SQL DOWN UNDER

Dr Greg Low

Snowflake for SQL Server Users

Part 1 - Core Concepts

Snowflake for SQL Server™ Users Part 1 – Core Concepts

Dr Greg Low

SQL Down Under Pty Ltd

@greglow

http://snowflakeforsqlserver.sqldownunder.com

First edition January 2020

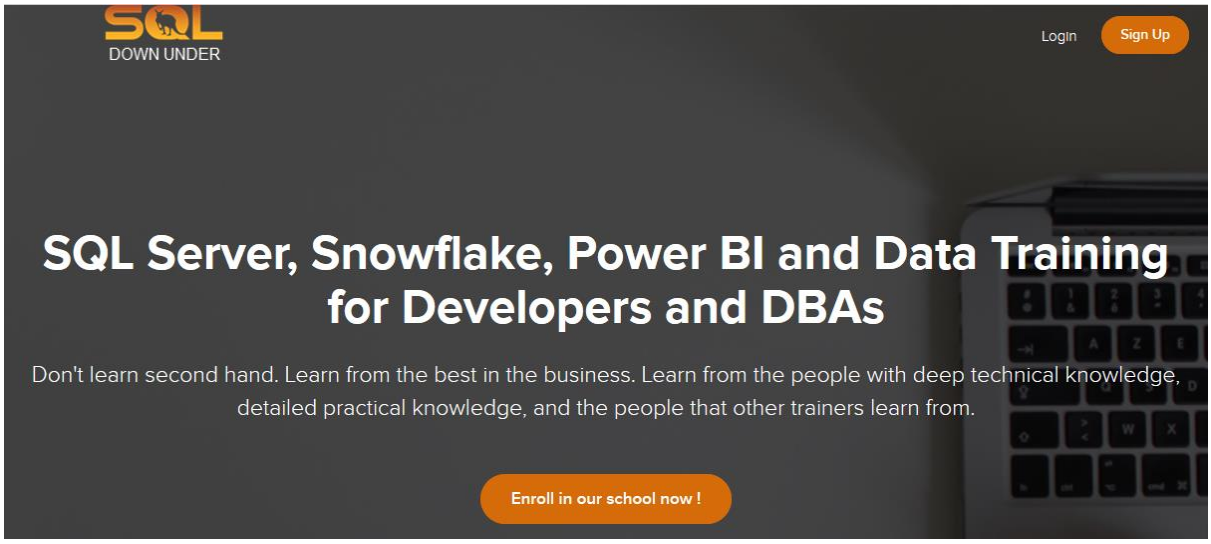
Cover Awesome image by the amazing **Aaron Burden**
(c/- Unsplash https://unsplash.com/photos/5AiWn2U10cw)

The Snowflake product described in this eBook is itself subject to constant change. The content and details described could change at any point in time. We've done our best to make this eBook as error free as possible at the time of publication, but we don't promise that it is error free or that anything we describe will work for you or continue to work for you.

Note from the author:

I've worked with databases like SQL Server™ and Snowflake for decades. This eBook is a compilation of my observations about the differences between the products based upon a series of blog posts that I have either made or are scheduled to be made as part of my Thursday "Snowflake for SQL Server Users" series. (On each heading, I've added the publication date and depending upon when you read this, some might be in the future).

We intend to keep enhancing and upgrading this book. If you have feedback for it, please send that to snowflakeforsqlserverusersbooks@sqldownunder.com



Need to learn about Snowflake or SQL Server ? SQL Down Under offer online on-demand courses that you can take whenever you want. We have many SQL Server courses, and our first Snowflake courses are being built right now.

You can learn with Greg right now !

We're rapidly expanding our list of courses.

