CSV_STAGE` using the `LIST` command, like this:

```
list @my_csv_stage;
```

This command will produce a list of all files in the external stage, along with their sizes, MD5 hash values, and the dates when the files were last modified, as shown in Figure 7-9.

name	size	md5
azure://csvcontainer.blob.core.windows.net/csvfiles/fruitbasket.csv	64	d6d03ea232c1

Figure 7-9. *List of files in the external stage*

Directory Tables

A directory table is an implicit object associated with an internal or external stage. It can be added to a stage when the stage is created using the `CREATE STAGE` command or later using the `ALTER STAGE` command. For example, to add a directory table to the `MY_CSV_STAGE` stage that you created earlier, use this command:

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

Directory tables store metadata about each of the data files in a stage. If you have the event messaging service available in cloud storage, you can use it to refresh the metadata for a directory table automatically. If not, you have to refresh the metadata manually using the `REFRESH` keyword in the `ALTER STAGE` command, like this:

```
alter stage my_csv_stage refresh;
```

Executing this command will synchronize the metadata in the directory table with the files located in the external stage.

You can query a directory table to retrieve a list of the file URLs in the stage along with their relative path, size, date the file was last modified, md5 hash, and ETag using this command:

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

Using the COPY Command

Once data is available in a cloud storage location and identified to Snowflake via a stage, you can load it into a table using the `COPY` command.

There is no requirement for the data files to have the same number and ordering of columns as the target table. You can perform simple transformations during a load, such as:

- Reordering columns
- Omitting columns that are not needed
- Casting to different data types
- Truncating character strings that exceed the target column length

To load the sample file `fruitbasket.csv` that has been staged in the external stage named `MY_CSV_STAGE` into the target table named `FRUITS`, you use the following `COPY` command:

```
copy into fruits
from @my_csv_stage
file_format = (format_name = 'fruit_format');
```

This command will look for all files in the `MY_CSV_STAGE` stage and load them into the `FRUITS` target table using the `FRUIT_FORMAT` file format that you defined earlier.

The COPY command can be used in many ways, with many different options and parameters. You can review all the options in the Snowflake documentation here:

https://docs.snowflake.com/en/sql-reference/sql/copy-into-table.html

Let's look at a few different ways to use the COPY command with the sample example file.

If you hadn't already created a file format, you can provide the format in the COPY command, like this:

```
copy into fruits
from @my_csv_stage
file_format = (type = csv field_delimiter = ';' skip_
header = 1);
```

If you hadn't already created an external stage, you can provide the object storage location and credentials in the COPY command, like this:

```
copy into fruits
from 'azure://csvcontainer.blob.core.windows.net/csvfiles/'
CREDENTIALS = ( AZURE_SAS_TOKEN = '?sv=...%3D')
file_format = (format_name = 'fruit_format');
```

You can mix and match the options as well. For example, you can provide the file format either in the stage definition or in the COPY command.

Copy Options

You can also add options to the COPY command that control the load behavior. A few of the most useful options are:

- ON_ERROR. When errors occur during the loading process, you can instruct Snowflake to either continue loading, skip the file, or abort the load process.

- SIZE_LIMIT. You can specify the maximum size of data to be loaded for a given COPY statement.
- PURGE. You can set this option to true or false, depending on whether you want to remove the data files from the stage after loading.
- MATCH_BY_COLUMN_NAME. This option allows semi-structured data to be loaded into columns in the target table that matches the corresponding columns represented in the data.
- ENFORCE_LENGTH. You can set this option to true or false, depending on whether you want to truncate character strings that exceed the target column length.

Using some of the copy options, you can construct a COPY statement to purge the loaded files from the stage and to skip the file when there's an error, like so:

```
copy into fruits
from @my_csv_stage
file_format = (format_name = 'fruit_format')
on_error = skip_file
purge = true;
```

Copy options can be specified in different locations and are applied in the following order of precedence:

- In the COPY command when loading data
- In the named stage definition by setting the COPY_OPTIONS in the CREATE STAGE command
- In the table definition by setting the STAGE_COPY_OPTIONS in the CREATE TABLE command

If you set COPY options in multiple locations, all of the options will be used, as long as they don't override each other. If they do override, they will be used in the order of precedence described here.

Lists of Files

The COPY command includes a FILES parameter, which allows you to choose files that you want to load by their name.

You can provide a list of file names, for example:

```
copy into fruits
from @my_csv_stage
files = ('fruitbasket.csv')
file_format = (format_name = 'fruit_format');
```

In addition to lists of files, you can also specify paths for an even more precise selection process.

The COPY command also supports the PATTERN parameter, which allows you to select files by pattern matching using a regular expression. For example, to load all files with a CSV extension, use this command:

```
copy into fruits
from @my_csv_stage
pattern = '.*\\.csv'
file_format = (format_name = 'fruit_format');
```

Note Remember that the PATTERN parameter accepts a regular expression, not simple wildcard matching. Therefore, you must escape the dot with a backslash and escape the backslash with another backslash, as shown in the previous example.

Preventing Data Duplication

The COPY command ensures that each file is loaded only once, thus preventing data duplication. For this purpose, Snowflake maintains metadata for each table into which data is loaded, including:

- File name
- File size
- File ETag
- Number of rows in the file
- Last load timestamp
- Errors during loading, if any

The load metadata is retained for 64 days. This is important to remember, because if the file was initially staged or modified fewer than or equal to 64 days ago, the COPY command will be aware of its load status and will not unnecessarily reload the data.

If the file was initially staged or last modified more than 64 days ago, the load status can still be determined by Snowflake if either of these events happened fewer than 64 days ago:

- The file was loaded successfully
- The initial set of data for the table was loaded

However, the COPY command cannot definitively determine whether a file has been loaded if the file was initially staged or last modified more than 64 days ago *and* the initial set of data was loaded into the table more than 64 days ago. In this case, the default Snowflake behavior is not to load the file so that data will not be duplicated in case it has already been loaded.

To change this default behavior, you can set the LOAD_UNCERTAIN_FILES keyword in the COPY command to TRUE. Another way to ensure that data is always loaded, regardless of any existing metadata about previous loads, is to use the FORCE keyword, which will load all files.

If you try to load the fruitbasket.csv file into the FRUITS table a second time using the same COPY command as before:

```
copy into fruits
from @my_csv_stage
files = ('fruitbasket.csv')
file_format = (format_name = 'fruit_format');
```

The data will not be loaded again. Instead, the returned status will be LOAD_SKIPPED and the message will be "File was loaded before".

Validation Before Using the COPY Command

Before loading data using the COPY command, you can validate that the data in the files will be loaded without errors. You can execute the COPY command in validation mode using the VALIDATION_MODE parameter. This parameter takes the following values:

- RETURN_n_ROWS. Replace _n_ in the parameter with an integer value that specifies how many rows will be validated. The validation will fail at the first error.
- RETURN_ERRORS. The validation will return all errors across all files specified in the COPY statement.
- RETURN_ALL_ERRORS. The validation will return all errors across all files specified in the COPY statement, including errors in partially loaded files from a previous load.

For example, to validate whether the sample fruitbasket.csv file will load successfully prior to actually loading any data, you use the following command:

```
copy into fruits
from @my_csv_stage
files = ( 'fruitbasket.csv' )
file_format = (format_name = 'fruit_format')
validation_mode = return_errors;
```

Removing Staged Files

Staged files can be deleted from a Snowflake stage using the following methods:

- Using the PURGE keyword with the COPY command. This will remove files that were loaded successfully from the stage.
- Executing the REMOVE command after the load has completed.

Snowflake recommends removing files from stages after successfully loading them for two reasons:

- To ensure that the files aren't loaded again.
- To reduce the number of files in the stage that must be scanned by the COPY command. This helps identify files that haven't been loaded already, which improves data loading performance.

Load History

The LOAD_HISTORY view in INFORMATION_SCHEMA stores the history of data loaded into tables using the COPY command. For example, you can use the following command to view all the COPY commands that were executed to load data into the FRUITS table:

```
select *
from INFORMATION_SCHEMA.LOAD_HISTORY
where table_name = 'FRUITS';
```

Loading from the Local File System

Loading data from the local file system is performed in two steps:

1. Upload data files to a Snowflake internal stage using the PUT command.
2. Use the COPY command to load the contents of the staged files into a table.

Types of Internal Stages

You can create internal named stages just as you create external stages. There are also internal stages that are automatically allocated to users and tables that you can use to stage files before loading. The properties of each type of internal stage are defined as follows:

- **User stage.** A user stage is available to each Snowflake user. It is used to stage files by a single user. The data from these files can be loaded into multiple tables.
- **Table stage.** A table stage is available for each table created in Snowflake. It is used to stage files by multiple users; the data from these files can be loaded into a single table.
- **Named stage.** A named internal stage is a database object created in a schema. It can be used to stage files by multiple users; the data from these files can be loaded into multiple tables.

You can create a named internal stage using the CREATE STAGE command, for example:

```
create stage my_internal_stage
file_format = (format_name = 'fruit_format');
```

The file format can be specified in the CREATE STAGE statement, but it can also be omitted, in which case it has to be provided later in the COPY command that's used to load data from the stage, similar to when using external stages.

To help you choose which type of internal stage to use in which situations, Table 7-1 provides a comparison of when to use each stage type and how to reference it.

Table 7-1. *When to Use and How to Reference Internal Stage Types*

Stage Type	When to Use	How to Reference
User stage	Files are accessed by a single user, but data can be copied into multiple tables.	`@~`
Table stage	Files are accessed by multiple users, but data is copied into a single table.	`@%table_name`
Named stage	Files are accessed by multiple users and data can be copied into multiple tables.	`@stage_name`

Loading Files into an Internal Stage

You can upload files from your local file system to any of the internal stages using the PUT command from the SnowSQL command-line interface that was introduced in Chapter 2.

The following PUT commands upload a file named fruitbasket.csv in the C:\mycsvfiles\ folder in your local file system to your user stage, the table stage associated with the FRUITS table, and the MY_INTERNAL_STAGE named internal stage, respectively:

```
put file://C:\mycsvfiles\fruitbasket.csv @~;
put file://C:\mycsvfiles\fruitbasket.csv @%fruits;
put file://C:\mycsvfiles\fruitbasket.csv @my_internal_stage;
```

Note This example loads the sample CSV file to all of the internal stage types for illustration purposes only. In practice, you will choose and use only one of the internal stage types.

To view a list of files that have been uploaded to each of the internal stages, use the LIST command:

```
list @~;
list @%fruits;
list @my_internal stage;
```

Using the COPY Command

Similarly to the external stages, you use the COPY command to load data from the internal stages into tables in Snowflake. For example, to load the staged fruitbasket.csv file from the internal stage MY_INTERNAL_STAGE to the FRUITS table, execute this command:

```
copy into fruits
from @my_internal_stage
file_format = (format_name = 'fruit_format');
```

Additional options for external stages that can be set with the `COPY` command—such as copy options, validation mode, and so on—can be used with internal stages as well. Purging files from stages and the `LOAD_HISTORY` view apply to internal stages as well.

Snowflake maintains the following metadata for each file uploaded into any of the internal stages:

- File name
- File size
- File timestamp

Snowflake keeps track of the historical data of all executed `COPY` commands for the last 14 days. This metadata can be used by the loading process if needed, similarly to metadata that is kept for external stages.

Loading Continuously Using Snowpipe

Rather than loading data in batches on a predefined schedule, Snowflake also supports continuous loading of data from files as soon as the files arrive in a stage. This functionality is provided by Snowpipe.

Snowpipe doesn't use compute resources from user-managed virtual warehouses. Instead, it uses Snowflake-provided compute resources, which are billed on a per-second basis. Snowflake decides the amount of resources that are required and seamlessly resizes them depending on the workload.

The main differences between Snowpipe and bulk data loading are summarized in Table 7-2.

Table 7-2. *Comparison of Snowpipe and Bulk Data Loading*

Attribute	Bulk Data Loading	Snowpipe
Authentication	Uses any of the authentication methods that are supported by the client.	Uses key pair authentication.
Load history	Retained in metadata for 64 days.	Retained in metadata for 14 days.
Transactions	Each load is performed as a single transaction.	Loads are dynamically combined or split into a single or multiple transactions based on the data volume.
Compute resources	Uses a user-managed virtual warehouse.	Uses Snowflake-provided compute resources.
Cost	Billed as per the user-managed virtual warehouse that was used for loading.	Billed for the Snowflake-provided compute resources on a per-second basis.

Detecting Staged Files

To allow Snowpipe to load files continuously, it must be able to detect newly arrived staged files. For this purpose, it can use the following mechanisms:

- Using cloud messaging
- Calling Snowpipe REST endpoints

Cloud messaging is based on event notifications from the cloud storage provider to notify Snowpipe that new files are available. To ensure that none of the files are lost, especially when many files arrive in short

time intervals, Snowpipe copies the files into a queue. Then the files are loaded into the target table using the commands that were provided when creating the Snowpipe object.

Depending on which cloud platform hosts the Snowflake account, cloud storage messaging might not be supported for all cloud storage services. Table 7-3 provides a mapping between Snowflake accounts hosted on each cloud platform to the cloud storage service support for automated Snowpipe.

Table 7-3. *Supported Cloud Storage Services for Each Snowflake Platform*

	Snowflake Account Cloud Platform		
Cloud Storage Service	**Amazon Web Services**	**Google Cloud Platform**	**Microsoft Azure**
Amazon S3	Yes	No	No
Google Cloud Storage	Yes	Yes	No
Microsoft Azure Blob Storage	Yes	No	Yes
Microsoft Data Lake Storage Gen2	Yes	No	Yes
Microsoft Azure General-purpose v2	Yes	No	Yes

As is evident from the table, only Snowflake accounts hosted on Amazon Web Services currently support notifications from other cloud providers.

Snowpipe can be called by a client application using REST endpoints with the name of a Snowpipe object and a list of file names. Similar to when using notifications, new files that are detected by the pipe object are queued for loading.

Continuous data loading is also supported by the Snowflake Connector for Kafka. This connector enables users to connect to an Apache Kafka server, read data from a Kafka topic that produces a stream of rows, and insert the data into a Snowflake table.

File Sizing for Snowpipe

Because Snowpipe is designed to load new data within minutes after a notification is received, Snowflake recommends that users follow the file sizing best practices and recommendations for batch loading described earlier in this chapter. As a general rule of thumb, load efficiency is best when data files are provided approximately once per minute.

When files are very large, loading data from these files may take a long time, especially in cases when data must also be decompressed, decrypted, or transformed. This may result in load latency. It also uses more compute resources, resulting in higher billing charges.

Each Snowpipe object manages its own queue of files waiting to be loaded. As new files arrive, Snowpipe appends them to the queue. There can be more than one process that pulls files from the queue and as a result, files may not be loaded in sequence by taking older files first. Snowpipe does not guarantee that files are loaded into tables in the same order that they arrived in the stage.

In addition to resource consumption, there is a charge for the overhead to manage files in the Snowpipe load queue. The charges are 0.06 credits per 1,000 files queued.

Preventing Data Duplication

Similar to the `COPY` commands, Snowpipe keeps track of file loading metadata to ensure that files are loaded only once and data is not duplicated. The load history for files that have been loaded using Snowpipe is preserved for 14 days.

Creating a Storage Integration

When creating an external stage, users must provide credentials to access the cloud storage provider. These credentials may be sensitive and should not be shared with a large group of users. To allow users to access files in an external cloud storage provider without knowing the access credentials, Snowflake provides *storage integration objects*.

A storage integration is also required by Snowpipe because data is loaded continuously and credentials to access the external storage location cannot be provided on a continuous basis.

A storage integration object is created by an administrator who encapsulates the authentication and access information in the object. Depending on the cloud provider, different parameters are used to provide the required information in the storage integration object:

- **AWS.** Identity and access management (IAM) user for the S3 cloud storage.
- **GCS.** Authentication for external cloud storage to a Snowflake service account.
- **Azure.** Service principal for the Azure cloud storage.

Note Only Snowflake account administrators or users with roles that include the `CREATE INTEGRATION` privilege can create storage integrations. You don't need to know the details of creating storage integrations for the SnowPro Core exam, as this is tested on the advanced exams.

Creating a Pipe

A Snowpipe, also called a pipe for short, is a Snowflake object that contains a COPY statement used to load data from the location where the source data files are staged to the target table. Snowpipe can load structured and semi-structured data.

A prerequisite for creating a pipe is to configure event notification in the cloud storage provider. This will notify Snowflake when new files have arrived. The following cloud provider messaging mechanisms can be used to configure the notification:

- **AWS.** Configure an S3 event notification or create an SNS topic and an SQS queue subscription to the topic.
- **GCS.** Configure the Pub/Sub service.
- **Azure.** Configure an Azure event grid with an event subscription.

For example, to create a pipe on AWS with an SNS topic, use this command:

```
create pipe my_aws_snowpipe
  auto_ingest = true
  aws_sns_topic = '<sns_topic_arn>'
  as
    copy into fruits
    from @my_csv_stage
  file_format = (format_name = 'fruit_format');
```

The AUTO_INGEST parameter must be set to TRUE to allow the pipe to receive event notifications sent from an S3 bucket to an SQS queue when new files are available.

In Azure and GCS, you must configure a notification integration object in Snowflake. In GCS, a notification integration contains the subscription name that was configured with the Pub/Sub service. In Azure, a notification integration contains the queue URL and tenant ID that was configured with the event grid.

For example, to create a pipe on Azure or GCS, use this command:

```
create pipe my_snowpipe
  auto_ingest = true
  integration = '<notification_integration_name>'
  as
    copy into fruits
    from @my_csv_stage
  file_format = (format_name = 'fruit_format');
```

Note that this example uses the `MY_CSV_STAGE` external stage that you created earlier in this chapter. The external stage creation for a Snowpipe is the same as creating an external stage for bulk loading from cloud storage.

Once a pipe is created, it will start loading new files as they arrive. To load files already in storage before the pipe was created, use the `REFRESH` option with the `ALTER PIPE` command:

```
alter pipe my_snowpipe refresh;
```

To check the status of a pipe, you can use the `SYSTEM$PIPE_STATUS` function. For example:

```
select system$pipe_status('my_snowpipe');
```

This function returns a JSON object containing name/value pairs, such as the pipe execution state, oldest file timestamp, pending file count, and more.

You can pause or resume a pipe by setting the `PIPE_EXECUTION_PAUSED` parameter to `TRUE` or `FALSE` in the `ALTER PIPE` command:

```
alter pipe my_snowpipe set PIPE_EXECUTION_PAUSED = true;
```

Removing Staged Files

Snowpipe doesn't support functionality to delete staged files after the data has been loaded into target tables. There is no `PURGE` option like in the `COPY` command that is used for bulk loading.

To remove staged files after Snowpipe has loaded the data, you have the following options:

- Manually execute the `REMOVE` command to delete files or schedule a process to execute this command periodically.
- Implement your own functionality using development or data lifecycle tools available in the cloud provider.

Loading Using External Tables

As an alternative to data loading, you can define external tables that enable querying data stored in external cloud storage using the usual Snowflake SQL statements. The data is physically stored in the external cloud storage location, but is visible to Snowflake users just like any other table in Snowflake. Because the external data is not managed by Snowflake, it cannot be deleted or modified using SQL queries.

For example, to create an external table named `FRUITS_EXT` that represents the `fruitbasket.csv` file located in the `MY_CSV_FILES` external stage, you can use the following command:

```
create external table fruits_ext (
  fruit_name varchar(100) as (value:c1::varchar),
  fruit_type varchar(100) as (value:c2::varchar),
  file_name varchar as metadata$filename)
with location = @my_csv_stage
file_format = (format_name = 'fruit_format')
auto_refresh = false;
```

The `value:c1::varchar` notation refers to the first column in the staged table, as indicated by `c1`. Similarly, the second column is indicated by `c2`, and so on. The `VALUE` keyword is a JSON representation of each row. The `::varchar` construct is type casting from the `VARIANT` data type of the JSON into the `VARCHAR` data type that will be used in the external table. `METADATA$FILENAME` is a pseudo-column in every stage that contains the file name and path.

The `AUTO_REFRESH` parameter in the `CREATE EXTERNAL TABLE` command, even when set to `TRUE`, works only when a notification is configured, similar to Snowpipe that was explained earlier in this chapter. If there is no notification service, you have to manually refresh the external table. Otherwise, it will not detect any new data that arrived in cloud storage. To do that, you can use the `REFRESH` keyword with the `ALTER TABLE` command, like this:

```
alter external table fruits_ext refresh;
```

You can query the external table using SQL statements, for example:

```
select *
from fruits_ext;
```

This query returns the data shown in Figure 7-10.

VALUE	FRUIT_NAME	FRUIT_TYPE	FILE_NAME ···
{ "c1": "Apple", "c2": "Red" }	Apple	Red	fruitbasket.csv
{ "c1": "Lemon", "c2": "Yellow" }	Lemon	Yellow	fruitbasket.csv
{ "c1": "Grape", "c2": "Blue" }	Grape	Blue	fruitbasket.csv

Figure 7-10. *Contents of the external table*

As you can see in Figure 7-10, the external table includes the `VALUE` column, which is included by default.

Querying external tables may be slow due to network overhead when accessing data that is stored in cloud locations. To improve query performance, you can create materialized views on external tables. Materialized views are described in more detail in Chapter 8.

Data Unloading

Similar to data loading, Snowflake supports bulk unloading data from a database table into various file formats, including delimited files such as CSV or TSV, as well as JSON and Parquet files. The process for unloading data into files is essentially the same as the loading process, except in reverse:

1. Use the `COPY` command to copy data from the Snowflake database table into one or more files in a Snowflake internal or external stage.
2. When unloading to an internal stage, download the file from the stage using the `GET` command, which works in the opposite manner as the `PUT` command.

   When unloading to an external stage, any tools that are available for the cloud storage provider to download the data files can be used.

There are a few differences between loading and unloading data. One of the differences is that when unloading data, you are not limited to unloading individual tables. Instead, you can specify a `SELECT` statement in the `COPY` command. The results of the query are written to the output file.

Unloading into Single or Multiple Files

When unloading data, you can create a single file or multiple files. This is especially useful when you are unloading a very large table and you don't want all of the data to be unloaded into a single file. You

can set the keyword SINGLE to TRUE or FALSE with the COPY command to indicate whether you want to unload data into a single file or multiple files.

When you choose to unload to multiple files, each file will have a unique name. If you specify the path in the COPY command, you can define a file name prefix that will be used in all file names. If you do not specify a prefix, Snowflake will prefix the file names with data_. A suffix will be appended to each file name to ensure that the file name is unique, even when the unload process was executed in more than one parallel thread. The suffix might take the form of 0_1_0 or similar.

When unloading data into multiple files, you should also provide a value to the MAX_FILE_SIZE keyword with the COPY command to define the maximum size of each file.

Using the COPY and GET Commands

This example shows how you unload data from the FRUITS table that you loaded earlier. First execute the COPY command to copy the contents of the table into a file in the internal stage named MY_INTERNAL_STAGE. Use this command to do so:

```
copy into @my_internal_stage/fruitoutput.csv
from fruits
file_format = (format_name = 'fruit_format')
single = true;
```

The output file is called fruitoutput.csv and the SINGLE option is set to TRUE so that only one file will be created. You must use a file format so that Snowflake knows in which format to unload the data. In the previous example, you used the FRUIT_FORMAT that you created earlier. But you could have used any other format to change this output format.

Instead of unloading data from a table, you can also unload data using a query. For example, to unload the results of the specified SQL SELECT statement into an output file named fruitsummary.csv, you can execute the following command:

```
copy into @my_internal_stage/fruitsummary.csv
from (select fruit_name, count(*) as cnt
      from fruits group by fruit_name)
file_format = (format_name = 'fruit_format')
single = true;
```

Once you have the output file in the internal stage, you can download it to the local file system by executing the GET command in SnowSQL, like this:

```
get @my_internal_stage/fruitoutput.csv file://C:\mycsvfiles\;
```

In the GET command, the internal stage name is specified as MY_INTERNAL_STAGE and the staged file name is called fruitoutput.csv. This example also specifies the place in the local file system where you want the file to be downloaded as C:\mycsvfiles\.

Summary

This chapter described file formats and properties for files that are used to load data into Snowflake. It reviewed different ways of bulk loading data into Snowflake, such as loading using the web interface, loading from cloud storage, and loading from the local file system. It described commands used to stage, load, and purge data. It also discussed continuous loading using Snowpipe and looked at creating external tables as an alternative approach to loading data. Finally, the chapter reviewed unloading data from Snowflake into data files.

The next chapter covers the various data transformation capabilities in Snowflake.

CHAPTER 8

Data Transformation

Snowflake supports standard SQL, including analytic extensions. This chapter does not review basic SQL commands; instead, it looks at a few Snowflake-specific SQL extensions. These extensions apply to standard data, semi-structured data, and unstructured data.

The chapter starts with an explanation of Snowflake data sampling and estimating functions that can be used on standard data. It demonstrates how to post-process query results in Snowflake. Then the chapter focuses on semi-structured data, including a review of data types used for storing semi-structured data, as well as loading and querying capabilities. It also discusses working with unstructured data.

Finally, the chapter describes materialized views that can be used to improve performance for workloads composed of common, repeated query patterns.

Data Sampling

The Snowflake `SAMPLE` function (or synonym `TABLESAMPLE`) returns a random sample of rows selected from a table. You can specify a value for the percentage of total rows returned from the table.

For example, to sample ten percent of rows from the `ORDERS` table in the Snowflake sample database, you would execute the following command:

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_8

```
select *
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.ORDERS
  sample (10);
```

In addition to sampling by rows, you can sample by block, which will return the specified percentage of blocks from the table. To sample by block, include the `BLOCK` keyword after the `SAMPLE` function.

You can also sample a fixed number of rows. In this case, the exact number of specified rows is returned. If the table contains fewer rows than the specified value for the sample, all rows in the table are returned. For example, to sample one million rows from the `ORDERS` table, execute the following command:

```
select *
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.ORDERS
  sample (1000000 rows);
```

Note The `SAMPLE` keyword applies to the table from which the query is selecting. When a query includes filtering, the `WHERE` clause is executed after the `SAMPLE` keyword.

Estimating Functions

Estimating functions are useful when you are working with very large data sets and you want an approximate result that can be calculated quickly. This is instead of waiting for the precise result, which may take longer to calculate and consume more credits. Individual estimating functions are described in the following sections.

Computing the Number of Distinct Values

To compute the precise number of rows that have distinct values in a chosen column, you can use the `COUNT` function with the `DISTINCT` keyword. Using the `ORDERS` table from the Snowflake sample database, execute the following command:

```
select COUNT(DISTINCT o_custkey)
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.ORDERS;
```

The query returns the value `999,982`. If you just need an approximate count of distinct values, you can use the `APPROX_COUNT_DISTINCT` function, like this:

```
select APPROX_COUNT_DISTINCT(o_custkey)
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.ORDERS;
```

The approximating query executes significantly faster than the precise query and returns `1,004,519`. This value is somewhat different from the value returned by the previous query, but as an approximation, it's close enough.

Estimating Similarity of Two or More Sets

Estimating the similarity of two tables can be useful for comparing two very large tables. For example, after you loaded data from the source to the target and want to verify that the data in the tables is the same. Instead of comparing the tables row by row, you can calculate a similarity coefficient between the two tables and use that to decide how closely the data matches.

The similarity coefficient is equal to 1 when the tables match exactly and is equal to 0 when the tables don't match at all. Any value between 0 and 1 indicates how closely the tables match.

To calculate approximate similarity, Snowflake uses the MinHash function on each of the data sets that you are comparing. Let's calculate approximate similarity between two sample data sets from the ORDERS table in the Snowflake sample database, where each of the sample data sets represents 50 percent of the randomly selected rows from the table.

To build this query, you start by selecting 50 percent of the randomly selected rows from the ORDERS table using this command:

```
select *
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.ORDERS
sample (50);
```

Then you calculate the MinHash of this sample using the following command:

```
select MINHASH(100, *) mh from (
  select *
  from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.ORDERS
    sample (50)
);
```

The MinHash function takes two parameters. The first parameter is the number of hash functions that will be used in the calculation. Snowflake recommends using 100, which the previous query uses. You can experiment with other values (up to 1024) for different results. The second parameter of the MinHash function is a list of columns to use for the calculation. This example specifies * because you want to calculate the similarity based on all columns.

Finally, you bring everything together by using the APPROXIMATE_SIMILARITY function on the MinHash of the two data sets, where each represents a 50 percent random sample of rows from the ORDERS table:

```
select APPROXIMATE_SIMILARITY(mh) from
(
```

```
  select MINHASH(100, *) mh from (
    select *
    from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.ORDERS
      sample (50)
  )
  union
  select MINHASH(100, *) mh from (
    select *
    from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.ORDERS
      sample (50)
  )
);
```

The result of this query is 0.4, thereabouts. Since you are using a random sample of rows, you will get a slightly different result each time you execute the query. The result indicates that the two data sets are approximately 0.4 similar on a scale of 0 to 1.

Estimating Frequent Values

In data analysis, one often calculates the frequency of values in a column in a table. As an example, using the PART table from the Snowflake sample database, you will examine the number of parts using the top ten containers in descending order.

To make the results more interesting, you will filter to select just the rows where the P_SIZE column has the value 33 and where the column P_BRAND equals 'Brand#44'. The query for this analysis is shown in Listing 8-1.

Listing 8-1. Query to Retrieve the Number of Parts Using the Top Ten Containers

```
select top 10 p_container, count(*) as cnt
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.PART
where p_size = 33 and p_brand = 'Brand#44'
group by p_container
order by 2 desc;
```

The results of executing the query from Listing 8-1 are shown in Table 8-1.

Table 8-1. *Number of Parts Using the Top Ten containers*

P_CONTAINER	CNT
MED CAN	68
WRAP JAR	53
SM BAG	52
MED PKG	51
SM JAR	47
SM PACK	47
LG PACK	47
WRAP BAG	46
LG PKG	46
WRAP BOX	46

When you are working with very large data sets, the query to calculate the frequency of values may take a long time to execute. Snowflake offers an alternative way to estimate approximate frequent values in large data sets that performs more efficiently.

For example, to calculate an approximate number of parts using the top ten containers, you can use the APPROX_TOP_K function:

```
with containers as (
 select APPROX_TOP_K(p_container, 10, 35) as containers_result
 from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.PART
 where p_size = 33 and p_brand = 'Brand#44'
)
select value[0]::varchar as container,
  value[1]::integer as cnt
from containers, lateral flatten(containers_result)
order by 2 desc;
```

The first parameter to the APPROX_TOP_K function is the column name where you want to calculate the frequency. This example uses P_CONTAINER. The second parameter indicates for how many different values you will be calculating the approximate frequency. This example uses ten because you are calculating the frequencies for the top ten containers.

The third parameter is used by Snowflake internally and defines the maximum number of distinct values that will be taken into consideration when performing the calculation. The approximation will be more accurate when you choose a value that is significantly larger than the total number of values. This example uses 35, but you can experiment with different values and see how the accuracy differs.

The results from APPROX_TOP_K are in JSON format. To format the results so that they resemble Table 8-2, you can add the LATERAL FLATTEN functionality to the query (the LATERAL FLATTEN syntax is covered later in this chapter).

Table 8-2. *Approximate Number of Parts Using the Top Ten Containers*

P_CONTAINER	CNT
MED CAN	68
WRAP JAR	56
LG PACK	51
LG BAG	50
SM BAG	50
SM JAR	49
MED PKG	48
WRAP BOX	48
WRAP BAG	47
SM PACK	46

As you can see in Table 8-2, the approximated counts are close, but not an exact match, to the results from the query used in Table 8-1.

Estimating Percentile Values

The definition of a *percentile value* is a number below which a certain percentage of values falls. More specifically, if a column in a table contains N values, the K percentile of this column is a number such that N*K of the numbers fall below this number.

Snowflake has the `PERCENTILE_CONT` function that you can use to calculate a percentile value. For example, to find the 50th percentile (or 0.5) value in the number of parts using the top ten containers shown in Table 8-1, you can execute the following SQL query (using the query from Listing 8-1 as a common table expression named `CONTAINER_FREQUENCY`):

```
with CONTAINER_FREQUENCY as (
select top 10 p_container, count(*) as cnt
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.PART
where p_size = 33 and p_brand = 'Brand#44'
group by p_container
order by 2 desc
)
select PERCENTILE_CONT(0.5) within group (order by cnt)
from CONTAINER_FREQUENCY;
```

The result is 47. You can verify visually that this result lies in the middle of the CNT column, as shown in Table 8-1.

When you are working with very large tables and want to calculate an approximate percentile value quickly, you can use the APPROX_PERCENTILE function:

```
with CONTAINER_FREQUENCY as (
select top 10 p_container, count(*) as cnt
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.part
where p_size = 33 and p_brand = 'Brand#44'
group by p_container
order by 2 desc
)
select APPROX_PERCENTILE(cnt, 0.5)
from CONTAINER_FREQUENCY;
```

The result is again 47. The estimated percentile value is the same as the precise percentile value that was calculated by the PERCENTILE_CONT function. Because you are working with a small data set for illustrative purposes, it's not surprising that the estimated result is the same as the precise result. However, approximating functions are meant to be used with very large data sets. In such cases, the estimations will likely differ from the precise results and will be calculated more quickly.

Post-Processing Query Results

Snowflake supports a useful feature that allows you to perform further processing on the result of a query that has been executed. Post-processing can be performed using the `RESULT_SCAN` table function. The function returns the results of the previous query as a table that can be used in a new query.

For example, execute the `SHOW` command with the `PARAMETERS` keyword to show all the parameters in the account:

```
show parameters in account;
```

The result of this statement is a long list of account parameters with their attributes. You can post-process these results using a SQL query, for example to select rows where the parameter value has been changed from the default value:

```
select "key", "value", "default"
from TABLE(RESULT_SCAN(LAST_QUERY_ID()))
where "value" <> "default";
```

This example uses the `RESULT_SCAN()` function with `LAST_QUERY_ID()` as the parameter to post-process the results of the last query that was executed. Alternatively, you could have provided a query ID of a previously executed query as the parameter value. The example also applies the `TABLE()` function to the `RESULT_SCAN()` function to allow the results to be queried like a table. The output of this query might look something like Figure 8-1 (your results might be different depending on your settings).

key	value	default
ODBC_QUERY_RESULT_FORMAT	ARROW	JSON
SNOWPARK_USE_SCOPED_TEMP_OBJECTS	false	true

Figure 8-1. *Result after post-processing the SHOW PARAMETERS command*

Note Column names in `SHOW` commands are usually stored in lowercase, which means that you must use double quotes to access them.

Semi-Structured Data

As opposed to structured data such as tables with columns, semi-structured data doesn't follow a predefined schema. Various types of semi-structured data can contain tags, names, keys, labels, or other keywords that identify individual entities in the data. Additionally, semi-structured data may be nested and the structures may evolve over time.

Supported Semi-Structured Data Formats

Snowflake supports storing and querying the following semi-structured data formats:

- JSON
- Avro
- ORC
- Parquet
- XML

As described in Chapter 3, data types that are used for storing semi-structured data include `ARRAY`, `OBJECT`, and `VARIANT`.

Note Don't forget that the maximum length of a `VARIANT` value is 16MB.

Let's briefly review each of the semi-structured data formats and look at how they are supported by Snowflake.

JSON (abbreviation for JavaScript Object Notation) is a collection of name/value pairs represented as objects. These contain lists of values represented as arrays. JSON syntax uses the following delimiters:

- Objects are enclosed with curly braces {}
- Names and values are separated by colons :
- Name/value pairs are separated by commas ,
- Arrays are enclosed with square brackets []

Using these delimiters, you can construct a sample JSON object enclosed in curly braces that contains two name/value pairs, separated by commas. Names must be enclosed in double quotes. If the values are provided as character strings, they must be enclosed in double quotes, like this:

```
{"fruit": "Apple", "color": "Red"}
```

The value in a name/value pair can contain most common data types, such as character strings, numbers, Boolean values, arrays, as well as nested objects. A value may also contain null.

Let's also construct a sample JSON array. You will enclose it in square brackets and the contents of the array will be represented by several objects, delimited by commas, for example:

```
[
  {"fruit": "Apple", "color": "Red"},
  {"fruit": "Lemon", "color": "Yellow"},
  {"fruit": "Grape", "color": "Blue"}
]
```

Avro is a row-oriented data serialization framework originally developed for the Apache Hadoop ecosystem. It serializes data in a binary format using JSON notation. Both the data and the schema are included in the output format, which makes it easier to deserialize.

Because Avro schemas use JSON notation, you can query Avro data in Snowflake using the same syntax as if you were querying JSON data.

ORC (Optimized Row Columnar) is a binary format that was developed in its own Apache project as an efficient way to store and access Hive data.

You can query ORC data that is stored in a `VARIANT` column in Snowflake using the same syntax as if you were querying JSON data.

Parquet is a column-oriented data file format that uses efficient data compression and decompression algorithms for highly performant data storage and retrieval within the Apache Hadoop ecosystem.

You can query Parquet data that is stored in a `VARIANT` column in Snowflake using the same syntax as if you were querying JSON data.

XML (eXtensible Markup Language) is a markup language used to encode documents in a format that is human-readable and machine-readable. XML was originally designed for storing documents, but it can be used to store and exchange many different types of data structures.

An XML document contains tags enclosed in angle brackets (< and >). These tags define the data structure and contain metadata according to the XML standard.

Elements in XML documents can begin with a start tag and end with a matching end tag or they can be made up of just a single tag. Anything between the start tag and end tag is the element's content.

You can query XML data that is stored in a VARIANT column in Snowflake using similar syntax as the other semi-structured data types.

Loading Semi-Structured Data

You can load semi-structured data from the supported semi-structured formats such as JSON, Avro, ORC, Parquet, and XML into Snowflake. Depending on your use case and the complexity of the semi-structured data, you can decide to load semi-structured data into a single VARIANT column or extract some of the structure and load it into more than one column with a corresponding data type.

If you decide to split semi-structured data into multiple columns while loading, you:

- Can explicitly extract columns from the source data using Snowflake commands for working with semi-structured data.
- If you are using Parquet, Avro, or ORC formats, you can use Snowflake functionality to automatically detect and retrieve column definitions, as explained in more detail in the following section.

You can load semi-structured data similar to loading structured data using file formats and stages. You can use the COPY statement to load data from a staged file into the target table, as discussed in Chapter 7. For example, when you are loading JSON data, you can specify the file format as `file_format = (type = 'JSON')` and the target table column data type as VARIANT.

You can also insert data into the target table by using the PARSE_JSON function, which converts source data such as JSON represented as a string into the VARIANT data type that is recognized by Snowflake; for example, `parse_json('{...}')`.

Detection of Column Definitions

Snowflake can automatically detect the schema in semi-structured data files and retrieve column definitions for Parquet, Avro, and ORC file formats. The following SQL functions are provided for this purpose:

- `INFER_SCHEMA`. Detects column definitions in staged data files and retrieves metadata, including column name, data type, ordinal position of the column, whether it is nullable, and so on.
- `GENERATE_COLUMN_DESCRIPTION`. Uses the `INFER_SCHEMA` function output to generate a list of columns in a format that can be used in a `CREATE TABLE` or `CREATE VIEW` statement.
- `USING TEMPLATE`. Can be used as part of a `CREATE TABLE` statement to automatically derive column definitions from the `INFER_SCHEMA` function output.