Check us out now at http://training.sqldownunder.com

Contents

1 Data Warehouses as a Service

1.1 Why Snowflake?

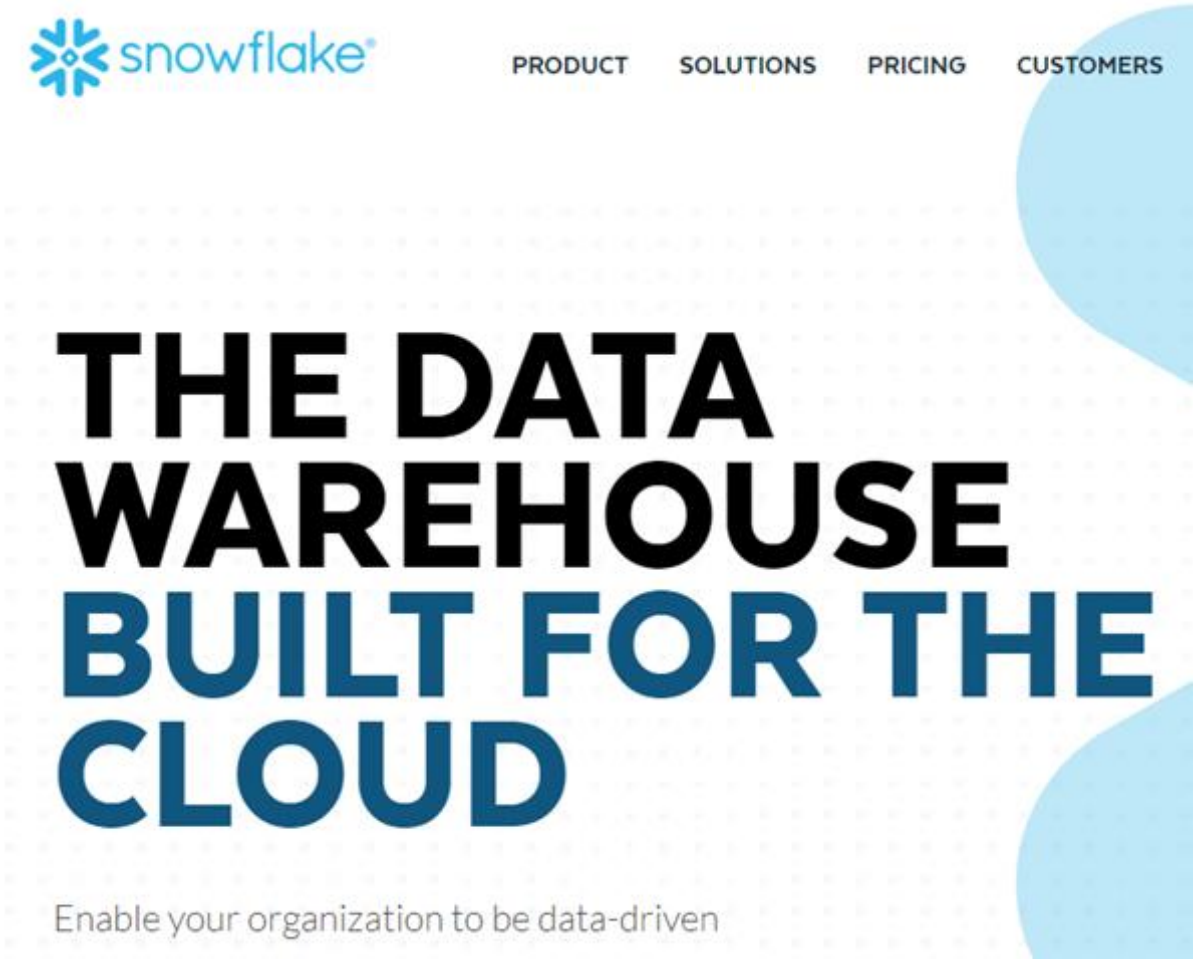


Image from Snowflake website

A few months back, I started noticing that many of our clients had started to mention Snowflake. (www.snowflake.com)

In recent years, I've been in lots of planning and architectural meetings where there was already a presumption that AWS was being used rather than Azure. Now while I'm a big fan of Azure, I put that down to a great selling job by the AWS people who got corporate IT folk locked into large enterprise agreements early. And it means that no matter what the technical question is, the answer will be something that runs on AWS.