Let's work with an example. First, to set up a sample Parquet file in a stage, you will create an internal stage named `MY_PARQUET_STAGE`. Then you will select a few rows from the `ORDERS` table in the Snowflake sample database that you will save as a Parquet file into the internal stage, like this:

```
create stage MY_PARQUET_STAGE;

copy into @MY_PARQUET_STAGE from (
  select top 10 * from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.CUSTOMER
) file_format = (type = parquet) single = true header = true;
```

To detect the schema of Parquet files staged in the `MY_PARQUET_STAGE` internal stage, you have to define a named file format and then use the `INFER_SCHEMA` function to view the names, data types, and other properties of columns in the files:

```
create file format MY_PARQUET_FORMAT type = parquet;

select *
  from TABLE(INFER_SCHEMA(
    location => '@MY_PARQUET_STAGE',
    file_format => 'MY_PARQUET_FORMAT'
  )
);
```

To generate column definitions that can be used to create a table, an external table, or a view, use the GENERATE_COLUMN_DESCRIPTION function:

```
select GENERATE_COLUMN_DESCRIPTION(
  array_agg(object_construct(*)), 'table') as columns
  from TABLE(INFER_SCHEMA(
    location => '@MY_PARQUET_STAGE',
    file_format => 'MY_PARQUET_FORMAT'
    )
  );
```

The output of this command can be used by manually copy/pasting the output into the DDL script. A more seamless option when creating a table is to use the USING TEMPLATE keyword to read the column definitions directly from the INFER_SCHEMA function, like this:

```
create table MY_NEW_TABLE
  using template(
    select array_agg(object_construct(*))
    from TABLE(INFER_SCHEMA(
      location => '@MY_PARQUET_STAGE',
      file_format => 'MY_PARQUET_FORMAT'
    )
));
```

Querying Semi-Structured Data

Once you have semi-structured data loaded into a Snowflake table in a VARIANT column, you can query it. This section explores the various ways to query semi-structured data using JSON sample data. For this exercise, create a table named FRUIT_INFO and load some sample data into it, as follows:

```
create table FRUIT_INFO (fruitbasket variant) as
select parse_json('{
  "farmersmarket": "River Edge",
  "marketinfo": {"address": "Sloan Blvd 55",
    "phone": "555-1234"},
  "fruits": [
    {"fruitname": "Apple",
     "varieties": ["Red delicious", "Gala", "Jonathan"]
    },
    {"fruitname": "Grape",
     "varieties": ["Thompson seedless", "Concord"]
    },
    {"fruitname": "Banana",
     "varieties": ["Chiquita"]}
  ]
}')
union all
select parse_json('{
  "farmersmarket": "Columbus Heights",
  "marketinfo": {"address": "Cross St 79",
    "phone": "555-9876", "contact": "Alice"},
  "fruits": [
    {"fruitname": "Apple",
     "varieties": ["Gala", "Braeburn"]},
```

```
    {"fruitname": "Grape",
     "varieties": ["Muscat", "Concord"]}
  ]
}');
```

The FRUIT_INFO table contains one column named FRUITBASKET of data type VARIANT and two records. The sample data includes first-level elements, for example "farmersmarket": "River Edge". It also includes nested objects, for example "marketinfo": {"address": "Sloan Blvd 55", "phone": "555-1234"}. And it includes nested arrays, for example ["Red delicious", "Gala", "Jonathan"].

When you query the table with a simple SELECT statement:

```
select * from FRUIT_INFO;
```

You'll see something like Figure 8-2.

Figure 8-2. *Sample output from the FRUIT_INFO table*

The output in Figure 8-2 is not very useful for reporting or further analyses. It's better to query the table so that the results are structured in tabular format.

Selecting First-Level Elements

You can select first-level elements from semi-structured data by adding a colon after the VARIANT column name and any first-level element name. For example, to select a list of all farmers markets, use the following query:

```
select fruitbasket:farmersmarket
from FRUIT_INFO;
```

The result is shown in Figure 8-3.

FRUITBASKET:FARMERSMARKET
"River Edge"
"Columbus Heights"

Figure 8-3. *Result of selecting the FARMERSMARKET first-level element*

As you can see in Figure 8-3, the output column values are enclosed in double quotes. This is because the query returns output in data type `VARIANT`, which is not a string. Rather, `VARIANT` values contain strings. You must explicitly convert the data type if you are looking for just the string values using the following syntax (the output column is called `FARMERSMARKET`):

```
select fruitbasket:farmersmarket::string as FARMERSMARKET
from FRUIT_INFO;
```

Another first-level element in the `FRUIT_INFO` data is `MARKETINFO`. Adding this column to the `SELECT` statement:

```
select fruitbasket:farmersmarket::string as FARMERSMARKET,
  fruitbasket:marketinfo as MARKETINFO
from FRUIT_INFO;
```

Gives the output shown in Figure 8-4.

FARMERSMARKET	MARKETINFO
River Edge	{ "address": "Sloan Blvd 55", "phone": "555-1234" }
Columbus Heights	{ "address": "Cross St 79", "contact": "Alice", "phone": "555-9876" }

Figure 8-4. *Result of selecting the MARKETINFO element*

As you can see in Figure 8-4, the MARKETINFO element is an object. You can extract elements from this object using dot notation, like this:

```
select fruitbasket:farmersmarket::string as FARMERSMARKET,
  fruitbasket:marketinfo.address::string as ADDRESS,
  fruitbasket:marketinfo.phone::string as PHONE,
  fruitbasket:marketinfo.contact::string as CONTACT
from FRUIT_INFO;
```

The result of this query is shown in Figure 8-5.

FARMERSMARKET	ADDRESS	PHONE	CONTACT
River Edge	Sloan Blvd 55	555-1234	null
Columbus Heights	Cross St 79	555-9876	Alice

Figure 8-5. *Result of selecting elements from the MARKETINFO element*

An alternative notation for selecting elements is the bracket notation. For example, you can achieve the same results as in Figure 8-5 using this query:

```
select fruitbasket['farmersmarket']::string as FARMERSMARKET,
  fruitbasket['marketinfo']['address']::string as ADDRESS,
  fruitbasket['marketinfo']['phone']::string as PHONE,
  fruitbasket['marketinfo']['contact']::string as CONTACT
from FRUIT_INFO;
```

Regardless of which notation you use, element names in semi-structured data are case-sensitive.

Using the FLATTEN Function

Nested arrays within a semi-structured object can be parsed using the FLATTEN function. This is a table function that returns a row for each object in the array, which you can use together with the LATERAL modifier to join the rows with any other data at a higher level of the object hierarchy.

For example, to get all of the fruits in each farmers market as a separate row in the output, use the following query:

```
select fruitbasket:farmersmarket::string as farmersmarket,
  value
from FRUIT_INFO,
LATERAL FLATTEN(input => fruitbasket:fruits);
```

The result of this query is shown in Figure 8-6.

FARMERSMARKET	VALUE
River Edge	{ "fruitname": "Apple", "varieties": ["Red delicious", "Gala", "Jonathan"] }
River Edge	{ "fruitname": "Grape", "varieties": ["Thompson seedless", "Concord"] }
River Edge	{ "fruitname": "Banana", "varieties": ["Chiquita"] }
Columbus Heights	{ "fruitname": "Apple", "varieties": ["Gala", "Braeburn"] }
Columbus Heights	{ "fruitname": "Grape", "varieties": ["Muscat", "Concord"] }

Figure 8-6. *Result of using the FLATTEN function on a nested array*

As you can see in Figure 8-6, the result is a separate row for each of the nested array elements in the "fruits" object. The output of the FLATTEN function is referenced as VALUE.

You can take this query one step further by listing each of the fruit varieties in a separate row. You do this by adding another LATERAL FLATTEN construct using the "varieties" array as the parameter. Since you will have two FLATTEN functions in the same query, you have to alias

them to distinguish the VALUE outputs from each of the functions. This is how you would write the query, using alias fs for the first FLATTEN function and alias fv for the second FLATTEN function:

```
select fruitbasket:farmersmarket::string as farmersmarket,
  fs.value:fruitname::string as fruitname,
  fv.value::string as variety
from FRUIT_INFO,
LATERAL FLATTEN(input => fruitbasket:fruits) fs,
LATERAL FLATTEN(input => fs.value:varieties) fv;
```

The result of this query is shown in Figure 8-7.

FARMERSMARKET	FRUITNAME	VARIETY
River Edge	Apple	Red delicious
River Edge	Apple	Gala
River Edge	Apple	Jonathan
River Edge	Grape	Thompson seedless
River Edge	Grape	Concord
River Edge	Banana	Chiquita
Columbus Heights	Apple	Gala
Columbus Heights	Apple	Braeburn
Columbus Heights	Grape	Muscat
Columbus Heights	Grape	Concord

Figure 8-7. *Result of using the FLATTEN function on two levels of nested arrays*

Using the GET Function

Another way to access elements of objects or arrays is using the GET function. One format of the GET function is to use an object or a variant and a field name as input arguments. For example, using the FRUIT_INFO table, you can list the values of the "farmersmarket" element from the FRUITBASKET column with the following syntax:

```
select GET(fruitbasket, 'farmersmarket')
from FRUIT_INFO;
```

The result of this query is the same as Figure 8-3.

Another format of the GET function is to use an array or a variant and an index as input arguments. For example, to get only the first element of the "fruits" array, you can use the following syntax (note that the first element of an array has index 0):

```
select GET(fruitbasket:fruits, 0)
from FRUIT_INFO;
```

The result of this query is shown in Figure 8-8.

GET(FRUITBASKET:FRUITS, 0)
{ "fruitname": "Apple", "varieties": ["Red delicious", "Gala", "Jonathan"] }
{ "fruitname": "Apple", "varieties": ["Gala", "Braeburn"] }

Figure 8-8. *Result of using the GET function with an index*

The ARRAY_SIZE function returns the size of an array and you can use it in combination with the GET function to retrieve the last element of an array, like this:

```
select GET(fruitbasket:fruits,
  ARRAY_SIZE(fruitbasket:fruits) - 1)
from FRUIT_INFO;
```

Querying Semi-Structured Data from a Stage

Snowflake supports querying semi-structured data directly from staged files. For example, supposing that the sample data that was used to populate the FRUIT_INFO table was stored in a file named fruit_info.json.gz in an internal stage, you could query the data directly, something like this:

```
select ...
from @MY_STAGE/fruit_info.json.gz
  (file_format => 'MY_JSON_FORMAT');
```

Although you can query the data in semi-structured format and produce tabular output suitable for storing in a structured table with columns, this is not always practical or recommended. Depending on the use case, it might be more efficient to store semi-structured data in a VARIANT data type and query it later. In some cases, Snowflake recommends extracting semi-structured data into separate columns, especially for the following types of data:

- Dates and timestamps
- Numbers within strings
- Arrays

Dates, timestamps, and numbers within strings are stored as strings when loaded into a VARIANT column. This means that queries using such columns would take a longer time to execute due to internal data type conversion. They might also consume more storage space because storing values as strings does not take advantage of more efficient storage options used for storing dates, timestamps, or numbers.

NULL Values

Snowflake supports two types of NULL values in semi-structured data:

- **SQL NULL.** Similar to structured data, SQL NULL in semi-structured data also means that the value is either missing or unknown.
- **JSON NULL or VARIANT NULL.** To distinguish JSON NULL values from SQL NULL values, JSON NULL values are stored in a VARIANT column as strings that contain the value "null".

To illustrate both types of NULL values, create a sample table that contains two JSON documents. In the first JSON document, the PHONE tag contains a JSON null value. In the second JSON document, the PHONE tag is not provided. Create the table using the following command:

```
create table MY_JSON_TABLE (my_json_column variant) as
select parse_json('{
  "farmersmarket": "River Edge",
  "marketinfo": {"address": "Sloan Blvd 55", "phone": null}
}')
union all
select parse_json('{
  "farmersmarket": "Columbus Heights",
  "marketinfo": {"address": "Cross St 79"}
}');
```

To query the first level elements from this table, use this command:

```
select my_json_column:farmersmarket as farmersmarket,
  my_json_column:marketinfo.address as address,
  my_json_column:marketinfo.phone as phone
from MY_JSON_TABLE;
```

You will see results shown in Figure 8-9.

FARMERSMARKET	ADDRESS	PHONE
"River Edge"	"Sloan Blvd 55"	null
"Columbus Heights"	"Cross St 79"	null

Figure 8-9. *First level elements*

As you can see in Figure 8-9, the first row contains the value "null" in the PHONE column as it was provided in the JSON document. The second row contains the SQL NULL value because the PHONE tag was not provided.

To convert JSON null values to SQL null values, cast them as strings, like this:

```
select my_json_column:farmersmarket::string as farmersmarket,
  my_json_column:marketinfo.address::string as address,
  my_json_column:marketinfo.phone::string as phone
from MY_JSON_TABLE;
```

You will see the results shown in Figure 8-10.

FARMERSMARKET	ADDRESS	PHONE
River Edge	Sloan Blvd 55	null
Columbus Heights	Cross St 79	null

Figure 8-10. *First-level elements cast to string data type*

Now both rows contain SQL null values in the PHONE column.

Unstructured Data

Unstructured data refers to information such as images, video, and audio that cannot be loaded into a data model based on a schema. You can work with unstructured data in Snowflake in the following ways:

- When unstructured data files are stored in one of the supported storage providers, you can access them securely from Snowflake.
- You can share file access URLs with users and applications.
- You can load metadata such as file access URLs and other properties into Snowflake tables.

Unstructured data files can be loaded into external or internal stages. Files in stages can be accessed using the following types of URLs:

- **Scoped URL.** Allows administrators to give temporary access to a staged file without granting privileges to the stage. The URL persists in the query results cache and will expire after 24 hours.
- **File URL.** Represents a permanent URL to a staged file. The file can be accessed only by a role that has been granted access privileges on the stage.
- **Pre-signed URL.** This URL can be used to access a file without authenticating to Snowflake using a pre-signed access token. The URL will expire when this access token expires, which is configurable using the `EXPIRATION_TIME` parameter.

Working with Pre-Signed URLs

This section shows an example of a pre-signed URL. First create an internal stage using the `CREATE STAGE` command:

```
create stage MY_INTERNAL_STAGE;
```

Using the SnowSQL `PUT` command, upload an unstructured file. This example uploads an image file named `snowflake.png`, stored in the `C:\myimgfiles\` folder on the local file system, to the internal stage:

```
put file://C:\myimgfiles\snowflake.png @MY_INTERNAL_STAGE;
```

To check that the file has been loaded successfully, list the contents of the internal stage:

```
list @MY_INTERNAL_STAGE;
```

You should see results similar to Figure 8-11.

name	size	md5
my_internal_stage/snowflake.png.gz	29,264	e1844875fbe8b0497

Figure 8-11. *List the internal stage with an unstructured file*

As you can see in Figure 8-11, the file extension is `.gz`, which indicates that the file has been automatically compressed in the internal stage.

To get a pre-signed URL for the file, use the `GET_PRESIGNED_URL` function, like this:

```
select GET_PRESIGNED_URL(@MY_INTERNAL_STAGE, 'snowflake.
png.gz');
```

This statement will return a URL that can be used to download or access the file without authenticating to Snowflake or passing an authorization token. This URL can be used for business intelligence applications or reporting tools to allow them to display the unstructured file contents.

Working with Scoped or File URLs

Staged unstructured data files can also be retrieved using the REST API `GET /api/files/`. To generate a scoped URL for the staged file, use the `BUILD_SCOPED_FILE_URL` function. Alternatively, to generate a file URL, use the `BUILD_STAGE_FILE_URL` function.

The application that calls the API must authenticate to Snowflake using either OAuth or key pair authentication. The authorization to access files differs depending on whether a scoped URL or file URL is sent in the `GET` request. With a scoped URL, only the user who generated the scoped URL can use the URL to access the referenced file. With a file URL, any role that has sufficient privileges on the stage can access the file.

Another way to read and process unstructured data in staged files is to use Java UDFs (user-defined functions) or tabular Java UDFs (user-defined table functions).

Materialized Views

A materialized view is defined by a `SELECT` statement used to pre-compute and persist the query result. This is useful for queries that are used frequently or for queries that would take a long time to execute. Because the data is pre-computed, users get results much faster as compared to executing the query used to define the materialized view.

Note Materialized views require Snowflake Enterprise edition or higher.

Although materialized views can significantly improve query performance, you should carefully consider whether to use them. Because the results are pre-computed and stored persistently, they consume storage cost and their maintenance consumes credits.

Snowflake recommends that materialized views be used when the query results contain a small subset of data from the table on which the view is defined. Another place to use materialized views is with queries that contain complex transformations or other constructs that take a long time to execute, such as:

- Querying semi-structured data
- Aggregating large volumes of data
- Querying external tables

To reduce cost for materialized view maintenance, another consideration is to use a materialized view when the view's base table does not change frequently.

Materialized views are maintained by Snowflake using a background service that continuously updates the persisted query results whenever the table used in the query is changed. This background service may take some time to complete, which means that the persisted results may not always be completely up to date. However, Snowflake guarantees that queries against the materialized view are always current, regardless of the background service completion. When a user queries a materialized view that is not completely up to date, Snowflake will try to update the materialized view immediately or use the already persisted materialized view and combine it with the freshest data from the table used in the query.

Comparison with Regular Views

Snowflake recommends creating a materialized view rather than a regular view when all of the following criteria are met:

- Query results of the materialized view don't change often.
- The materialized view is queried significantly more often than the query results change.
- The query consumes a large amount of resources, which means that when the query results are materialized, resources are consumed only once.

On the other hand, Snowflake recommends creating a regular view rather than a materialized view when any of the following criteria are met:

- Query results of the materialized view change often.
- The materialized view is not queried often when compared to the rate at which the results change.
- The query does not consume a large amount of resources, so it can be executed on the fly whenever needed.

Comparison with Tables and Cached Results

Materialized views are similar to tables in some ways because they allow creating clustering keys and they consume storage. Materialized views also have some similarities with cached query results because both enable storing query results for future reuse. Both materialized views and regular views can be used for data security because they can hide sensitive data and expose only data that is allowed to be queried by users.

Table 8-3 summarizes the key similarities and differences among tables, regular views, cached query results, and materialized views.

Table 8-3. *Comparison of Tables, Regular Views, Cached Query Results, and Materialized Views*

	Table	Regular View	Materialized View	Cached Query Result
Improves performance			Yes	Yes
Used for security		Yes	Yes	
Simplifies query logic		Yes	Yes	
Supports clustering	Yes		Yes	
Uses storage	Yes		Yes	
Uses credits for maintenance			Yes	

Materialized Views in the Query Optimizer

Snowflake's query optimizer can detect when you have a materialized view defined for a given SQL query. In such cases, the optimizer can automatically rewrite queries so that they use the materialized view instead of executing the query. This works even in complex queries, where the optimizer can use a materialized view instead of just one table that participates in the query.

The optimizer will not choose a materialized view over the query by default. If the optimizer finds a more efficient way to execute the query without using the materialized view, it will do so. You can always check whether the optimizer decided to use a materialized view instead of executing the query by reviewing the Query Profile.

Privileges Related to Materialized Views

To create a materialized view, you need the `CREATE MATERIALIZED VIEW` privilege on the schema that will contain the materialized view.

Because materialized views are database objects, they are owned by a role. To allow access to a materialized view to other roles, respective grants must be assigned. However, when Snowflake's query optimizer rewrites a query to use the materialized view instead of the base table, the user who is executing the query does not need privileges to use the materialized view explicitly because query rewriting is handled by Snowflake internally.

As with regular views, when a user executes a query against a materialized view, they must have privileges granted only on the view, but not necessarily on the objects that are used by the query that defines the view.

Creating Materialized Views

To create a materialized view, use the `CREATE MATERIALIZED VIEW` command. This example creates a materialized view named `LINEITEM_RAIL` based on the `LINEITEM` table from the Snowflake sample database. The materialized view will filter for line items whose shipping mode equals `'RAIL'`. This is the SQL command to create the materialized view:

```
create materialized view LINEITEM_RAIL as
select *
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.LINEITEM
where l_shipmode = 'RAIL';
```

Snowflake recommends using the fully qualified table name in the query that defines the materialized view to ensure there is no ambiguity about the reference.

The filter specified in the materialized view can be utilized by queries against the base table that use the same filter or a more restrictive filter. For example, the following query against the LINEITEM base table that filters on both the shipping date and the shipping mode might use the materialized view instead of the base table:

```
select *
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.LINEITEM
where year(l_shipdate) = 2019
and l_shipmode = 'RAIL';
```

Depending on the size of the base table, the CREATE MATERIALIZED VIEW statement might take a long time to complete. A background process will automatically maintain the data in the materialized view. You can monitor the last time that a materialized view was refreshed by checking the REFRESHED_ON and BEHIND_BY columns in the output of the SHOW MATERIALIZED VIEWS command:

```
show materialized views;
```

By using the RESULT_SCAN function, as described earlier in this chapter, you can post-process the result of the previous query:

```
select "name", "refreshed_on", "behind_by"
from table(result_scan(last_query_id()))
where "name" = 'LINEITEM_RAIL';
```

The result of this query is shown in Figure 8-12.

name	refreshed_on	behind_by
LINEITEM_RAIL	2022-04-15 05:42:49.144 -0700	0s

Figure 8-12. *Result of post-processing the SHOW MATERIALIZED VIEWS command*

Other ways to view information about materialized views include:

- Using the SHOW VIEWS command. The command returns information about regular views and materialized views.
- Querying the TABLES view in INFORMATION_SCHEMA. The result indicates 'MATERIALIZED VIEW' in the TABLE_TYPE column.

Note INFORMATION_SCHEMA.VIEWS does not show materialized views. Materialized views are shown by INFORMATION_SCHEMA.TABLES.

You can suspend the automatic background maintenance of a materialized view by executing the ALTER MATERIALIZED VIEW command with the SUSPEND keyword, like this:

```
alter materialized view LINEITEM_RAIL suspend;
```

This command will suspend the maintenance of a materialized view and mark it as invalid. You should not query the materialized view when maintenance has been suspended because it may not be up to date with the base table. You can resume maintenance by executing the ALTER MATERIALIZED VIEW command with the RESUME keyword.

When a base table for a materialized view is dropped, the materialized view will become suspended. If you are re-creating the base table and would like to keep the materialized view with the same definition, you must re-create the materialized view as well.

To monitor the maintenance costs for materialized views, you can query the MATERIALIZED_VIEW_REFRESH_HISTORY table function in INFORMATION_SCHEMA, like this:

```
select * from table(INFORMATION_SCHEMA.MATERIALIZED_VIEW_
REFRESH_HISTORY());
```

The result of this query will show the materialized view name along with the credits used, the start time, and the end time of materialized view maintenance.

Limitations on Creating Materialized Views

Materialized views are maintained by Snowflake and, as such, you cannot perform DML operations such as `INSERT`, `UPDATE`, or `DELETE` on them. You also cannot truncate a materialized view.

Similar to other relational databases, there are limitations with respect to creating materialized views, such as:

- A materialized view can query only a single table.
- A query that defines a materialized view cannot include any joins or nested subqueries.
- A materialized view cannot include UDFs (user defined functions), window functions, `HAVING` clauses, `ORDER BY` clauses, or `LIMIT` clauses.
- Many aggregate functions are not allowed in a materialized view definition (refer to the Snowflake documentation for a detailed list of the supported aggregate functions).

Adding Clustering Keys to Materialized Views

To increase performance of a materialized view, you can define a clustering key if needed. When you do that, keep in mind that a materialized view is defined as a query on a base table and that base table may already have a clustering key. If so, a clustering key on a materialized view will be beneficial only if it is defined on a different column than the clustering key on the base table.

Snowflake recommends that, if you want to define a clustering key on the base table and the corresponding materialized view, you create the clustering key on the materialized view first. Then monitor performance and cost to assess the benefits before adding a clustering key on the table as well.

Another recommendation with respect to clustering keys on materialized views is that you load data into the base table first, then create the materialized view with the clustering defined. This will reduce credit consumption because the clustering will be performed at the same time as the initial materialization.

Summary

This chapter reviewed the Snowflake-specific SQL extensions that apply to standard data, semi-structured data, and unstructured data.

It explained the Snowflake data sampling and estimating functions that can be used on standard data. It discussed how to post-process query results in Snowflake. It reviewed semi-structured data, including data types used for storing semi-structured data, as well as loading and querying semi-structured data. It also discussed how to work with unstructured data. Finally, the chapter described materialized views that can be used to improve performance of workloads composed of common, repeated query patterns.

The next chapter looks at various aspects of continuous data protection in Snowflake.

CHAPTER 9

Data Protection

Snowflake provides many features that protect data against human error, malicious acts, and software failure. Chapter 4 covers some of these features, such as network policies, user authentication, and access control.

This chapter introduces more features related to continuous data protection. It discusses time travel, which allows you to access historical data as it existed before it was changed or deleted. The chapter reviews the failsafe feature, which ensures that data is protected against a system failure or other unexpected events. It also looks at data encryption as a means to prevent third parties from reading data while in transit to and from Snowflake. Finally, the chapter also provides an overview of zero-copy cloning and describes database replication.

Time Travel

Snowflake time travel enables querying, cloning, and restoring historical data in tables, schemas, and databases. All Snowflake editions support time travel, although with different limitations. Snowflake Standard Edition supports time travel for up to one day. Snowflake Enterprise Edition and higher editions support time travel for up to 90 days.

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_9

Snowflake time travel allows you to perform the following activities, depending on the duration of time that you have enabled time travel:

- Query data as it was in the past.
- Clone tables, schemas, and databases at or before points in time in the past.
- Restore tables, schemas, and databases that have been dropped within the duration of time that you have enabled time travel.

After the time travel period ends, the data is moved into Snowflake failsafe, where time travel can no longer be performed.

Keep in mind that when using time travel, additional cost is incurred due to the additional data storage required to store historical data.

Data Retention Period

The amount of time that data is available for time travel is defined by the data retention period. The data retention period is measured in days. It determines how long Snowflake preserves historical data after it has been modified, deleted, or dropped.

The standard retention period that is automatically enabled in all Snowflake accounts is one day, where each day represents a 24-hour time period. Depending on the Snowflake edition, retention periods can be set for longer periods of time as follows:

- The retention period in Snowflake Standard Edition can be set to zero or one day, either for the entire account or for individual objects such as databases, schemas, and tables.
- The retention period in Snowflake Enterprise Edition and higher editions can be set to any value between 0 and 90 days for permanent objects such as databases,

schemas, or tables. The retention period for transient and temporary databases, schemas, or tables can be set to zero or one day.

When an object has a retention period of zero days, this means that time travel is not available for that object.

Snowflake administrators with the `ACCOUNTADMIN` role can specify the data retention period for time travel at various levels:

- The `DATA_RETENTION_TIME_IN_DAYS` parameter can be set at the account level to define the default retention period for the entire account.
- The `DATA_RETENTION_TIME_IN_DAYS` parameter can also be set at the database, schema, or table level. In this case, it takes precedence over the data retention period that was defined for the account.
- The `MIN_DATA_RETENTION_TIME_IN_DAYS` parameter can be set at the account level to define the minimum retention period for the entire account. When this parameter is set at the account level, it determines the minimum data retention period for all objects in the account, regardless of the value that was set in the `DATA_RETENTION_TIME_IN_DAYS` parameter. By setting this parameter, Snowflake administrators can ensure that all objects created by users have a minimum retention period for time travel.

Even when an administrator sets the `DATA_RETENTION_TIME_IN_DAYS` parameter value to `0` at the account level, this doesn't disable time travel in the account. This parameter only defines the default value at the account level, but users can still create databases, schemas, and tables with data retention periods of their choosing, limited by the Snowflake edition that they are using.

Users can create databases, schemas, and tables with a data retention period of zero days to indicate that no time travel is required for the objects. However, if the `MIN_DATA_RETENTION_TIME_IN_DAYS` is set at the account level, this parameter value will take precedence, allowing the objects to have time travel as defined in this parameter.

Snowflake recommends that the data retention period be set to one day for all objects. This ensures that users can restore accidentally dropped objects or previous versions of data that were updated erroneously within the last day.

To define the retention period for a database, schema, or table, the `DATA_RETENTION_TIME_IN_DAYS` keyword can be used together with the `CREATE` or `ALTER` command. For example, to set the data retention period to 90 days when creating a database named `SALES`, you would execute the following command:

```
create database SALES DATA_RETENTION_TIME_IN_DAYS = 90;
```

Retention periods set at parent objects are inherited by child objects within the object hierarchy. For example, when a retention period is defined for a database, all schemas and tables in the database inherit the retention period. Similarly, when a retention period is defined for a schema, all tables in the schema inherit the retention period.

You can change the data retention period for a table using the `ALTER` command. When you do so, this affects data that is currently in time travel as follows:

- **Data retention period is increased.** The data that is currently in time travel will be retained for the longer retention period.
- **Data retention period is decreased.** If the data currently in time travel is still within the shorter retention period, it will remain in time travel. If not, it will move to failsafe.

To change the retention period for a database, schema, or table, execute the ALTER command with the DATA_RETENTION_TIME_IN_DAYS keyword. For example, to change the retention period for the SALES database to 30 days, execute the following command:

```
alter database SALES set DATA_RETENTION_TIME_IN_DAYS = 30;
```

When you change the data retention period for the account, for a database or for a schema, all child objects lower in the object hierarchy inherit the retention period unless they have a retention period explicitly defined.

When the data retention period for time travel expires, data is moved to failsafe. This is done by Snowflake using a background process, which may take some time to complete. Until all data is moved to failsafe, it might still be available in time travel.

When parent and child objects have different data retention periods defined, this affects how the retention periods are honored when dropping objects. For example, when a database is dropped, the child objects are dropped as well, regardless if their retention periods have been set differently. Similarly, when a schema is dropped, the tables are dropped as well, regardless if their retention periods have been set differently. In such cases, Snowflake recommends that you drop the child objects first, because this action will honor the child object retention period individually, then drop the parent object.

Querying Historical Data

To support time travel, Snowflake provides the AT | BEFORE clause as an SQL extension. This clause can be used in SELECT and CLONE statements. The clause uses one of the following parameters to limit the historical data that the query will return:

- TIMESTAMP
- OFFSET, defined by the time difference in seconds from the current timestamp
- STATEMENT, defined by the query ID of a statement

The UNDROP command used to restore tables, schemas, and databases that have been dropped will also retrieve the objects from time travel.

Let's work with some examples. Assuming that you created a database named SALES using the command in the previous section, create a schema named SALESDATA using this command:

```
create schema SALESDATA DATA_RETENTION_TIME_IN_DAYS = 1;
```

You can specify any value up to 90 days for the DATA_RETENTION_TIME_IN_DAYS parameter value, depending on the Snowflake edition you are using. For the following time travel exercises, make sure that this value is set to at least one day so that you have some time travel data to work with.

Create a table named NATION by selecting data from the Snowflake sample database:

```
create table NATION as
select * from SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.NATION;
```

You can check the record for France in the NATION table using a regular SQL SELECT statement, like this:

```
select * from NATION where n_name = 'FRANCE';
```

You will see a generated comment that you copied from the Snowflake sample database—something like Table 9-1.

Table 9-1. *Selecting from the NATION Table*

P_NATIONKEY	N_NAME	N_REGIONKEY	N_COMMENT
6	FRANCE	3	Fulfill final requests…

Next, update the comment for the France record in the NATION table:

```
update NATION
set n_comment = 'Comment for France'
where n_name = 'FRANCE';
```

After you update the comment in the record for France, find the query ID of the UPDATE statement. You can do this by clicking the three dots at the top right in the pane next to the results pane in Snowsight and selecting Copy Query ID, as shown in Figure 9-1.

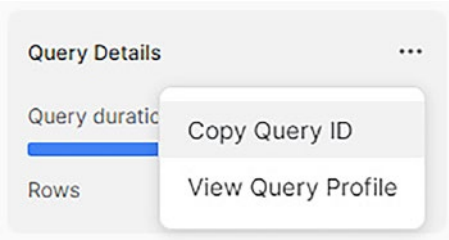

Figure 9-1. *The Copy Query ID option in Snowsight*

Paste this query ID somewhere on your worksheet because you will need it later.

After updating, you can again check the record for France using the same SELECT statement as before:

```
select * from NATION where n_name = 'FRANCE';
```

The result of this query is shown in Table 9-2, where you can see that the comment was updated.

Table 9-2. *Selecting from the NATION Table After Updating the Comment*

P_NATIONKEY	N_NAME	N_REGIONKEY	N_COMMENT
6	FRANCE	3	Comment for France

To query historical data from the NATION table as it was before you updated the record, use the BEFORE option with the statement parameter, like this:

```
select *
from NATION before(statement => '01a7-xxxx-yyyy-a166')
where n_name = 'FRANCE';
```

Be sure to use your own query ID that you retrieved earlier using the Copy Query ID option in the statement parameter.

The result of the previous query is the same as the result shown in Table 9-1, before you updated the NATION table.

Similarly, you can query the NATION table in the past using the AT option with the offset parameter. For example, to query the table as it was 60 seconds ago, use the following syntax:

```
select *
from NATION at(offset => -60)
where n_name = 'FRANCE';
```

An example of querying the table using the AT option with the timestamp parameter is as follows:

```
select *
from NATION at(timestamp => '2022-04-15 11:30:00'::timestamp)
where n_name = 'FRANCE';
```

This query returns data from the NATION table as it was at the specified timestamp. Be careful when using the timestamp parameter to be sure that you have correctly identified the time zone and converted to the timestamp data type.

If the value of the timestamp, offset, or statement parameter specified in the AT | BEFORE option is beyond the data retention period for the table, the query fails and returns an error.

Cloning Historical Objects

Snowflake databases, schemas, and tables can be cloned using the CLONE keyword with the respective CREATE statement command. The AT | BEFORE clause can be used to specify the historical point in time that will be used as the reference for cloning.

For example, to create a table named NATION_HISTORICAL by cloning the NATION table at the point in time represented by the timestamp parameter in the AT clause, use the following syntax:

```
create table NATION_HISTORICAL
clone NATION at(timestamp => '2022-04-15 11:30:00'::timestamp);
```

To create a schema named SALESDATA_HISTORICAL by cloning the SALESDATA schema and all objects contained in the schema at the point in time represented by the offset parameter in the AT clause, use the following syntax:

```
create schema SALES.SALESDATA_HISTORICAL
clone SALES.SALESDATA at(offset => -3600);
```

This statement uses the value -3600 for the OFFSET parameter, which defines the point in time 3,600 seconds (or one hour) ago from the current timestamp.

Similarly, you can clone a database using a point in time in the time travel history.

Cloning a database or a schema will fail in the following situations:

- If the point in time for cloning exceeds the data retention time of the objects to be cloned.
- If the point in time for cloning is before the timestamp when the object was created.

Cloning is discussed in more detail later in this chapter.

Dropping and Restoring Objects

To restore objects that have been dropped, Snowflake uses time travel to retain the object in case it must be restored. The data retention period is used to determine the length of time that the object will be retained in time travel.

To drop a table, schema, or database, use the DROP command for the respective object. As an example, drop the NATION table that you created earlier:

```
drop table NATION;
```

You can view dropped tables, schemas, and databases using the SHOW command with the HISTORY keyword. For example, to view dropped tables in the SALES database and SALESDATA schema, execute this command:

```
show tables history in SALES.SALESDATA;
```

This command will return a list of all current tables in the SALESDATA schema of the SALES database, as well as tables that have been dropped. You can tell that a table has been dropped by checking the DROPPED_ON column in the output. This column contains a value when the table has been dropped, but is NULL otherwise.

To view dropped schemas or databases, use one of the following commands, respectively:

```
show schemas history in SALES;
show databases history;
```

Similar to the SHOW TABLES output, the value in the DROPPED_ON column contains the date and time when the object was dropped. If an object has been dropped more than once, it will be included in the output with a separate row for each time it was dropped.

After the retention period for an object has expired and it is no longer available in time travel, the `SHOW HISTORY` command output will no longer include that object.

A dropped object that is still available in time travel can be restored using the `UNDROP` command. For example, to undrop the `NATION` table that you dropped earlier, execute the following command:

```
undrop table NATION;
```

Similarly, to undrop a schema or a database, you can use the following commands, respectively:

```
undrop schema SALESDATA;
undrop database SALES;
```

When you undrop an object, you cannot rename it to a different name; you can only restore it with the name of the object. If, after dropping an object, you create a new object with the same name as the dropped object, you cannot undrop the previously dropped object. You must remove the newly created object with the same name first, then restore the dropped object.

To restore an object, you must have the following privileges:

- `OWNERSHIP` privilege on the object
- `CREATE` privilege in the database or schema where the object is restored

You can restore objects only in your current database or schema.

Failsafe

Failsafe is used to protect historical data in the event of a system failure, a security breach, or other unexpected events. It is not meant as an

extension for time travel because it is not accessible by users. Failsafe can be accessed only by Snowflake to restore data in extreme situations.

After the time travel retention period ends, data is moved into failsafe for an additional seven days, during which time it can be recovered by Snowflake. The seven-day failsafe period is not configurable by users and cannot be extended.

A graphical representation of how failsafe fits into Snowflake's continuous data protection lifecycle is shown in Figure 9-2.

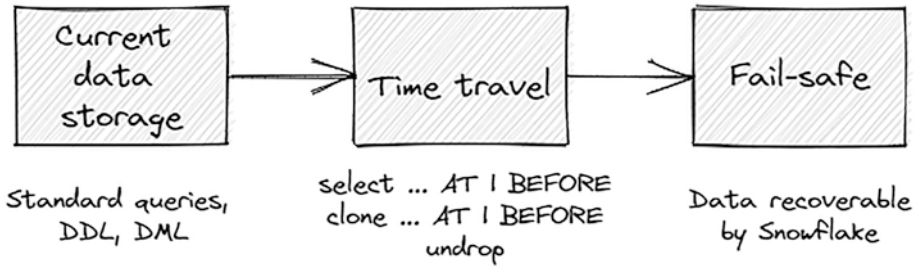

Figure 9-2. *Continuous data protection lifecycle*

Account administrators can view the total amount of data storage for their account, including data in time travel and failsafe, via the Snowsight user interface. To view data storage in your account, ensure that `ACCOUNTADMIN` is selected as your current role. Then click the Admin option in the left menu and choose Usage. In the window that appears, click the All Usage Types drop-down list and choose Storage. Chapter 5 contains more information about data storage monitoring.

Cost for Time Travel and Failsafe

Historical data that is stored in time travel and failsafe incurs storage costs. Snowflake calculates this cost based on 24-hour time periods. To minimize the amount of storage, Snowflake stores only data that has been changed

or deleted in time travel and failsafe. Only tables that have been dropped are retained in their entirety so that they can be restored when needed.

Temporary and transient tables are used to lower storage costs for time travel and failsafe as compared to permanent tables. When the data retention time for a transient table is set to one day, it will incur time travel related data storage cost for that day. When the data retention period for a temporary table is set to one day, it will also incur time travel related data storage cost, but only while the table exists.

Note Transient and temporary tables have no failsafe period.

With respect to managing costs for time travel and failsafe, Snowflake offers some guidelines to help you choose whether to use permanent, temporary, or transient tables:

- Temporary tables exist only in the session in which they were created. After such tables are dropped or when the session ends, temporary tables are not retained in time travel or failsafe and consequently cannot be recovered.
- Transient tables can have a time travel retention period defined and can be recovered during this time from time travel. As they have no failsafe period, they cannot be recovered after the time travel period expires. Tables that are used in ETL processes to store intermediate results can be defined as transient.
- Tables that are required by users, such as data warehouse fact and dimension tables, should always be protected by time travel and failsafe in case of unexpected events. As such, they should be created as permanent.

Data Encryption

When Snowflake is hosted in a secure virtual private cloud (VPC) on AWS or in a virtual network (VNet) on Azure, end-to-end encryption (E2EE) ensures that data is secure against being read by third parties while in transit to and from Snowflake. E2EE is supported when loading and unloading data using either external or internal stages.

The following scenarios apply when loading data:

- **Loading from external stages.** Snowflake recommends that data files hosted in cloud storage of a cloud storage provider are encrypted using client-side encryption. If data is not encrypted, Snowflake will encrypt it before loading it into a table.
- **Loading from internal stages.** Data files are automatically encrypted by the Snowflake client prior to being loaded into the internal stage. They will also be encrypted after they have been loaded to the stage.

The following scenarios apply when unloading data:

- **Unloading to an external stage.** Results are optionally encrypted using client-side encryption.
- **Unloading to an internal stage.** Results are automatically encrypted by Snowflake.

Client-Side Encryption

Client-side encryption means that data is encrypted before it is loaded into cloud storage. When data files are encrypted, the user shares the master used for encryption with Snowflake. The user then encrypts each file in cloud storage with a random encryption key, which is in turn encrypted with the master key. Both the encrypted file and the encrypted random

key are uploaded to cloud storage. Since data is encrypted at all times, the cloud storage service or any other third party is unable to view the actual data.

To load client-side encrypted data from an object store, use the `MASTER_KEY` parameter with the `CREATE STAGE` command when creating an external stage. The `MASTER_KEY` parameter requires a 128-bit or 256-bit Advanced Encryption Standard (AES) key encoded in Base64.

For example, to create an external stage in Snowflake that points to an AWS storage location with client-side encryption, execute the following command:

```
create stage ENCRYPTED_EXTERNAL_STAGE
url = 's3://bucket/datafiles/'
credentials = (aws_key_id = 'xxx' aws_secret_key = 'xxx')
encryption = (master_key = 'xxx... = ');
```

You will replace your own AWS credentials and master key information in this command when using it in a real use case.

Once the external stage is created, you can use `COPY` commands to load data into Snowflake tables, as described in Chapter 7.

Similarly, when unloading data from Snowflake, data files are copied into the external stage and encrypted using the master key that was defined in the stage. Users can then download the encrypted data files using a client that supports client-side encryption.

Key Rotation

All data stored in Snowflake is encrypted by default using strong encryption standards. Keys used for encrypting data are rotated by Snowflake on a regular basis, and data is re-encrypted on a regular basis.