I still think that Azure offers a far stronger cloud story than AWS but I'm looking at the whole end-to-end cloud-based story, particularly with platform services. I just see a stronger, more secure, and better envisaged story in Azure.

CLOUD TRANSFORMATION

A big part of this issue is that many organizations that I go into say they're making a "cloud transformation", yet what they're doing is just moving a whole lot of virtual machines into a cloud hosting center.

That's just re-hosting; it's not making a cloud transformation.

For cloud-hosting providers, it's also a no-win game. When all you're doing is re-hosting VMs, you're mostly competing on price, and that's a race to the bottom that no-one really wins.

DATA WAREHOUSES

One bright note on the horizon though is around data warehouses. Almost all these organizations seem to get the idea that they want a cloud data warehouse as a service, not just a bunch of re-hosted virtual machines.

For customers who've already made the decision to use AWS (before we come into the meeting), when they look around for cloud data warehouses, Snowflake is one that's often chosen quickly.

A bonus for me, is that it's now available on both AWS and on Azure.

I've been spending a lot of time lately digging into Snowflake, and I want to explain Snowflake from an existing SQL Server user's perspective. I hope you find it useful.

I'm planning this to be the first book in at least a three-part series:

- Part 1: Core Concepts
- Part 2: SQL Language
- Part 3: Administration

Check back at http://snowflakeforsqlserverusersbooks.sqldownunder.com for updates.

1.2 Cloud-First Design

Awesome image by Dallas Reedy

In recent years, I've done a lot of work in software houses (Microsoft calls them ISVs or Independent Software Vendors). Many of these software houses (vendors) have worked out that they won't be able to just keep selling their on-premises applications because their customers are asking for cloud-based solutions.

Customers want to be able to set up or tear down installations of applications fast. They want to use recurrent expenditure to pay for them, as capital is increasingly difficult to obtain, particularly for new projects, and even more so for proofs of concept.

These customers want the vendors to manage the applications. They don’t want to manage them. Managing applications requires customers to either have highly trained staff or to periodically hire in such staff.

So, many of the vendors start trying to convert their on-premises applications into Software as a Service (SaaS) applications.

And the outcome? Is almost always really, really poor.

CLOUD FIRST

The real problem is that the applications haven't been written with a cloud-first mentality. They really are often the "square peg in a round hole". What often happens is that the cloud-based versions of the applications have large numbers of limitations compared to the on-premises versions of the same applications, because recreating everything in the cloud (when there were things that weren't designed for the cloud) is often nearly impossible.

I'm a big fan of Azure SQL Database, and they've done a great job on it (way better than almost any other application), but it's still quite a distance short of where we already were with SQL Server on-premises. I wish the marketing for the product would focus on how what is there (i.e. a truly amazing

product) but the discussion always seems to be around what's missing, and how it compares to the on-premises product. In fact, I'm sure the entire reason that the Managed Instance versions of Azure SQL Database appeared were to address some of the shortcomings.

If the first time you'd seen SQL Server was to see Azure SQL Database, you'd come away saying how amazing it is. But if you've come from the on-premises product, chances are that you might be already using something that isn't there.

You might recall that Microsoft released a version of SQL Server Reporting Services in Azure, and later removed it. They learned that converting an application into a Software as a Service offering is hard.