Snowflake manages keys for each account in a hierarchical model that consists of four levels:

- Root key
- Account master keys
- Table master keys
- File keys

This hierarchical key model limits the amount of data that each key protects and limits the time when it can be used. In this model, a table master key encrypts a single table, while a file key encrypts a single file. Key rotation ensures that keys are active for a limited time.

Snowflake rotates keys that are older than 30 days. Such keys are retired, and new keys are created in their place. When a key is active, it is used to encrypt data. After it is retired, it is used only to decrypt data. Retired keys are destroyed when they are no longer needed to decrypt data because that data no longer exists.

Periodic Rekeying

Periodic rekeying is only available in Snowflake Enterprise Edition or higher. As opposed to key rotation, where keys are retired but still used to decrypt data, periodic rekeying ensures that keys are destroyed on a regular basis. All the data that was encrypted by the destroyed key is re-encrypted using new keys.

When periodic rekeying is enabled in an account, Snowflake creates a new encryption key for every key that is older than one year, re-encrypts the data that was protected by the previous key, and destroys the previous key. Because the data is re-encrypted using the new key, this key is used to decrypt data going forward.

Users with the `ACCOUNTADMIN` role can enable rekeying using the `PERIODIC_DATA_REKEYING` keyword with the `ALTER ACCOUNT` command, like this:

```
alter account set PERIODIC_DATA_REKEYING = true;
```

Periodic rekeying is a Snowflake process that takes place in the background, without affecting any user workloads, even while data is being rekeyed at the time.

Customer-Managed Keys

Snowflake customers can manage their own customer-managed keys in the cloud provider where their Snowflake account is hosted. Each cloud provider offers its own key management service:

- **AWS.** AWS Key Management Service (KMS)
- **Google Cloud.** Cloud Key Management Service (Cloud KMS)
- **Microsoft Azure.** Azure Key Vault

Snowflake offers a feature named Tri-Secret Secure that enables creating a composite master key by combining the customer-managed key with a Snowflake-managed key.

Some of the reasons that a customer would choose to maintain a customer-managed key include:

- **Access control.** The customer has complete control over their master key and, if the key is not shared with Snowflake, the data cannot be accessed.
- **Disable access.** The customer can disable access to their master key at any time, which will consequently disable access to data in Snowflake. This is especially useful in security incidents when access to data must be disabled immediately.

Snowflake recommends that customer-managed keys are not rotated on a regular basis because it will not be able to decrypt data when keys are changed.

When using customer-managed keys, it is crucial that keys are safeguarded according to best practices associated with confidentiality, integrity, and availability. If a customer-managed key becomes invalid or is unavailable to Snowflake for any reason, all data operations will cease until the key is available again.

Snowflake allows temporary key availability outages of up to ten minutes that may be caused by common issues, such as network communication failures. After ten minutes, Snowflake considers the key not available and will cease data operations. When the key is provided, operations will resume.

Tri-Secret Secure

The Tri-Secret Secure feature is available in Snowflake Business Critical Edition and higher. The purpose of Tri-Secret Secure is to protect data in Snowflake by combining a Snowflake-maintained key with a customer-managed key to create a composite master key. This key is placed at the top level of the key hierarchy above all lower levels of keys and, as such, is not used to encrypt data.

Tri-Secret Secure ensures that data can only be decrypted if the customer-managed key is available to Snowflake. To enable Snowflake Tri-Secret Secure in your account, you must contact Snowflake support.

Cloning

Snowflake's zero-copy cloning feature allows you to create a copy of an object without actually creating a physical copy of the object because both objects initially share the same underlying storage.

When a table is cloned, the cloned table shares all micro-partitions with the original table. This means that the cloned table will not incur additional storage costs. After the initial cloning, the original and the

cloned table follow different data lifecycles. Data can be inserted, deleted, or updated in the cloned table or the original table independently. Whenever data is changed in the cloned table, new micro-partitions that are owned only by the clone are created. These new micro-partitions will incur additional cost because they are not shared with the original table.

Note Clones can be cloned, with no limitation on the number or iterations of clones that can be created.

Cloning is primarily used for creating zero-copy clones of databases, schemas, and tables. It can also be used to create clones of other schema objects such as external stages, file formats, and sequences that are contained within the cloned objects.

When cloning a database or a schema, granted privileges are inherited as follows:

- The clone inherits all granted privileges on the clones of all child objects contained in the source object.
- The clone of the database or schema itself does not inherit the privileges granted on the source database or schema.

Cloning commands don't copy grants on the source object to the cloned object. However, `CREATE` commands that support the `COPY GRANTS` clause, such as `CREATE TABLE` or `CREATE VIEW`, allow you to copy grants to object clones.

For example, to create a copy of the `NATION` table by cloning it as `NATION_CLONED`, including grants on the original table, use the following command:

```
create table NATION_CLONED clone NATION COPY GRANTS;
```

When cloning a parent container, here are some considerations with respect to cloning child object types within the container:

- **Object parameters.** Cloned objects inherit object parameters that were set on the source object.
- **Default sequences.** When cloning a table with a column that uses a sequence for generating unique values, if the database or schema containing both the table and sequence is cloned, the cloned table will reference the cloned sequence. Otherwise, it will reference the source sequence.
- **Foreign key constraints.** When cloning a table with a foreign key constraint that references a primary key in another table, if the database or schema containing both tables is cloned, the cloned table with the foreign key will reference the primary key in the cloned table. Otherwise, the cloned table will reference the primary key in the original table.
- **Clustering keys.** When a table with a clustering key is cloned, the new table is created with the same clustering key. Automatic clustering is suspended after cloning and you must manually resume it.
- **External stages.** External named stages can be cloned.
- **Internal stages.** Internal named stages cannot be cloned.
- **Pipes.** Pipes that reference an internal stage cannot be cloned. Pipes that reference an external stage are cloned, but must be manually resumed after cloning.

- **Streams.** When a stream is cloned, it will begin accumulating changed data at the time of cloning. Any unconsumed records that were tracked in the stream before cloning are disregarded.
- **Tasks.** When tasks are cloned, they are suspended immediately after cloning and must be manually resumed.

Snowflake recommends that no DML statements be executed on the source objects while cloning is in progress. Because Snowflake attempts to clone table data as it existed when the cloning operation began, it relies on time travel data to reconstruct the data at that point in time.

If data in the source table has been changed due to DML operations while cloning is in progress and the `DATA_RETENTION_TIME_IN_DAYS` parameter has been set to `0` for the table, the cloning operation will fail because there will be no time travel data from which Snowflake can reconstruct the data at the time the cloning operation began.

If you can't guarantee that there will be no DML operations during cloning, then set the `DATA_RETENTION_TIME_IN_DAYS` parameter to `1` for all tables that will be cloned so that the cloning process has time travel data available.

Database Replication

The Snowflake replication feature can be used to replicate databases between Snowflake accounts in the same organization. Databases can be replicated to different regions and even to different cloud providers.

One or more databases that will be replicated are referred to as primary databases. These can be either permanent or transient. Each primary database can be replicated to one or more accounts. In the account where the primary database is replicated, a copy of this database

is created. The copy is referred to as the secondary database. Data can only be changed in the primary database while the secondary database is read-only.

Database replication requires that data is physically copied between accounts. This means that data may become outdated after it has been originally copied. To keep replicated data in secondary databases in sync with primary databases, periodic replication tasks are usually set up.

To secure data in transit while it is replicated to another account, Snowflake encrypts the data using a random encryption key.

Replication Restrictions

Not all database objects can be replicated. The following database objects can be replicated to a secondary database:

- Permanent and transient tables
- Clustering on tables
- Table constraints
- Sequences
- Views
- Materialized views
- Secure views
- File formats
- Stored procedures
- UDFs (SQL, JavaScript, Python)
- Masking policies, including tag-based masking policies
- Tags

When a database is replicated to a secondary database, the privileges granted on database objects are not replicated.

With respect to replicating parameters, the following applies:

- Account parameters are not replicated.
- Database parameters are not replicated.
- Table and schema parameters that were explicitly set on objects are replicated.

There are a few additional limitations with respect to replication:

- When the primary database is Business Critical or higher, replication to lower editions is not supported.
- Databases that contain external tables cannot be replicated.
- Databases that are created from shares cannot be replicated.

Billing for Database Replication

Database replication incurs cost for data transfer between accounts and compute resources that were utilized for the transfer. Data transfer cost is incurred because cloud providers charge for data transfer between regions. Compute resources cost is incurred because replication uses Snowflake-provided compute resources. The costs are billed to the target account where the secondary database is located.

In addition to data transfer cost and compute resources cost, the target account also incurs storage cost for the data in the secondary database.

When a replication operation fails for any reason, such as a network outage, the next refresh can reuse any data that was already copied in the previous refresh, but only within 14 days.

Account administrators can use Snowsight to view the amount of data that was replicated in a chosen time period. To view the data transfer amounts for your account, log in to the account where the secondary database is located. Ensure that `ACCOUNTADMIN` is selected as your current role. Then click the Admin option in the left menu and choose Usage. In the window that appears, click the All Usage Types drop-down list and choose Data Transfer. Select your date range depending on the time period for which you want to view results. You will see a diagram similar to Figure 9-3.

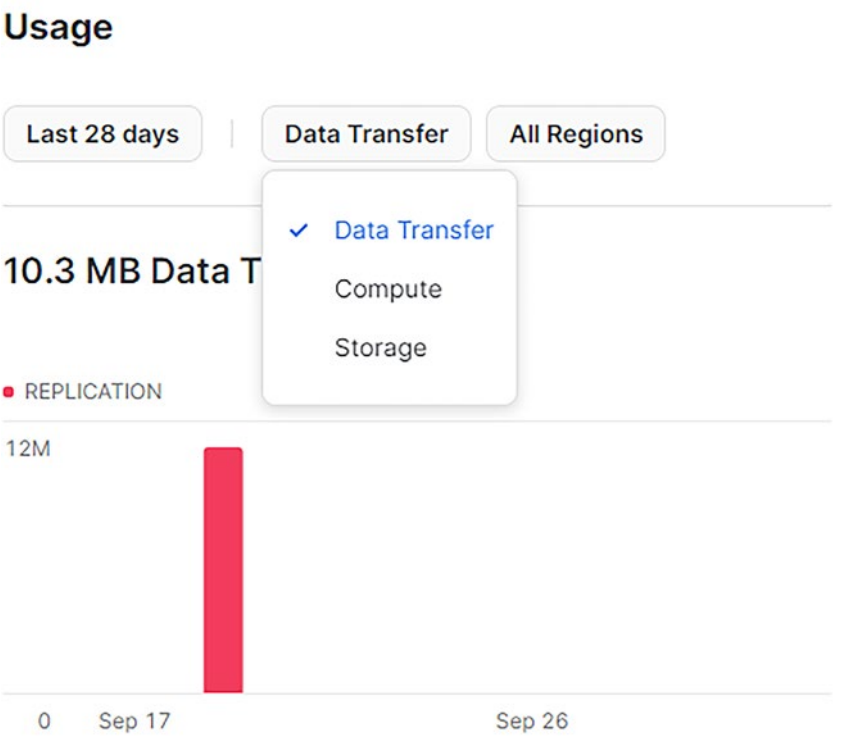

Figure 9-3. *Data transfer usage*

You can also view data transfer usage and associated credit charges by executing SQL commands.

You can query the REPLICATION_USAGE_HISTORY table function in Snowflake Information Schema. This function returns replication usage activity within the last 14 days. For example, to view data transfer usage within the last ten days, execute the following command:

```
select *
  from table(information_schema.REPLICATION_USAGE_HISTORY(
    date_range_start => dateadd(d, -10, current_date),
    date_range_end => current_date));
```

Alternatively, you can query the REPLICATION_USAGE_HISTORY view in Account Usage. This view returns replication usage activity within the last 365 days. For example:

```
select * from snowflake.account_usage.REPLICATION_USAGE_
HISTORY;
```

Both of these queries return similar results, where you will see the start time and end time of each data transfer, along with the database name, bytes transferred, and credits used.

Working with Database Replication

Setting up database replication is performed by Snowflake administrators and as such you are not required to know all the details for the SnowPro Core exam. But since you are expected to understand the concepts of database replication, this section walks through the steps required to set up database replication.

Database replication works between accounts, which means that you need access to at least two accounts in an organization. This section assumes that accounts have been created by the Snowflake administrator.

Before setting up database replication, the organization administrator or a user with the ORGADMIN role must enable replication for the source

and target accounts. This is done by executing the `SYSTEM$GLOBAL_ACCOUNT_SET_PARAMETER` function to set the `ENABLE_ACCOUNT_DATABASE_REPLICATION` parameter to `TRUE` for both the primary and secondary accounts that will participate in replication.

Once replication has been enabled, a user with the `ACCOUNTADMIN` role can click the Data option in the left menu and choose Databases. A window with a list of databases will appear. The user must click the three dots next to the database to be replicated—in this case, the `SALES` database—and choose Enable Replication in the drop-down list, as shown in Figure 9-4.

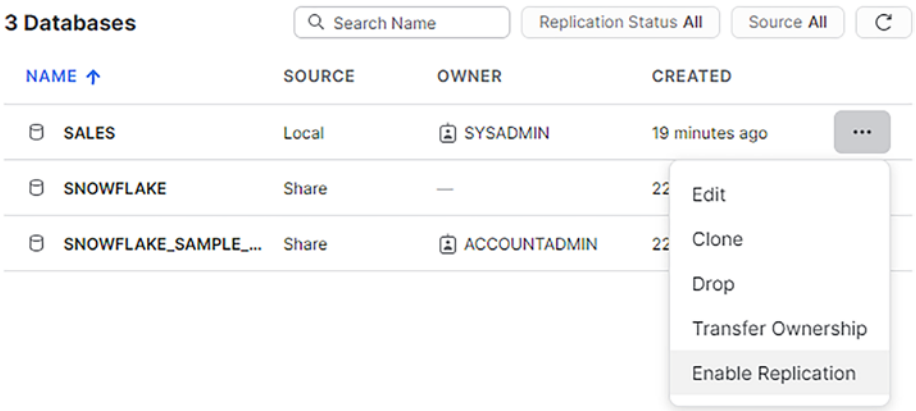

Figure 9-4. *Enable replication for a database*

A window will appear, as shown in Figure 9-5, where the user can choose the secondary account where the database will be replicated and click the Enable Replication button.

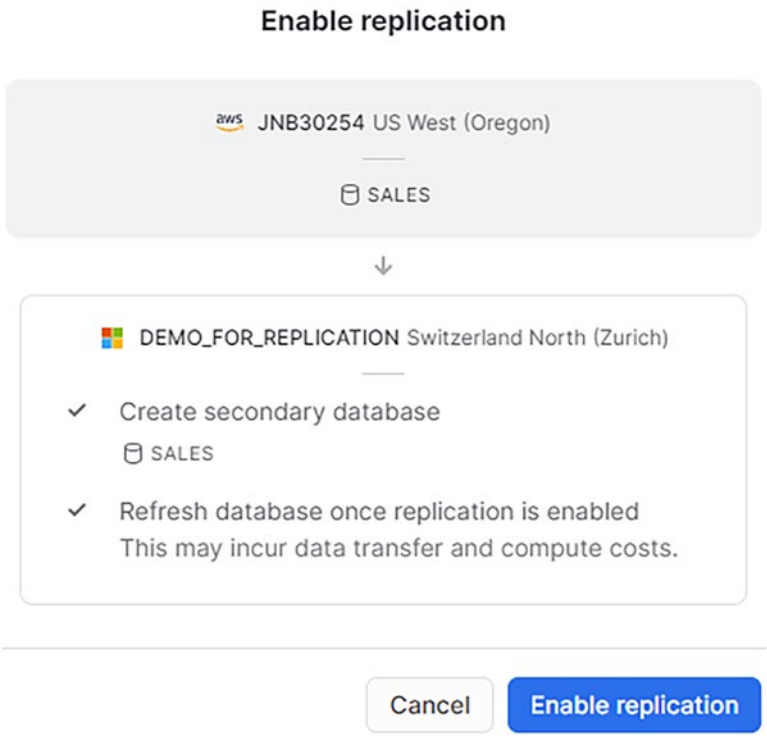

Figure 9-5. *Secondary account for replication*

To verify that the replication has succeeded and to view the replicated database in the secondary account, you must log in to the secondary account. Once you log in, click the Data option in the left menu and choose Databases. A window with a list of databases will appear, as shown in Figure 9-6.

NAME	REPLICATION	SOURCE	OWNER
SALES	Secondary	Replication	ACCOUNTADMIN
SNOWFLAKE	—	Share	—

Figure 9-6. *List of databases in the secondary account*

As you can see in Figure 9-6, the SALES database has appeared on the list of databases in the secondary account. The database source is Replication and the replication type is Secondary.

You can click the SALES database to view the details. On the Replication tab, you will see something like Figure 9-7.

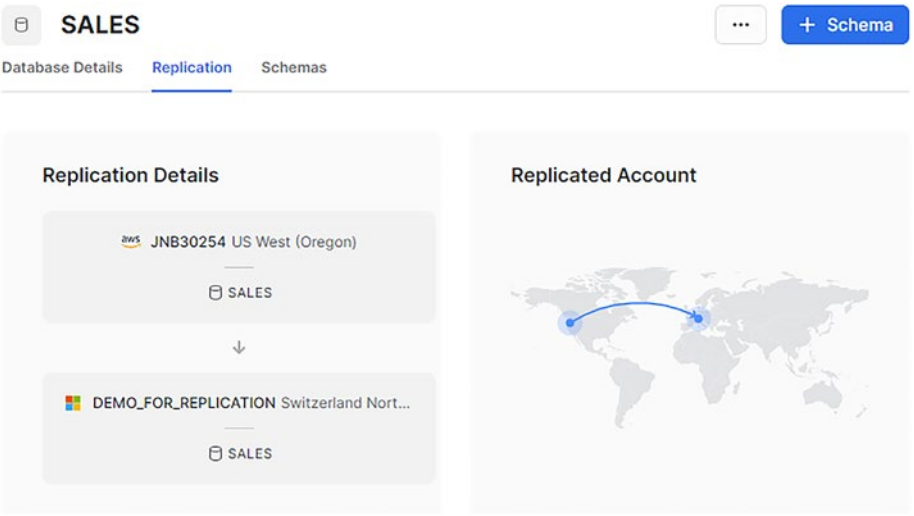

Figure 9-7. *Replication tab of the replicated database*

As you can see in Figure 9-7, the SALES database is replicated from an AWS account located in the US West region to an Azure account located in the Switzerland North region.

Remember that when setting up database replication, the replication is only done once. In order to enable periodic updates of the replicated database, you must set up a Snowflake task that will refresh the replicated database on a regular basis.

Snowflake currently supports replication only at the database level, but not at schema or table level. When you have a database with a large number of objects, but you want to replicate only a selected portion of the database, you can set up the following scenario:

1. Clone the desired objects that will be replicated into a new database in the same account.
2. Create streams on the source objects to track changes and tasks to keep the cloned data in sync.
3. Replicate the cloned database to another account in the organization in a different region.
4. Create a task that will refresh data in the replicated database and schedule it on a regular basis.

Summary

This chapter reviewed Snowflake features related to continuous data protection. It discussed time travel, which allows you to access historical data as it existed before it was changed or deleted. It reviewed the failsafe feature, which ensures that data is protected against a system failure or other unexpected events. The chapter looked at data encryption, which prevents third parties from reading data while in transit to and from Snowflake. This chapter also provided an overview of zero-copy cloning and described and showed an example of database replication.

The next chapter discusses data sharing in the form of Snowflake shares, as well as secure views and secure user-defined functions.

CHAPTER 10

Data Sharing

Secure data sharing is a Snowflake feature that enables users to share objects in a Snowflake account with other Snowflake accounts. Data that is shared is not physically copied between the accounts, which means that it does not consume additional storage and it can be set up quickly.

This chapter describes some of the products based on secure data sharing that are available in Snowflake. It first provides an overview of secure data sharing, followed by discussions of direct shares, Snowflake Marketplace, and data exchange. It also looks at secure views and secure user defined functions (UDFs), which are often used together with secure data sharing.

Secure Data Sharing

Data sharing in Snowflake is enabled through shares. A *share* is created by the account that wants to share database objects, also referred to as a data provider. An account that uses the share is referred to as a data consumer. Secure data sharing does not physically copy any data between accounts. The data remains stored in the provider account and can only be queried, but not modified by consumers.