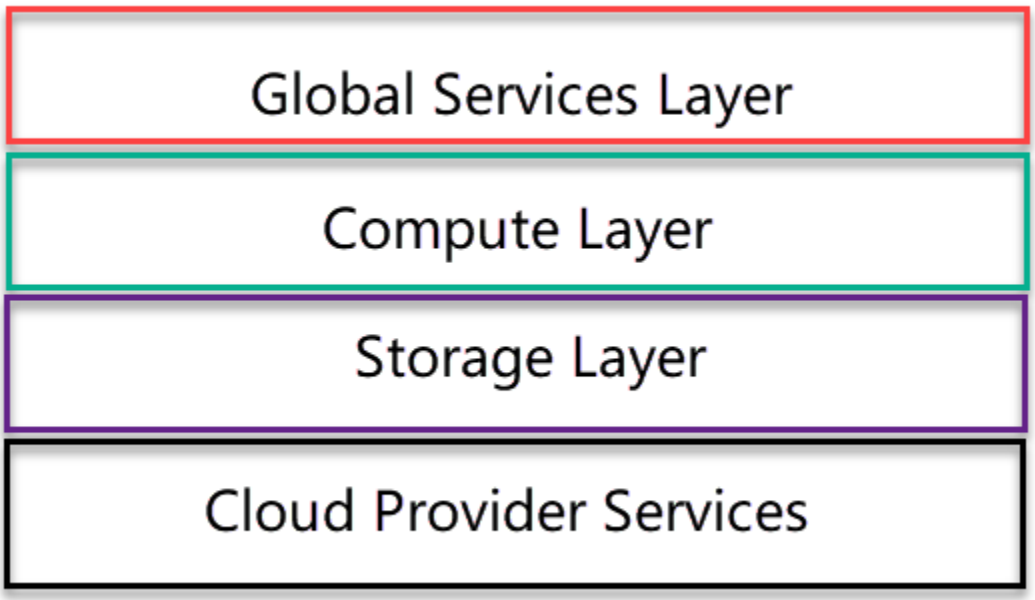
NICE TO HAVE A BLANK SLATE

Even if you were brand new to Azure SQL Database though, you'd find aspects of how the product is designed that are based on thinking from decades ago and were designed for systems that were available back then. It's very hard to make major changes when you're bound to trying to keep backwards compatibility.

One key advantage that the team building Snowflake had was a clean slate where they could design a product that targeted the cloud and the cloud-based services, from day one.

2 Core Architecture

2.1 Service Layers



The first thing to understand about Snowflake is that it has a very layered approach. And the layers are quite independent, including how they scale.

CLOUD PROVIDER SERVICES

The lowest level shown in the diagram above isn't part of Snowflake; it's the services that are provided by the underlying cloud provider. As a cloud native application, Snowflake is designed to use services from the cloud provider that they are deployed to, rather than providing all the services themselves. At the time of writing, that means AWS or Microsoft Azure. Deployment on Google's cloud platform is currently in preview.

Each deployment of Snowflake includes the upper three layers that I've shown in the diagram above.

STORAGE LAYER

This layer is exactly what it says. It uses storage from the cloud provider to provide a way to store anything that needs to be persisted. That includes, the obvious things like databases, schemas, tables, etc. but it also includes less obvious things like caches for the results of queries that have been executed.

Storage is priced separately from compute. While the prices differ across cloud providers and locations, the Snowflake people claim they aren't aiming to make a margin on the storage costs. They're pretty much passing on the cloud provider's cost.

In addition, when you're staging data (that I'll discuss later), you can choose to use Snowflake managed storage or your own storage accounts with a cloud provider to hold that staged data.

COMPUTE LAYER

This is the heart of where Snowflake make their income. The compute layer is made up of a series of virtual clusters that provide the compute power to work on the data. Importantly, each of the virtual clusters (called Virtual Warehouses) can independently access the shared underlying storage.

Compute is charged by consuming what are called "credits". What is interesting is that you only pay for compute while a virtual warehouse is running. Interestingly, some queries that only return metadata can be executed without using a virtual warehouse, but most queries will require one.

While there are some blocking commands, the idea is that you should do most of those types of operations in staging areas, to keep your shared storage accessible in a highly concurrent way. The aim is to have a virtual warehouse happily querying the data, while another virtual warehouse is in the middle of bulk loading other data.

The virtual warehouses can each be different sizes.

GLOBAL SERVICES LAYER

This is the layer where all the metadata and control lives. It's also where transactions are managed, and where security and data sharing details live.