Since data is not physically copied to the consumer account, it does not consume any additional storage after it has been shared. The consumer pays only for the compute resources used to query the shared

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_10

data. All operations related to data sharing can be performed within Snowflake's services layer and metadata.

The following database objects can be shared:

- Tables
- External tables
- Secure views
- Secure materialized views
- Secure UDFs

To set up secure data sharing, the provider creates a share based on a database in the account. Selected objects can then be added to the share by granting access. The objects that are added to a share can be sourced from multiple databases within the Snowflake account that is creating the share. After the share has been created, one or more accounts with which the objects will be shared are added to the share.

Once a share has been set up by the provider account, and at least one consumer account has been added, the consumer can then create a read-only database from the share. From the point of view of the consumer, the share will appear just like another database in their account, with query capabilities and access that is controlled using the usual Snowflake role-based access control.

Snowflake Share Object

Shares are named Snowflake objects that include:

- Privileges that grant access to databases and schemas that will be shared.
- Privileges that grant access to selected objects that will be shared.
- Consumer accounts with which the database and its objects are shared.

Shares are created and managed by the provider account. After a share has been set up by the provider and the consumer, all objects in the share will be available to users in the consumer account. The provider account can add new objects to the share and these objects will become immediately available to users in the consumer account.

The provider can revoke access to a share for any of the consumers that have been added to the share object.

There is no limit to the number of shares that a provider can create or the number of accounts that a provider can add to a share. There is also no limit to the number of shares that a consumer account can configure from shares that have been set up by data providers. The only limit in a consumer account is that only one database can be set up for each share.

Reader Accounts

Since data sharing is supported between Snowflake accounts, both the provider and consumer must have access to a Snowflake account. Data sharing is still possible even with a consumer who does not have a Snowflake account.

Data sharing with consumers who do not have Snowflake accounts is possible by creating *reader accounts*. After a provider has created a share, they can also create a reader account that then belongs to the provider. The reader account can create a database from the share, similar to a consumer account. Only shares that have been shared by the provider that created the reader account can be configured by the reader account.

Direct Share

Sharing data using a direct share is supported only between Snowflake accounts in the same region. Data can't be shared between accounts in different regions; in this case, database replication is used, as explained in Chapter 9.

To create a share, you must use the ACCOUNTADMIN role or a role that has been granted the CREATE SHARES global privilege. Then you can either execute the appropriate commands, such as CREATE SHARE, ALTER SHARE, and GRANT, or use the Snowsight user interface.

Before you can walk through the examples for sharing a database with another account, you must have a database available to work with and at least one additional account created with which you will share the database.

Create a database named SUPPLIERS, a schema named SUPPLIERS_DATA, and a couple of tables, using these commands:

```
create database SUPPLIERS;
create schema SUPPLIERS_DATA;
create table SUPPLIER as
  select * from SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.SUPPLIER;
create table REGION as
  select * from SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.REGION;
```

The organization administrator or a user with the ORGADMIN role must create an additional account with which you will share the database. But you can skip this for now, because you can create a direct share in the current account without specifying the account with which it will be shared. You can always add that account later.

Creating a Direct Share

Let's review how to create a direct share using the Snowsight user interface. Using the ACCOUNTADMIN role, select Data in the left menu and choose the Private Sharing option, as shown in Figure 10-1.

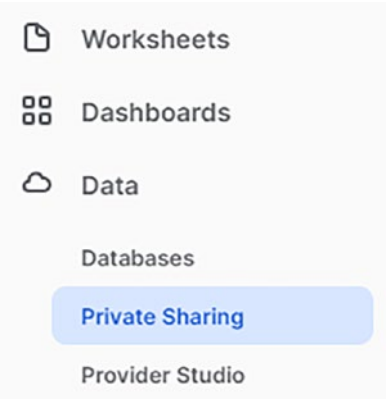

Figure 10-1. *Private Sharing option in Snowsight*

A window with three tabs will appear and the first tab, named Shared With Me, will be selected. Here, you will most likely see databases that have been shared with the current account: `ACCOUNT_USAGE` and `SAMPLE_DATA`. But you are not interested in these direct shares at this time. Instead, click the Shared By My Account tab. To create a new share, click the Share Data button at the top right. A new window will appear, as shown in Figure 10-2.

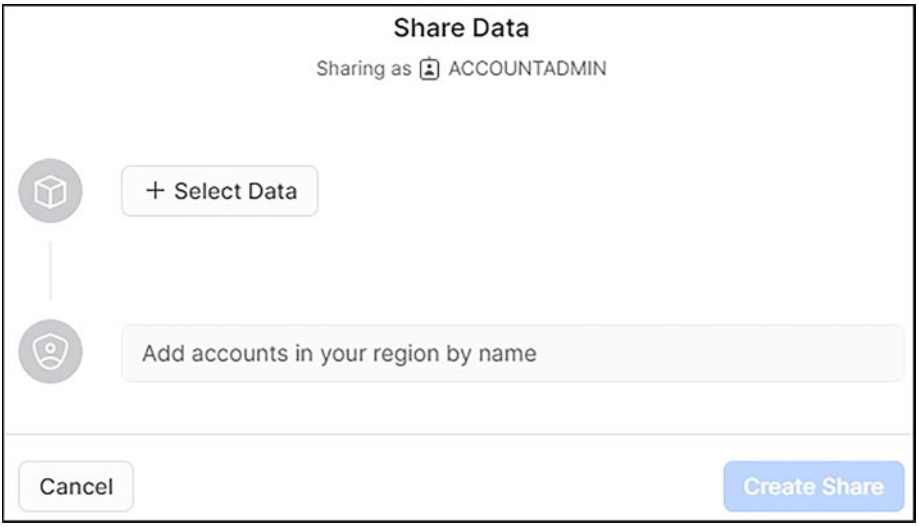

Figure 10-2. *Share Data window in Snowsight*

To add a database and selected objects to be shared, click the + Select Data option. Use the navigation in the pop-up window to select the `SUPPLIERS` database, the `SUPPLIERS_DATA` schema, and the two tables in this schema, as shown in Figure 10-3. Then click the Done button at the bottom right.

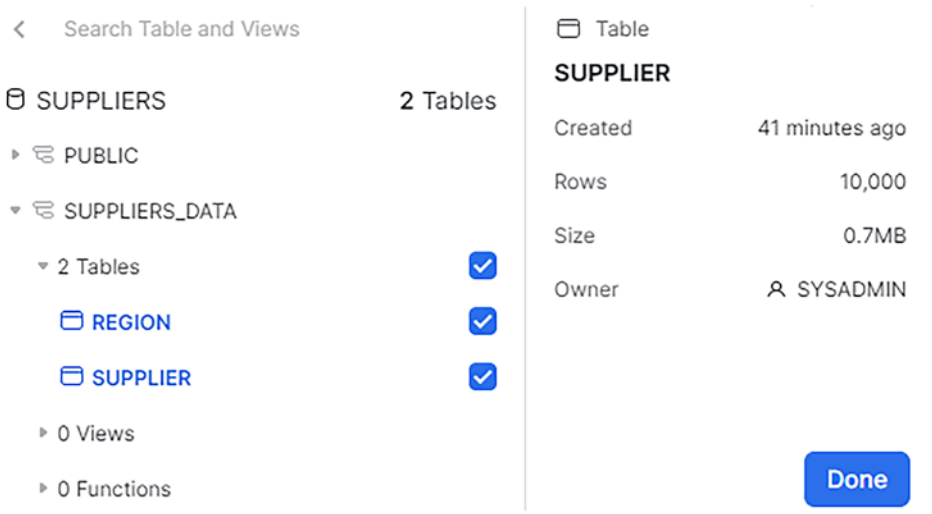

Figure 10-3. *Select Data window in Snowsight*

You will be returned to the Share Data window, as shown in Figure 10-4, where you can name your share in the Secure Share Identifier text box. This example calls the share `SUPPLIERS_SHARE`. You can add an optional description in the Description text box. If you already know the accounts in your region with which the direct share will be shared, you can enter them now. Otherwise, skip them for now, because you will be able to add them later. Click the Create Share button when you are done.

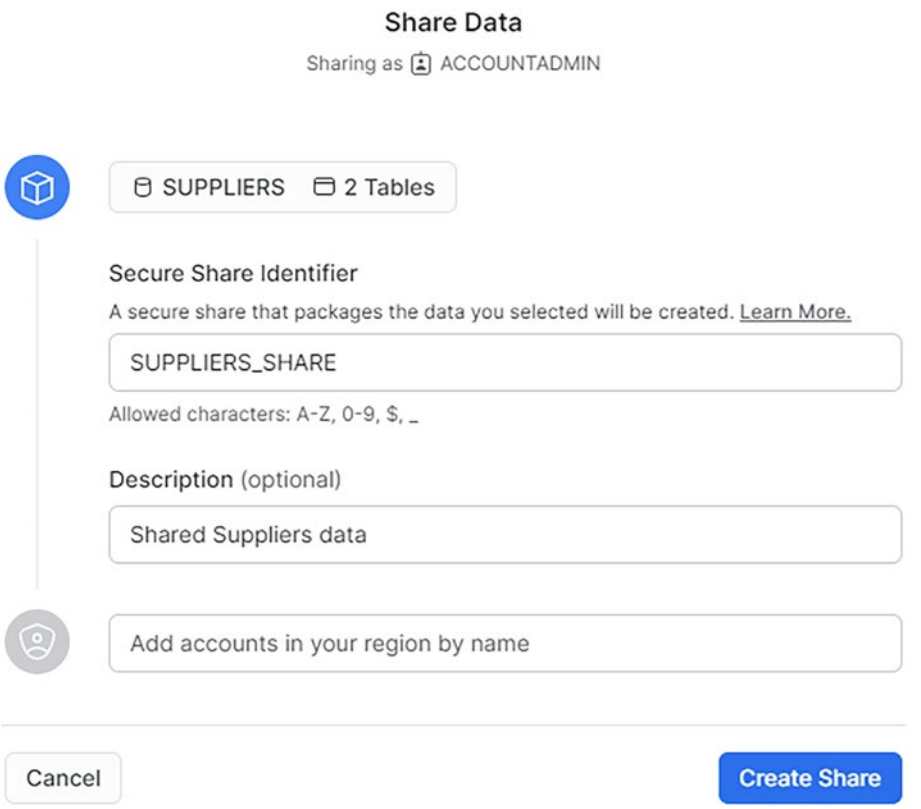

Figure 10-4. *Share Data window in Snowsight after selecting database objects*

Instead of using the Snowsight user interface, you can complete the previous steps by executing the following commands:

```
create share SUPPLIERS_SHARE;
grant usage on database SUPPLIERS
  to share SUPPLIERS_SHARE;
grant usage on schema SUPPLIERS.SUPPLIERS_DATA
  to share SUPPLIERS_SHARE;
grant select on table SUPPLIERS.SUPPLIERS_DATA.SUPPLIER
  to share SUPPLIERS_SHARE;
```

```
grant select on table SUPPLIERS.SUPPLIERS_DATA.REGION
  to share SUPPLIERS_SHARE;
```

As you can see from these commands, objects are added to the share using GRANT commands to grant the following privileges:

- USAGE privilege on the database
- USAGE privilege on each schema
- SELECT privilege on each shared object, including tables, external tables, secure views, secure materialized views, and secure UDFs

Finally, you use the ALTER SHARE command to add one or more consumer accounts to the share. These are the accounts with which you will share the objects that have been added to the share. For example, to add account BH09635 to the SUPPLIERS_SHARE direct share, execute this command:

```
alter share SUPPLIERS_SHARE add accounts = BH09635;
```

The share is now ready to be consumed by the specified accounts.

If you prefer executing SQL statements instead of using the Snowsight user interface, here are some useful commands for working with shares.

To view all the shares in your account, use the SHOW SHARES command, like this:

```
show shares;
```

The output of this command will list all inbound and outbound shares along with their properties.

To view the details about a single share, use the DESC command like this:

```
desc share SUPPLIERS_SHARE;
```

The output of this command will list the databases, schemas, and tables that are included in the specified share.

To view the grants that have been assigned to the share, use the SHOW GRANTS command, like this:

```
show grants to share SUPPLIERS_SHARE;
```

The output of this command will list the grants that have been assigned to the specified share, such as USAGE on the database and schema and SELECT on the tables that are included in the share.

Consuming a Direct Share

After you create a direct share and add one or more consumer accounts to it, log in to one of the consumer accounts. You can use the Snowsight user interface to view databases that have been shared with the account.

Using the ACCOUNTADMIN role, select Data in the left menu and choose the Private Sharing option, as shown in Figure 10-1.

On the Shared With Me tab, you will see the SUPPLIERS_SHARE direct share that has been created with the provider account, as shown in Figure 10-5. It should be in the Ready to Get status. Click the box with this share.

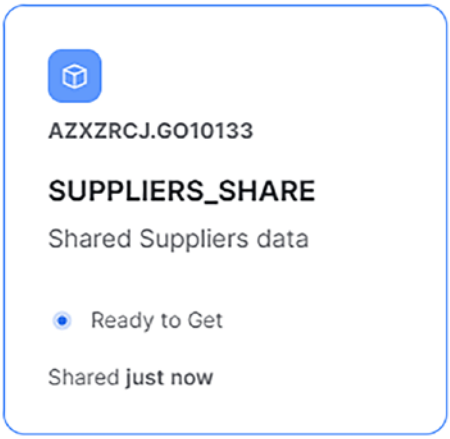

Figure 10-5. *Direct Share ready to get*

Once you click the box, the Get Data window will appear, as shown in Figure 10-6.

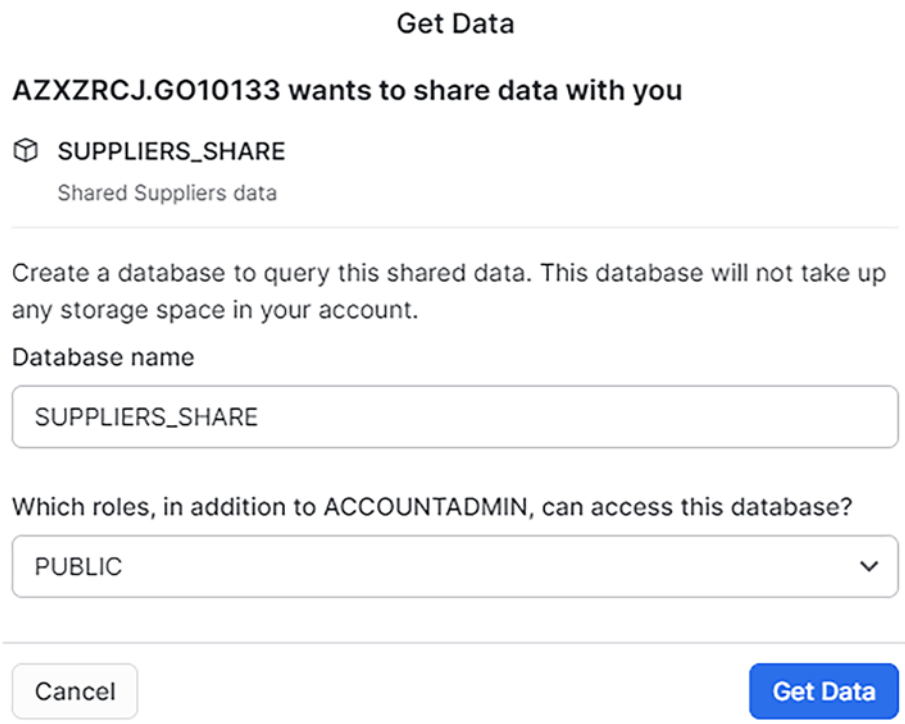

Figure 10-6. *Get Data window*

In this window, you provide the database name as it will appear in the consumer account. You also have the option to choose any roles, other than ACCOUNTADMIN, to access this database. When you are done entering the information, click the Get Data button. A database object will be created in the consumer account from the direct share. This database can be queried just like any other database in your account.

Instead of using the Snowsight user interface, you can complete the previous steps by executing the following commands in the consumer account:

```
create database SUPPLIERS_SHARE
  from share AZXZRCJ.GO10133.SUPPLIERS_SHARE;
grant imported privileges on database SUPPLIERS_SHARE
  to role PUBLIC;
```

Creating a Reader Account

If you want to share the SUPPLIERS database with a user who is not a Snowflake customer, you can create a reader account for them. You can do this in the Snowsight user interface.

Using the ACCOUNTADMIN role, select Data in the left menu and choose the Private Sharing option, as shown in Figure 10-1. Then select the Reader Accounts tab. Click the + New button at the top right. A window will appear, as shown in Figure 10-7.

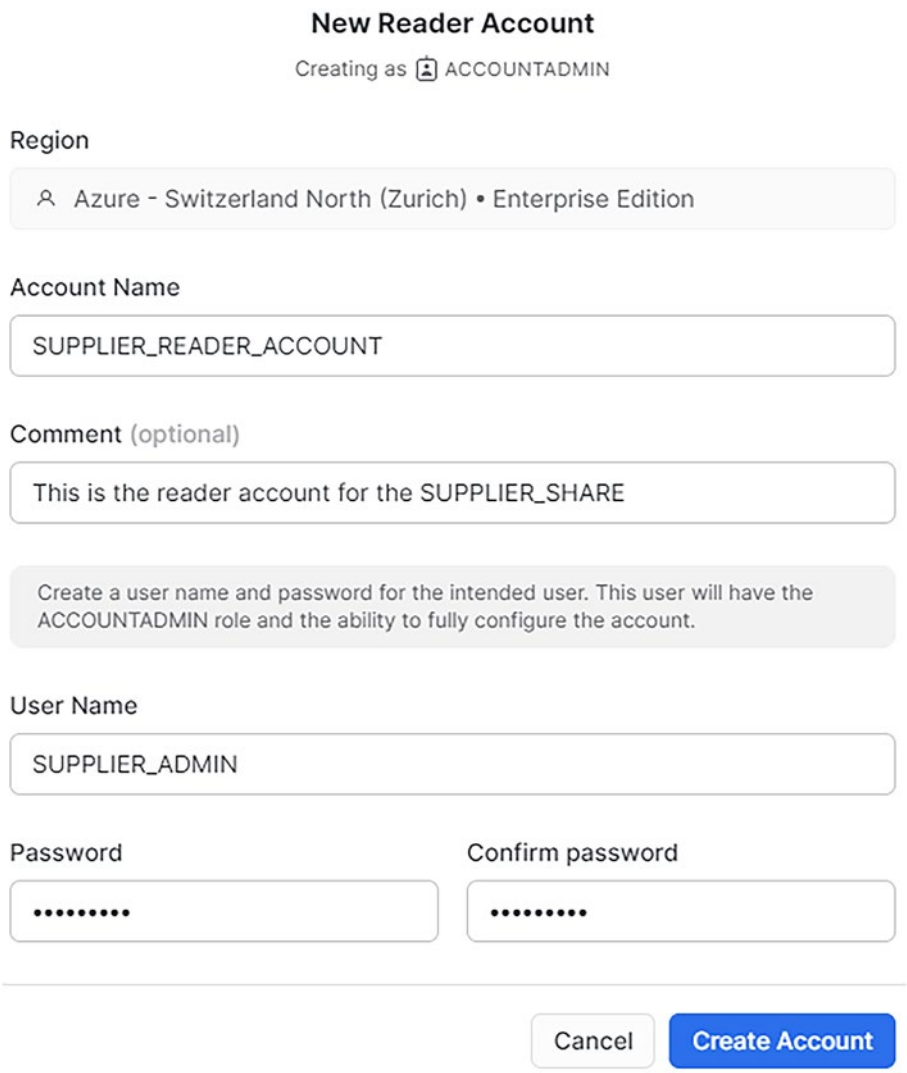

Figure 10-7. *New Reader Account window*

In this window, you provide the reader account name in the Account Name text box. You can add an optional comment for the account. Then provide the username and the initial password for the reader account. When you are done entering the information, click the Create Account button.

Once you have created the reader account, go back to the SUPPLIERS_SHARE direct share that you created earlier and add SUPPLIER_READER_ACCOUNT to the share.

Now you can log in to the new reader account using the credentials that you specified when creating the reader account.

As a reader, you will create a database from the SUPPLIERS_SHARE direct share just like you would for a consumer account, as explained in the previous section.

Configuring a Reader Account

A reader account is created with only a single user that has been defined when creating the account. This user is the administrator for the reader account and should be used for account configuration and monitoring only. It shouldn't be shared with any of the users who will be querying data. To allow additional users to access the reader account, the account administrator must create users, custom roles if desired, and virtual warehouses for executing queries.

Each reader account is created with the standard, system-defined roles, including SYSADMIN, SECURITYADMIN, and PUBLIC. You can create additional custom roles to enable control over the activities that users in the reader account can perform and objects that they can access. You can use roles to:

- Grant access control privileges to users who will query data.
- Grant privileges on virtual warehouses that will be used for querying.
- Grant additional privileges to any users who will perform tasks that require such privileges.

Then you will create users who can log in to the reader account and access the data. Grant appropriate roles to each of the users depending on what they will be allowed to do in the reader account.

To enable querying the objects in the shared database, you must create a virtual warehouse and grant the `USAGE` privilege on the warehouse to the users.

Since the reader account is not a Snowflake customer, the provider account will be charged for any credits consumed by the reader account. As such, it is strongly recommended to create one or more resource monitors and decide whether they control all warehouses in the account or individual warehouses.

To enable querying data shared with the reader account, you must also grant `IMPORTED PRIVILEGES` on each database that was created from the share to the appropriate roles, for example:

```
grant IMPORTED PRIVILEGES on database SUPPLIERS_SHARE
to <role>;
```

Snowflake Marketplace

The Snowflake Marketplace allows third-party data providers to share data so that it is accessible directly from Snowflake accounts, where users can query the data just like any other data in their account. Snowflake Marketplace is available to all Snowflake accounts hosted on non-VPS regions.

Using Snowflake Marketplace, data consumers can:

- Discover and evaluate third-party data that could be potentially useful for their business cases.
- Access data that is provided by vendors and readily available in Snowflake for querying using their preferred tools.

With Snowflake Marketplace, data is always up to date because it is maintained by the data providers without any additional effort on the part of the consumer.

Types of Data Listings

Data in Snowflake Marketplace is available through listings. There are several types of data listings, including:

- **Marketplace listing.** Data providers can list their data offerings that can be shared with many consumers.
- **Free listing.** Consumers have immediate access to the data.
- **Personalized listing.** Consumers can request data from providers, for example because it is premium data that must be paid for or data that is customized for the consumer.

Each of these listings contains information about shared data, sample queries, and details about the data provider.

To access data from a free listing, click the Get button in Snowsight (as explained later in this chapter), provide a name for the database, access roles, and agree to the provider's terms of use and Snowflake's consumer terms. The database is immediately created in the Snowflake account and becomes available just like any other database.

To access data from a personalized listing, submit a request by clicking the Request button in Snowsight (as explained later in this chapter) and provide your contact information. Once a request is submitted, the data provider is notified. The data provider will respond with their commercial terms. Once those terms are accepted, the personalized dataset is created and shared with the consumer who requested it.

Get or Request Listings

You can only access the Snowflake Marketplace if it is activated in your region. All users can browse listings in the Snowflake Marketplace, but only users with the `ACCOUNTADMIN` role or the `IMPORT SHARE` privilege can get or request data.

If you do not have the necessary privileges, you can take one of the following steps:

- Ask your Snowflake account administrator to grant you `IMPORT SHARE` privilege.
- Ask your Snowflake account administrator to get the data and grant you `IMPORTED PRIVILEGES` on the database created from the share.

To access the Snowflake Marketplace from Snowsight, click the Marketplace option in the left menu, as shown in Figure 10-8.

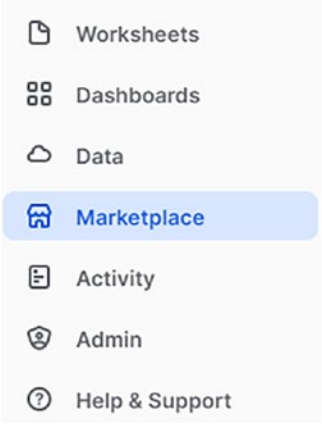

Figure 10-8. *Marketplace option in Snowsight*

The Snowflake Data Marketplace window will appear, where you can browse the listings.

Let's explore the free listings and create a database from one of them. Click the Browse Data Products tab. Then, in the first drop-down list, under Availability, choose the Ready to Query option. You will see a window similar to the one shown in Figure 10-9.

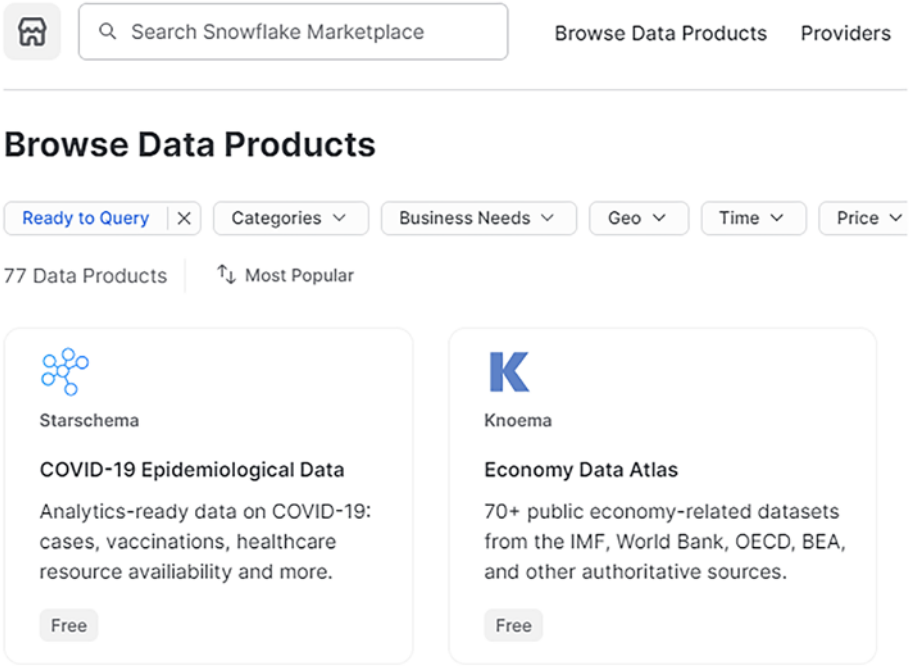

Figure 10-9. *Ready to Query listings*

You can scroll through the available listings and read the descriptions. You can also click each listing to find more details along with sample queries. Once you choose a listing that you want to use, click the Get button on the listing's details page.

For this example, we chose the Global Weather & Climate Data for BI from Weather Source, LLC. After clicking the Get button, a window similar to what is shown in Figure 10-10 will appear.

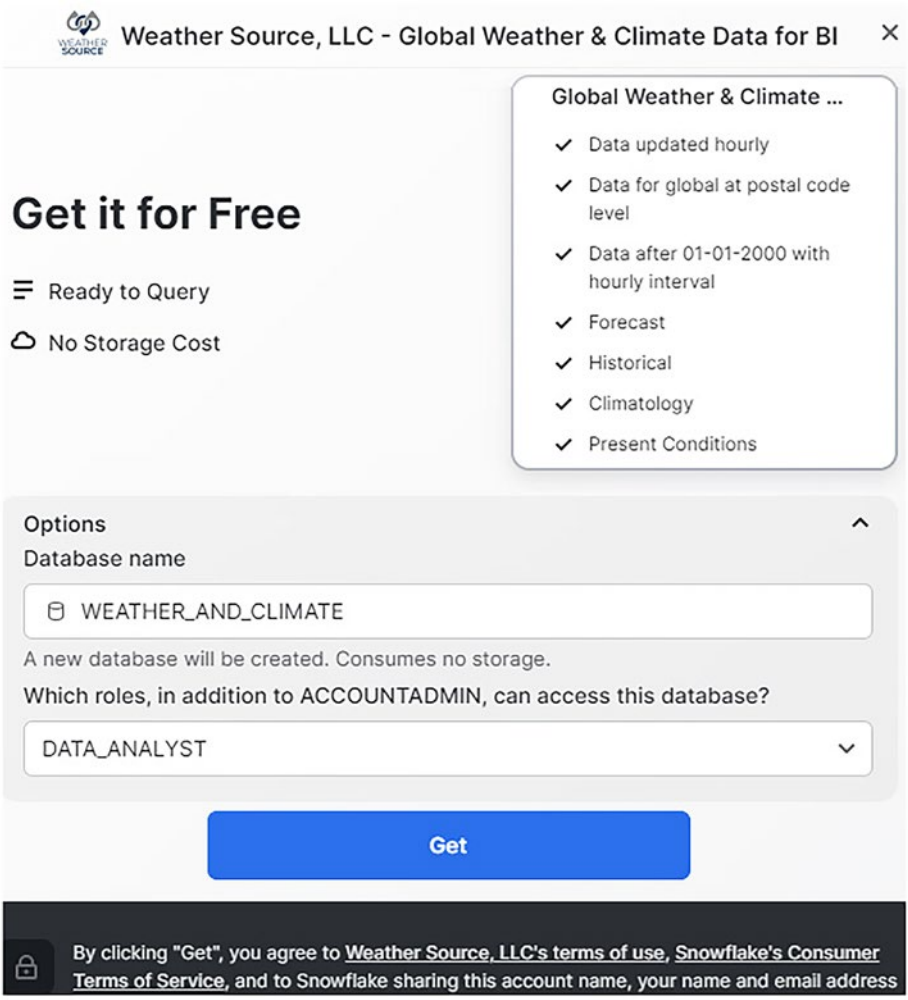

Figure 10-10. *Get the Global Weather & Climate Data for BI free listing*

In this window, you can provide the database name that will be created in your account as well as any roles that will be allowed to access the share. Then, if you agree to the terms and conditions at the bottom of the window, click the Get button again. A message informing you that the data is ready to query will appear.

Now go back to the main menu on the left, click the Data option, and expand the `Databases` folder. You should see the newly created database named `WEATHER_AND_CLIMATE`.

If you want access to a personalized listing, you can browse these similar to the free listings. However, to access these listings, instead of a Get button, you will see a Request button. After clicking it, you provide your contact information and wait for the provider of the listing to get in touch with you to agree on commercial terms and conditions.

Become a Provider

If you have data that you want to share by making it available on the Snowflake Marketplace, you can request to join as a provider. First click the Marketplace option on the left navigation pane in Snowsight, then scroll to the bottom of the page and click the Become a Provider button. You have to provide additional information; once you're approved, you will be able to create listings using the Provider Studio option in Snowsight.

You can also submit a request for new data to be added to the Snowflake Marketplace by clicking the Let Us Know button at the bottom of the Marketplace page and filling out the requested information in the form that appears.

The Snowflake Marketplace can also be accessed directly from a web browser at this URL:

`www.snowflake.com/en/data-cloud/marketplace/`

From there, you can choose the option to browse listings or any other options that provide additional information about the Snowflake Marketplace.

Data Exchange

Snowflake data exchange is a data hub that a Snowflake customer can use to share data between a group of members that they invite. Similar to Snowflake Marketplace, this feature enables providers to publish data that can be discovered by consumers.

At this time, a data exchange can only be enabled by Snowflake support. A Snowflake customer must submit a support request and wait until the data exchange is provisioned and configured. Then they can invite members to the exchange and specify whether they can consume data, provide data, or both.

A data exchange can be used to share data between business units in a company, share data with external partners, or even share data with other data exchanges.

The Snowflake account that hosts the data exchange, also referred to as the *data exchange administrator,* configures the data exchange and manages members. Members of the exchange are Snowflake accounts such as providers and consumers that are added by the data exchange administrator. The administrator can also grant and revoke access to members, grant and revoke access to shared data, audit data usage, and define any access controls to the shared data.

As in Snowflake Marketplace, providers in a data exchange can create listings, define the type of listing, publish the listing, and grant access in case the listing is not free. Consumers in a data exchange can discover data by browsing the available listings and access data that they have selected from the listings.

Secure Views

Snowflake provides secure views as a means to hide sensitive information that could be exposed through standard views. For example, Snowflake

internal optimizations may require access to the underlying data in base tables for the view, which could expose data that should be hidden from users.

For this reason, secure views don't use the same optimizations as standard views so that data is not accidentally exposed. A consequence of using different optimizations in secure views is that secure views typically execute more slowly than standard views. Therefore, the use of secure views is not recommended if the purpose of the view is to simplify queries.

Views should be defined as secure when they are specifically required for data privacy to limit access to sensitive data that should not be exposed to all users of the underlying tables.

The query expression used to create a standard view is visible to users in various commands and interfaces. If you don't want the users to see the underlying tables or the query details for a view, you should create a secure view, whose view definition is visible only to authorized users.

To understand how data can be exposed indirectly, we review the concept of pushdown. Let's illustrate this concept using the following query:

```
select *
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.LINEITEM
where l_shipmode = 'RAIL';
```

This query might be executed using these steps:

1. Start with the `FROM` clause and read all rows from the `LINEITEM` table into memory.
2. Then apply the `WHERE` clause to filter out any rows that don't match `'RAIL'`.
3. Finally, use the list of columns in the `SELECT` clause to select the columns from the rows still remaining in memory.

With this approach, data is loaded first and filtered later, which is not very efficient. It's better to filter as early as possible. Early filtering is also referred to as pushing the filter down deeper into the query plan, or *pushdown*.

In the query in this example, it would probably be more efficient to load fewer records into memory by first filtering on the `WHERE` clause and loading only the filtered rows into memory. This wouldn't save filtering time because the filter would still have to be applied, but it could save memory usage. Because the number of rows has been reduced first, subsequent processing time would also be reduced, because there is less data to process.

Pushdown applies to many different types of queries. The filter that is most selective is usually pushed deepest to be executed earliest.

If the Snowflake query optimizer applies the filters in a way that allows a more selective filter to execute before any filters used to secure data, details about the underlying data could be exposed. Creating a view as secure tells the optimizer to not push down such filters.

Creating Secure Views

Secure views are defined using the `SECURE` keyword in the `CREATE VIEW` or `CREATE MATERIALIZED VIEW` command. For example, to create a secure view named `LINEITEM_RAIL` that selects data from the Snowflake sample database table called `LINEITEM`, and filters for rows where the shipping mode equals `'RAIL'`, you would use the following command:

```
create secure view LINEITEM_RAIL as
select *
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.LINEITEM
where l_shipmode = 'RAIL';
```

To verify that the secure view has been created and that it is indeed secure, you can use the `SHOW VIEWS` command, like this:

```
show views like '%RAIL%';
```

The output of this command should include the LINEITEM_RAIL_SECURE view and the IS_SECURE column should contain the value TRUE.

Similarly, when you create secure materialized views, use the SHOW MATERIALIZED VIEWS command and check the IS_SECURE column to determine whether the view is secure.

To convert an existing view to a secure view, set the SECURE keyword in the ALTER VIEW or ALTER MATERIALIZED VIEW command. To convert a secure view to a standard view, unset the SECURE keyword. For example, to convert the secure view LINEITEM_RAIL that you created earlier to a standard view, you use the following command:

```
alter view LINEITEM_RAIL unset SECURE;
```

To set the same view back to a secure view, you use this command:

```
alter view LINEITEM_RAIL set SECURE;
```

Only users who have been granted the role that owns a secure view are allowed to see the definition of the secure view. Unauthorized users will not see the definition of a secure view through any of the following means:

- Executing the SHOW VIEWS or SHOW MATERIALIZED VIEWS commands.
- Using the GET_DDL utility function.
- Selecting from the VIEWS view in Information Schema.
- Selecting from the VIEWS view in Account Usage.

The query profile will also not display any secure view details. This applies even to the owner of the secure view, because other users may be able to view the owner's query profile.

Snowflake doesn't display the amount of data scanned or the total amount of data for queries with secure views. This is to hide the information about data size from users who only have access to a subset of the data from which they could deduce the total amount of data.

Best Practices for Using Secure Views

Although secure views prevent users from viewing data that is filtered by the view, there are still ways that data could be exposed.

One such example is using a sequence for generating primary keys in a table. If these keys are shown to users who do not have access to all of the underlying data, they might still be able to guess the data distribution. For example, using the ORDERS table from the Snowflake sample database, let's create a view that filters the orders for one of the clerks:

```
create secure view ORDERS_CLERK682 as
select *
from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.ORDERS
where o_clerk = 'Clerk#000000682';
```

Supposing that clerk number 682 is allowed to see only their own order data by using the secure view ORDERS_CLERK682. If the primary key column called O_ORDERKEY is generated from a sequence, then this clerk could still deduce the total number of orders created between the creation timestamps of two orders that the user has access to. The user might execute the following query:

```
select o_orderkey, o_orderdate
from ORDERS_CLERK682
order by o_orderdate;
```

Based on the result, the user might guess how many orders were created between two order dates by subtracting the sequence numbers in the order key. If you don't want to expose this information to users, you can use various methods to hide details:

- Do not include the sequence-generated column in the view if it is not needed.
- Replace values generated by a sequence with random identifiers or obfuscate them using hashes.

Even after hiding all sensitive data and query details by creating a secure view, users could still make observations about the amount of underlying data based on how long the query executes.

To prevent users from knowing how long a query from a view executes, it is recommended to materialize data in the form of materialized secure views instead of creating secure views. In the case of the `ORDERS` table, a separate secure materialized view can be created for each clerk, which contains only the orders accessible by that clerk, and a corresponding role would be granted to allow access to the view to each clerk.

Secure User Defined Functions

As with views, some of the internal optimizations for SQL user defined functions (UDFs) require access to the underlying data in tables used by the function. This might allow data that is hidden from the users of the UDF to be exposed.

To ensure that users don't have access to the underlying data, Snowflake provides secure UDFs whose definition and details are visible only to users who are granted the role that owns the UDF.

The Snowflake query optimizer uses different optimizations in secure UDFs as compared to regular UDFs. This might reduce the performance of secure UDFs. Therefore, you should create secure UDFs only when you want to protect the data or function details from being exposed to the users, but not in cases when a regular UDF would suffice.

If you limit access to data by using secure objects such as secure views or secure UDFs, be sure that all objects that access the same data are secure. Otherwise, if one type of object is secure and another isn't, the data would still be exposed.

Creating Secure UDFs

To create a secure UDFs, use the SECURE keyword in the CREATE FUNCTION command. For example, say you need to calculate a special customer discount for a few selected customers, but you don't want the users to know the calculation method that applies this discount. For this purpose, you create a secure function named CALC_CUST_DISCOUNT that calculates a special discount for a few selected customers based on orders from the ORDERS table in the Snowflake sample database. You create the secure function using the following command:

```
create secure function CALC_CUST_DISCOUNT ()
returns table (o_orderkey number, cust_discount number)
language sql
as
$$
  select o_orderkey,
    case when o_custkey in (22768, 918242, 995239, 1070557)
         then o_totalprice * 0.1
    else 0 end as cust_discount
  from SNOWFLAKE_SAMPLE_DATA.TPCH_SF10.ORDERS
$$
;
```

To verify that the secure UDF has been created and that it is indeed secure, you can use the SHOW FUNCTIONS command, like this:

```
show functions like '%DISCOUNT%';
```

The output of this command should include the CALC_CUST_DISCOUNT function and the IS_SECURE column should contain the value 'Y'.

To convert an existing function to a secure function, set the SECURE keyword in the ALTER FUNCTION command. To convert a secure function to a standard function, unset the SECURE keyword. For

example, to convert the secure function called CALC_CUST_DISCOUNT that you created earlier to a standard function, you use the following command:

```
alter function CALC_CUST_DISCOUNT unset SECURE;
```

To set the same function back to a secure function, you use this command:

```
alter function CALC_CUST_DISCOUNT set SECURE;
```

Only users who have been granted the role that owns a secure UDF are allowed to see the definition of the secure function. Unauthorized users will not see the definition of a secure function through any of the following means:

- Executing the SHOW FUNCTIONS command
- Using the GET_DDL utility function
- Selecting from the FUNCTIONS view in Information Schema

The query profile will also not display any secure UDF details. This applies even to the owner of the secure UDF, because other users may be able to view the owner's query profile.

Snowflake doesn't display the amount of data scanned or the total amount of data for queries using secure UDFs. This is to hide the information about data size from users who only have access to a subset of the data from which they could deduce the total amount of data.

Best Practices for Secure UDFs

Although secure functions prevent users from viewing data that is filtered by the function, there are still ways that data can be exposed.

One such example is using a sequence for generating primary keys in a table. As with secure views, if these keys are visible to users who do not have access to all of the underlying data, they might still be able to guess the data distribution. For this reason, either hide or do not include the sequence-generated column as output from the function, just like you wouldn't show it in a secure view.