2.2 T-Shirt Sizing of Virtual Warehouses

Size	Description	Number of Servers per Cluster
XS	Extra small	1
S	Small	2
M	Medium	4
L	Large	8
XL	Extra large	16
2XL	Extra extra large	32
3XL	Three times extra large	64
4XL	Four times extra large	128

I the previous topic, I mentioned that the Compute layer of Snowflake is basically made up of a series of Virtual Warehouses (VWs). Each VW is an MPP (massively parallel processing) compute cluster that can comprise one or more compute nodes.

The number of nodes in the compute cluster is called its "size" and the sizing options are made to resemble T-Shirt sizing, as you can see in the diagram above. (These obviously could change in the future but were correct at the time or writing)

Note that XS (extra small) is the smallest VW size. Using an XS for one hour consumes one Snowflake credit. In Australia, in Azure, right now, that's a little over $2 USD.

The number of credits used per hour for a VW is directly related to the number of compute nodes that it contains. So, a 2XL VW consumes 32 credits per hour.

CONCURRENT THREADS

The other important aspect of this sizing is the number of threads. Currently, there are 8 threads per compute node.

This means that the number of concurrent threads for a VW goes from 8 at the XS level, up to 1024 at the 4XL level.

In a later topic, I'll mention threads again because they become important when you're trying to get good parallel loading of files happening, and when you want significant concurrency in query execution.

2.3 Snowflake Editions

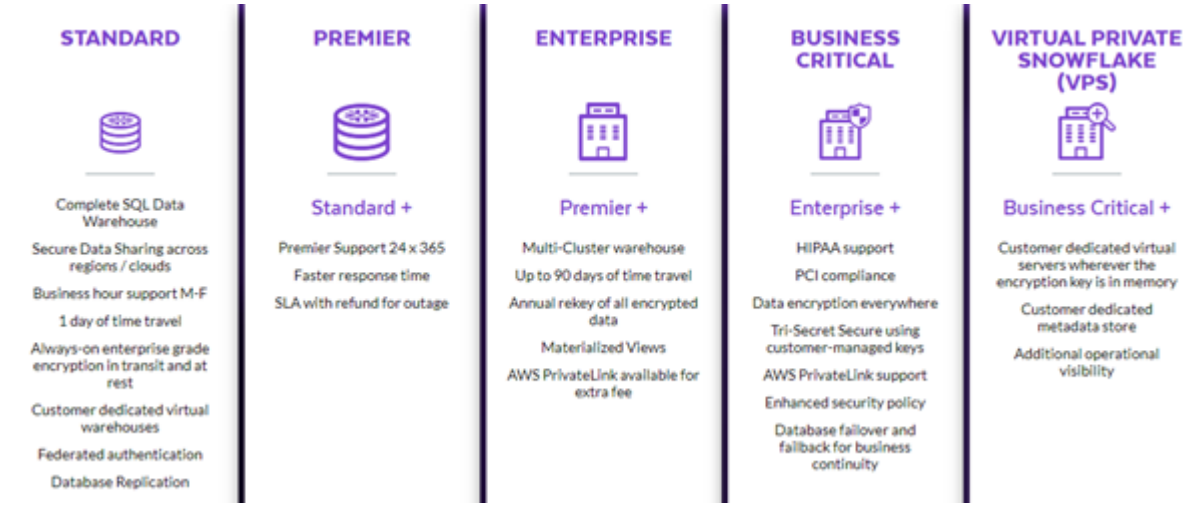


Image from Snowflake website

Like most products, Snowflake comes in several editions, and you can see the current editions in the main image above. (Keep in mind that this was taken from the Snowflake website at the time of writing and could always change at any time. Make sure to check their site for the current options).

First thing I need to say is that I really like the way that most of the SQL code surface is pretty much identical across editions. I wish that was complete coverage, but it currently doesn't include materialized views.

Note: That was a great change when the SQL Server team did the same back in SQL Server 2016 SP1. I think it's important that developers can write application code without worrying which edition of a product it will be deployed against.

There is no free edition like we have with SQL Server Express. Similarly, there's no free option for developers like we have with SQL Server Developer Edition. That's not surprising though, as they aren't the actual cloud provider; they are purchasing services from cloud providers. I find that the Standard edition is pretty low cost though: you only pay a fairly low amount for storage, and you only pay for compute when you use it. So that's not too bad.

If you just want to try the product, there is a trial account you can set up.

DIFFERENCES IN EDITIONS

The main difference between the Standard and Premier editions is that the latter comes with premier support. So that's not a bad distinction from say development tasks, to production tasks.

Greg's Wish: I'd rather see that as just a single edition, with support an optional extra over all editions.

Snowflake has a feature called Time Travel. This allows you to see what data looked like at earlier times. It's a bit like temporal tables but also quite different to it. I discuss it further in later topics.

Standard and Premier both have one day of time travel though, and Enterprise edition takes you up to 90 days. I like to see that sort of feature used as a differentiator between editions. It is unlikely to require application code changes when working with different editions.

Business Critical basically introduces more security. It adds HIPAA and PCI compliance, and the ability to use customer-managed encryption keys.

Greg's Wish: I can't say that I love the idea of core compliance as a distinction between editions. I wish all editions had the core compliance offerings because everyone's data is important to them.

Customer managed keys are a good edition differentiator though.

Snowflake data gets encrypted when stored, and with a key that changes each month (for new data). But on lower editions, it doesn't get re-keyed. What Business Critical also adds is annual key rotation. Data that's a year old gets decrypted and re-encrypted with a new key. I've discussed that in a later topic as well.

VPS – FOR SERIOUS PLAYERS

VPS or Virtual Private Snowflake is for people who can't tolerate the idea of any sort of shared Snowflake infrastructure. The Snowflake team provide a separate deployment of the entire Snowflake stack, just for each customer. It's super expensive (I heard it starts at over $50M AUD) and so I can't imagine too many customers using it, but I'm sure there will be a few, including right here in Australia.

I was told that VPS was only available on AWS at the time of writing, but I'm sure that will change. And I'm guessing if you turn up with $50M+, and say you want it on Azure, it's unlikely they'd say no. (Just a guess 😊)

2.4 Case-Sensitivity

Awesome image by Jason Rosewell

There are many things I like about Snowflake. How they handle case and collations is not one of them. There are currently no rich options for handling case like you have in SQL Server, with detailed options around both collations, and case sensitivity.

I've previously written about how I think that case-sensitivity is a pox on computing. I see absolutely no value in case-sensitivity in business applications, and a significant downside.

Case preservation is a different thing. I expect systems to remember the case that I define things with, but 99.9% of the time, I want to search for them without caring about case. All that case-sensitivity does provide is the ability to have two objects in the same scope that differ only by different capital letters in their names. It's hard to imagine that ever being a good idea.

Snowflake is basically case-sensitive.

Greg's Wish: I hope the Snowflake team fix the collation/case-sensitivity issue before it gets to hard to change. It's a misstep in the product.

OBJECT NAMES

There are, however, some workarounds.

To get around some of the problems that case-sensitivity causes, Snowflake automatically upper-cases object names when you define or use them. By default, Snowflake treats the objects Customers, customers, and CUSTOMERS as the same object.

In fact, if you execute a statement like:

```
CREATE TABLE Customers
```

what it will create is a table called CUSTOMERS.

If you execute

```
SELECT 2 AS Value;
```

you'll get back the value 2 in a column called VALUE, not the column name that you asked for.

That breaks my basic desire (in any language) for case-preservation. Most objects that you see in almost every Snowflake presentation have names that are all capitalized, with underscores separating words. (i.e. what people call SCREAMING_SNAKE_CASE). It makes code much harder to read, in the same way that you'd dislike reading an email from someone that was all in capitals.

DOUBLE QUOTES

You can, however, get around this by quoting each name with double quotes.

```
CREATE TABLE "Customers"
```

And then you need to do that for every table, every column, every object, etc. from then on. If you execute:

```
SELECT 2 AS "Value";
```

you'll get the value 2 with the column name that you're after.