Summary

This chapter provided an overview of secure data sharing, followed by discussions of direct shares, Snowflake Marketplace, and data exchange. It also looked at secure views and secure user defined functions (UDFs), which are often used together with secure data sharing.

The next chapter reviews SQL scripting and Snowpark.

CHAPTER 11

SQL Scripting and Snowpark

In addition to querying data using SQL, Snowflake supports working with data programmatically in the form of writing procedural blocks of code, creating stored procedures and functions, and creating API calls.

This chapter reviews Snowflake scripting and Snowpark, two features for working with data programmatically in Snowflake. For the SnowPro Core exam, you don't need to know all the details about writing code, because this is tested on the advanced exams. But you do need an overview understanding of Snowflake scripting and Snowpark, which is described in this chapter.

Snowflake Scripting

When just querying data using SQL statements is not sufficient for your requirements, Snowflake provides scripting constructs that allow you to write procedural logic. You can use Snowflake scripting to write stored procedures, functions, and blocks of procedural code.

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1_11

Snowflake Scripting Blocks

A Snowflake scripting block is made up of sections that are delimited by keywords, such as the following:

```
DECLARE
  -- (variables, cursors, exceptions, result sets)
BEGIN
  -- (Snowflake Scripting and SQL statements)
EXCEPTION
  -- (statements for handling exceptions)
END;
```

Each section can contain various elements that serve different purposes, as follows:

- `DECLARE`. In this section of the block, you can declare any variables, cursors, exceptions, or result sets that you will be using. This is an optional section.
- `BEGIN … END`. The section of the block between the `BEGIN` and `END` keywords is where you write SQL statements and other Snowflake scripting constructs.
- `EXCEPTION.` In this section of the block, you can write any exception handling code, if you will be using it for catching exceptions. This is an optional section.

The most basic Snowflake scripting block requires only the keywords `BEGIN` and `END`. For example, this is a block of code that creates two tables named `FRUIT` and `FRUIT_INVENTORY`:

```
BEGIN
  create table FRUIT (
    fruit_id number,
    name varchar,
```

```
    color varchar
  );
  create table FRUIT_INVENTORY (
    fruit_id number,
    inv_date date,
    amount_kg number
  );
END;
```

To execute the block, just execute it like any SQL query in Snowsight, using a database, schema, and virtual warehouse of your choice.

Note The examples in this chapter use Snowsight. If you happen to be using the Classic UI, refer to the Snowflake documentation on how to format and execute the code.

This code block is called an *anonymous* block because it can be executed independently and is not named. Alternatively, you can use a block in the body of a stored procedure. Here is an example of a stored procedure that contains a Snowflake scripting block:

```
create procedure CREATE_SAMPLE_TABLES()
  returns number
  language sql
  as
BEGIN
  create or replace table FRUIT (
    fruit_id number,
    name varchar,
    color varchar
  );
  create or replace table FRUIT_INVENTORY (
```

```
    fruit_id number,
    inv_date date,
    amount_kg number
  );
END;
```

To execute the stored procedure, use the `CALL` statement, like this:

```
call CREATE_SAMPLE_TABLES();
```

Declaring Variables

In Snowflake scripting, you can declare variables and define their data types. Variables can be used in expressions, SQL statements, or as part of Snowflake scripting constructs. You can define the data type of a variable explicitly by naming the exact data type. Alternatively, you can provide an initial value for the variable and Snowflake will determine the data type based on the value that you have provided.

You can declare a variable within the `DECLARE` section of the block by using one version of the following syntax:

```
<variable_name> <data_type>;
<variable_name> DEFAULT <expression>;
<variable_name> <data_type> DEFAULT <expression>;
```

In these examples, `<variable_name>` is the name of the variable and `<data_type>` is the data type of the variable. The data type can be one of the following:

- SQL data type
- `CURSOR`
- `RESULTSET`
- `EXCEPTION`

Here is an example of a code block where variables are declared in the DECLARE section of the block:

```
DECLARE
  weight_lb number;
  weight_kg number;
BEGIN
  weight_lb := 145;
  weight_kg := 0.453592 * weight_lb;
  return weight_kg;
END;
```

This example declares two variables named WEIGHT_LB and WEIGHT_KG. It assigns the value 145 to the variable WEIGHT_LB. It then uses this variable in a calculation (to convert the value from pounds to kilograms), assigns the calculated result to the variable WEIGHT_KG, and returns the value of this variable. Variables that are declared inside a block cannot be used outside of the block.

You can also declare a variable within the BEGIN and END keywords of the block before you use the variable with the LET command, for example:

```
DECLARE
    weight_kg number;
BEGIN
    let weight_lb := 145;
    weight_kg := 0.453592 * weight_lb;
    return weight_kg;
END;
```

When you declare a variable without explicitly specifying the data type, like in the previous example, Snowflake scripting infers the data type from the expression that you assign to the variable. To avoid ambiguity, especially when an expression can evaluate to more than one data type, Snowflake recommends that you define the data type explicitly.

Assigning Values to Variables

To assign a value to a variable, use the := operator, like this:

```
<variable_name> := <expression>;
```

You can also use a variable in a SQL statement, sometimes referred to as *binding*. In this case, you must prefix the variable name with a colon. For example, to insert the value of the WEIGHT_KG variable into a table, you use the following syntax:

```
insert into FRUIT_INVENTORY (amount_kg) values (:weight_kg);
```

A variable can also be used as the name of an object, such as the name of a table. In this case, enclose the variable by the IDENTIFIER keyword. For example, to select data from a table whose name is stored in the TABLE_NAME variable, you use the following syntax:

```
select sum(amount_kg) from identifier(:table_name);
```

If you are using a variable in an expression or with a Snowflake scripting language element like RETURN, you do not have to prefix the variable with a colon. For example, to return the value of the WEIGHT_KG variable, there is no need for the colon prefix:

```
return weight_kg;
```

When code blocks are used in stored procedures, they can use variables that are passed as parameters to the procedure. For example, create a stored procedure named CONVERT_WEIGHT based on the weight conversion anonymous block from earlier:

```
create procedure CONVERT_WEIGHT()
  returns number
  language sql
  as
```

```
DECLARE
  weight_lb number;
  weight_kg number;
BEGIN
  weight_lb := 145;
  weight_kg := 0.453592 * weight_lb;
  return weight_kg;
END;
```

Of course, this stored procedure isn't very useful because the WEIGHT_LB variable that contains the value of the weight in pounds is defined inside the code block. It would be much more useful if you provide the value for the WEIGHT_LB variable as a parameter to the procedure, like this:

```
create procedure CONVERT_WEIGHT(weight_lb number)
  returns number
  language sql
  as
DECLARE
  weight_kg number;
BEGIN
  weight_kg := 0.453592 * weight_lb;
  return weight_kg;
END;
```

Because the WEIGHT_LB variable is used as a parameter in the stored procedure, it does not need to be declared in the code block. When you use it in the stored procedure, it is referenced without a colon prefix.

To execute the stored procedure, use the CALL statement and supply the value of the input parameter, like this:

```
call CONVERT_WEIGHT(145);
```

Branching and Loops

Snowflake scripting provides various branching and looping statements, including the IF and CASE statements to allow branching, as well as the FOR, WHILE, REPEAT, and LOOP statements, which can be used for looping within the procedural logic.

The following is an example of an IF statement:

```
BEGIN
  let weight_lb := 114;
  IF (weight_lb < 100) THEN
    return 'less than 100';
  ELSEIF (weight_lb = 100) THEN
    return 'exactly 100';
  ELSE
    return 'more than 100';
  END IF;
END;
```

As you can see, the IF statement behaves similarly as IF constructs in many programming languages, including the IF, THEN, ELSEIF, and ELSE keywords.

Note The expression that is evaluated immediately following the IF keyword must be included in parentheses.

A CASE statement behaves similarly to an IF statement, but can have multiple conditions. In a CASE statement, you define different WHEN clauses for different possible values of a given expression. Snowflake executes the first WHEN clause for which the value matches the expression.

For example, to execute different statements based on the value of the FRUIT_NAME variable, you can define a WHEN clause for each possible value

of this variable and provide the corresponding statement to be executed, like this:

```
BEGIN
  let fruit_name := 'banana';
  CASE (fruit_name)
    WHEN 'apple' THEN
      return 'red';
    WHEN 'banana' THEN
      return 'yellow';
    WHEN 'avocado' THEN
      return 'green';
    ELSE
      return 'unknown';
  END;
END;
```

A FOR loop can be based on a counter, similar to many programming languages, to repeat a set of commands as many times as the value of the counter. In Snowflake scripting, a FOR loop can also be used to loop through each row of a result set. For example, the following counter-based FOR loop executes five times:

```
DECLARE
  counter integer default 0;
  maximum_count integer default 5;
BEGIN
  FOR i in 1 to maximum_count do
    counter := counter + 1;
  END FOR;
  return counter;
END;
```

An example of a cursor-based FOR loop is demonstrated in the next section.

Additional types of loops—such as WHILE, REPEAT, and LOOP—are supported in Snowflake scripting. Refer to the Snowflake documentation for syntax details and additional information.

In a loop construct, you can control when the loop should terminate early by executing the BREAK command. This stops the current iteration and skips any remaining iterations. You can use the CONTINUE command to jump to the end of the iteration of a loop, skipping the remaining statements in the loop but continuing with the next iteration.

Cursors

In a Snowflake scripting block, you can use the INTO clause to set variables to the values returned by a SELECT statement. For example:

```
DECLARE
  v_id integer;
  v_name varchar;
  v_color varchar;
BEGIN
  select fruit_id, name, color
    into v_id, v_name, v_color
    from FRUIT
    where fruit_id = 1;
END;
```

When you use this syntax, the first variable V_ID is set to the value of the first column in the SELECT statement; the second variable V_NAME is set to the value of the second column; and so on. However, this syntax works correctly only when the SELECT statement returns a single row—that's why there's an additional WHERE condition in the query that filters for a single FRUIT_ID.

When you want to process multiple rows that are returned from a SELECT statement, you can use a cursor to iterate through query results one row at a time. To use a cursor, do the following:

- Declare the cursor in the DECLARE section of the block by providing the query that will be executed by the cursor.
- In the main section of the code block, between the BEGIN and END keywords, execute the OPEN command to open the cursor, execute the query, and load the results into the cursor. Then execute the FETCH command to fetch each row and process it using looping commands. When you have completed processing the rows, execute the CLOSE command to close the cursor.
- Alternatively, execute the cursor-based FOR loop to iterate over a result set in the main section of the code block, between the BEGIN and END keywords. When using a cursor in a FOR loop, you don't need to open and close the cursor explicitly.

Use the FRUIT and FRUIT_INVENTORY tables that you created earlier and insert some data so that you have an example to work with:

```
insert into FRUIT values
  (1, 'apple', 'red'),
  (2, 'banana', 'yellow'),
  (3, 'avocado', 'green');

insert into FRUIT_INVENTORY values
  (1, sysdate(), 110),
  (2, sysdate(), 80),
  (3, sysdate(), 145);
```

Now declare a cursor for a SELECT statement in the DECLARE section of a block and then process the cursor using a cursor-based FOR loop in the main section of the block. Use the following syntax to do this:

```
DECLARE
  c1 cursor for
    select amount_kg from FRUIT_INVENTORY;
  total_amount number default 0;
BEGIN
  FOR record in c1 do
    total_amount := total_amount + record.amount_kg;
  END FOR;
  return total_amount;
END;
```

In this code block, you declared a cursor named C1 for the `select amount_kg from FRUIT_INVENTORY` query. You declared a variable named TOTAL_AMOUNT with an initial value of 0. In the main section of the code block, between the BEGIN and END keywords, you coded a FOR loop (beginning with the FOR keyword and ending with the END FOR keywords), which used the RECORD variable to iterate through each of the rows returned by the cursor. For each of the rows, the value of the AMOUNT_KG column (referenced as `record.amount_kg`) was added to the TOTAL_AMOUNT variable. After completing the FOR loop, a RETURN statement returned the TOTAL_AMOUNT variable.

Instead of looping through each record of a cursor, you can also return a table from the cursor. For this purpose, use the RESULTSET_FROM_CURSOR function together with the TABLE keyword, like this:

```
DECLARE
  c2 cursor for
    select FRUIT.name, FRUIT_INVENTORY.amount_kg
    from FRUIT_INVENTORY
```

```
    inner join FRUIT
      on FRUIT.fruit_id = FRUIT_INVENTORY.fruit_id;
BEGIN
  open c2;
  return TABLE(resultset_from_cursor(c2));
END;
```

In this code, you declared a cursor named `C2` for a query that joins the `FRUIT` and `FRUIT_INVENTORY` tables and returns some columns in the query result. In the main section of the code block, between the `BEGIN` and `END` keywords, you used an `OPEN` statement that opened the cursor and executed the query. Then you used a `RETURN` statement that returned a table from the code block.

Number of Rows Affected

After each DML command is executed, Snowflake scripting sets the following global variables:

- `SQLROWCOUNT`. Number of rows affected by the last DML statement.
- `SQLFOUND`. True if the last DML statement affected one or more rows.
- `SQLNOTFOUND`. True if the last DML statement affected zero rows.

The following example uses a code block to update a record in the `FRUIT_INVENTORY` table and returns the value of `SQLROWCOUNT`, which indicates how many rows have been updated:

```
BEGIN
  update FRUIT_INVENTORY
    set amount_kg = 175 where fruit_id = '2';
```

```
  return SQLROWCOUNT;
END;
```

In this example, one row was updated and therefore the returned value is 1. Similarly, you could have returned the value of the SQLFOUND variable, which would have been TRUE.

Query ID of the Last Query

Another useful global variable is SQLID. This variable contains the query ID of the last query that was executed.

Using the previous example with a code block where you updated the FRUIT_INVENTORY table, you can return the value of SQLID:

```
BEGIN
  update FRUIT_INVENTORY
    set amount_kg = 150 where fruit_id = '3';
  return SQLID;
END;
```

In this example, the query ID of the UPDATE statement is returned from the code block.

Handling Exceptions

Similar to many programming languages, where you can raise an exception when an error occurs, Snowflake scripting allows you to raise an exception in a scripting block. Exceptions can occur due to various reasons, such as a database error, syntax errors in your scripting code, data type conversion errors, and so on.

Since Snowflake scripting blocks can be nested, exception handlers can be nested as well. When an exception is raised in a Snowflake scripting block, Snowflake looks for an exception handler in the following order of precedence:

- If the block in which the exception occurred has its own exception handler that catches the exception, then that exception handler is executed.
- If the block does not have its own exception handler, then the exception handler in the block that encloses the current block is executed.
- Exception handling nesting continues, following the nesting hierarchy of the scripting blocks. Each time an error occurs, if there is no exception handler in the current block, the exception is sent one layer outwards until an appropriate exception handler is found or until the outermost layer is reached.

To write an exception handler in Snowflake scripting, use the `EXCEPTION` clause. In this section of the block, you can use `WHEN` clauses to address individual exceptions. Some common exceptions are already built-in with Snowflake scripting, but you can also declare your own exceptions if needed. The following built-in exceptions are currently available in Snowflake scripting:

- `STATEMENT_ERROR`. Raised due to an error while executing a statement.
- `EXPRESSION_ERROR`. Raised due to an error related to an expression; for example, if you attempt to assign the value of the expression to a variable with an incompatible data type.

Additional information about built-in exceptions is available via a few variables that are returned by each exception:

- `SQLCODE`. An integer indicating the error code
- `SQLERRM`. A string containing the error message
- `SQLSTATE`. A code based on the ANSI SQL standard

To handle any other exceptions that are not named, use the OTHER keyword in the WHEN clause.

For example, the following code block attempts to select more than one row from the FRUIT table and store the result in a variable using the INTO clause. As you know from earlier, this statement will fail because you can't store more than value in a variable. You will catch the exception in the EXCEPTION section that has been added to the code block:

```
DECLARE
  v_name varchar;
BEGIN
  select name
  into v_name
  from FRUIT;
  return v_name;
EXCEPTION
  WHEN statement_error THEN
    return object_construct('Error type', 'Statement error',
      'SQLCODE', sqlcode, 'SQLERRM', sqlerrm,
      'SQLSTATE', sqlstate);
  WHEN expression_error THEN
    return object_construct('Error type', 'Expression error',
      'SQLCODE', sqlcode, 'SQLERRM', sqlerrm,
      'SQLSTATE', sqlstate);
  WHEN other THEN
    return object_construct('Error type', 'Other error',
      'SQLCODE', sqlcode, 'SQLERRM', sqlerrm,
      'SQLSTATE', sqlstate);
END;
```

After executing this code block, you should see the following value returned from the EXCEPTION section:

```
{
  "Error type": "Statement error",
  "SQLCODE": 92219,
  "SQLERRM": "A SELECT INTO statement expects exactly 1
returned row, but got 3.",
  "SQLSTATE": "22000"
}
```

Similar to other programming languages, once an exception has been handled, the code block itself will not end in an error state. It is up to the downstream programming logic to check if an exception occurred.

More Information

The previous sections provided an overview of Snowflake scripting to give you a basic understanding of what functionalities are available. For more detailed information about Snowflake scripting, refer to the Snowflake documentation.

Snowpark

Snowpark is a developer framework for Snowflake that enables users to write code in their familiar programming language. They can use an API for querying data in Snowflake. This is very efficient because the code is executed inside Snowflake. Use cases for Snowpark include data transformation, data augmentation, machine learning, and implementing many aspects of business logic.

With Snowpark you can create your own user defined functions (UDFs) in one of the supported programming languages, using your preferred external libraries and custom modules. The code is then executed in

Snowflake where the data is also stored. Because your data and code are both contained in a single system, rather than moving between platforms, the system is secure and protected from external threats. As such, it takes advantage of Snowflake's security and governance features.

Currently, Snowpark supports the following programming languages:

- Scala
- Java
- Python

Creating UDFs

In Snowpark, you can write user defined functions (UDFs) in the same language that you use to write your client code—for example, as anonymous functions in Scala or lambda functions in Python.

Here is an example of a UDF written in the Python language:

```
create or replace function CONVERT_WEIGHT_UDF(weight_lb number)
  returns number
  language python
  runtime_version = '3.8'
  handler = 'f_convert_weight'
as
$$
def f_convert_weight(weight_lb):
  return weight_lb * 0.453592
$$;
```

As you can see, the `LANGUAGE` keyword specifies Python as the language that will be used in the function. All Python functions require the `RUNTIME_VERSION` keyword, where you specify the Python version that will be used

to execute the code. The HANDLER keyword indicates the name of the Python function that will be called by the stored procedure; this function must be coded in the main body of the procedure.

To execute the function, use it in a SELECT statement and supply the value of the input parameter, like this:

```
select CONVERT_WEIGHT_UDF(145);
```

Similarly, you can write functions and stored procedures in other supported programming languages. Refer to the Snowflake documentation for more information related to each programming language.

Data Frames

To work with data in Snowpark, you use data frames. Data is loaded into data frames and Snowpark provides methods that you can use to work with the data. To populate a data frame, you use the corresponding methods to retrieve the data from Snowflake, most commonly by executing a SELECT statement. The methods used to work with the data allow you to create sessions, execute SQL statements, use built-in functions, and more.

Another benefit of Snowpark is that code is executed lazily. This means that data is transferred between Snowflake and the client application only when needed, such as by calling a method that evaluates the data frame in your client application. Until then, the data frame in your code is not populated with data.

More Information

For more information about Snowpark, particularly with respect to your preferred programming language, refer to the Snowflake documentation. Keep in mind that details about each of the programming languages supported by Snowpark are not tested on the SnowPro Core exam.

What's Next

This chapter reviewed Snowflake scripting and Snowpark, two features for working with data programmatically in Snowflake.

This is the last chapter in the SnowPro Core certification study companion. Snowflake is constantly evolving and therefore it's possible that some details about Snowflake functionality and the Snowsight user interface that are described in this study companion have changed since it was written. For that reason, you are encouraged to read the Snowflake documentation in addition to this book when you prepare for your exam. Reading the Snowflake documentation serves two purposes:

- It will reinforce your knowledge because you will be reviewing the material a second time.
- It will enable you to check for the most up-to-date information in case any features have been updated since this study companion was written.

Once you have studied all the chapters in this study companion and worked through all the exercises, and you feel confident that you have mastered the material, you are ready to take the exam. Go back to Chapter 1 and follow the guidelines to register for the exam.

Index

A

B

M. Ferle, *SnowPro™ Core Certification Companion*, Certification Study Companion Series,
https://doi.org/10.1007/978-1-4842-9078-1

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X, Y, Z