CASE-RELATED COMPARISONS

To get around the idea that most people won't want to compare strings in a case-sensitive way, Snowflake has added operators to deal with case. For example, if you use

```
WHERE "CustomerName" ILIKE 'Fred%'
```

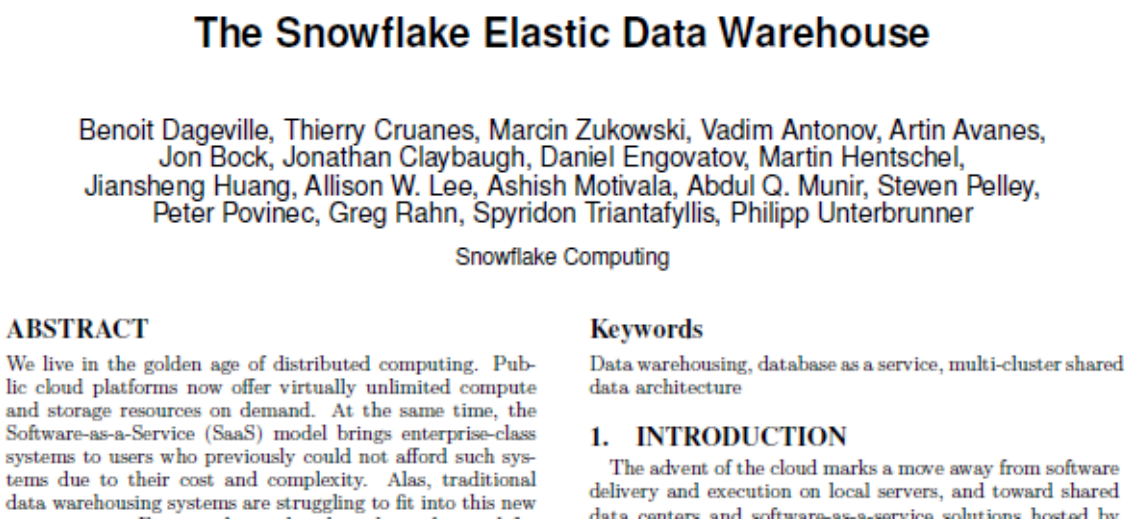
you get a case-insensitive version of LIKE.

NEED FOR CHANGE

The Snowflake people really need to fix how they handle case. You can see from the requests in their user forums that I'm not the only one that thinks so.

This aspect of the product feels very Oracle-like and feels like being back in the 1960s. Humans don't like screaming snake case. I really hope they will fix it soon as it's currently one of the weakest aspects of an otherwise good product.

2.5 Internal Storage in Micropartitions



The Snowflake Elastic Data Warehouse

Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, Philipp Unterbrunner

Snowflake Computing

ABSTRACT

We live in the golden age of distributed computing. Public cloud platforms now offer virtually unlimited compute and storage resources on demand. At the same time, the Software-as-a-Service (SaaS) model brings enterprise-class systems to users who previously could not afford such systems due to their cost and complexity. Alas, traditional data warehousing systems are struggling to fit into this new

Keywords

Data warehousing, database as a service, multi-cluster shared data architecture

1. INTRODUCTION

The advent of the cloud marks a move away from software delivery and execution on local servers, and toward shared

Title of the Snowflake whitepaper at ACM

If you want to learn a lot about how Snowflake works internally, it's worth spending some time reading the ACM whitepaper that described the service.

You'll find it here: https://dl.acm.org/citation.cfm?id=2903741

It describes the overall structure of the service, and then describes how data is stored, followed by a description of how the encryption of the data within the service is accomplished. (More on encryption later)

COLUMNSTORES

Columnstores will be familiar to SQL Server users ever since 2012, and it's no surprise that the storage within Snowflake is essentially columnstore based.

Columnstores are great for adding new data and are excellent for reading large amounts of data in bulk. What they aren't great for, is being updated. Clustered columnstore indexes in SQL Server are updatable but only because they have an associated rowstore that's used for the delta store. In Snowflake, the micropartions that hold the data are immutable. (i.e. they never change)

MICROPARTITIONS

In Snowflake, the columnstores are formed by a series of what are called micropartitions. Each of these is a contiguous storage location that holds up to 16MB of compressed data (50 to 500 MB of uncompressed data), but importantly as I just mentioned, the micropartitions are immutable i.e. once they are created, they are never modified.

The metadata for the micropartitions records the range of values for each of the columns that are stored in the micropartition, the number of distinct values, and a few other properties.

Tables get partitioned automatically during insert and load operations, based upon the order of the incoming data.

Note: Interestingly, the file extension for these is FDN which is short for flocon de neige i.e. the French word for Snowflake.

Query performance against the columnstores is highly dependent upon being able to determine which micropartitions need to be read to satisfy a particular query.

3 Security

3.1 Authentication

Awesome image by Kyle Glenn

Authentication in Snowflake is based around user identities. That's very similar to what we have today with SQL Server authentication (i.e. SQL Server logs you on), as opposed to Windows Active Directory authentication (i.e. Windows AD logs you on) or Azure Active Directory authentication (i.e. AAD logs you on).

To create users, I must be a member of either the SECURITYADMIN or ACCOUNTADMIN roles.

PASSWORD POLICY

Snowflake has a single specific password policy:

- at least 8 chars long
- at least 1 digit
- at least 1 uppercase and 1 lowercase character

As I can with SQL Server logins, I can force a user to change password the first time.

I'm not a fan of these types of password policies, and the NIST recommends against them. Their recommendations are documented here:

https://pages.nist.gov/800-63-3/sp800-63b.html

And there's a good explanation of the core requirements and ideas here:

https://www.enzoic.com/surprising-new-password-guidelines-nist/

Greg's Wish: I would like to see Snowflake (and SQL Server) have a more up-to-date password policy in place.

Users can be disabled/enabled via ALTER USER and you can set default warehouse and role for a user. DESCRIBE USER and SHOW USERS are useful commands for checking out who is configured.

MUTLI-FACTOR AUTHENTICATION (MFA)

Multi-factor authentication is also supported. It's recommended that you should have at least your administrators enrolled for MFA.

Enabling MFA currently means using the Duo application on your phone or device for the extra authentication.

Greg's Wish: To be really good citizens within Azure and within Google Cloud, I'd like to see them also support Microsoft Authenticator and Google Authenticator as well.

INTEGRATION/FEDERATION

OAuth 2.0 is supported, and you can also achieve a degree of single sign on by implementing federation. Snowflake seem to work closely with Octa for single sign on and MFA.

While Snowflake list Azure Active Directory as a potential integration, it really needs a lot more work to integrate properly with AAD.

One big concern for me is that there is currently no concept of granting access to Snowflake or role membership within Snowflake to groups in AAD.

Many of my customers would consider this a showstopper.

Greg's Wish: I hope Snowflake add proper support for AAD based groups.

Because there is no concept of group membership, the DCL commands are GRANT and REVOKE, and there is no concept of a DENY. (DENY is only needed on systems that support groups).

There is also no concept of a logon trigger.

MANAGING GROUPS VS USERS

I really hope they will address concepts like groups and logon/logoff triggers quickly. I do not like to see database administrators getting involved with individual user access. They should manage at a group level, and someone else in the organization should be deciding who is or isn't a member of a group.

Better still would be to see a detailed integration with Azure Active Directory (AAD) and let the directory service do the hard work. For example, there's no way to enforce admins to use MFA or to implement detailed MFA-related policies for other users and so on.

I heard they are looking to implement AAD integration via Octa. Direct integration would be my preference. Full integration with AAD would provide what's needed for this and much, much more.

3.2 Role-Based Security

Awesome image by Liam Tucker

Security in Snowflake is quite different to what's implemented in SQL Server.

OBJECT OWNERSHIP

In SQL Server, objects have owners that can either be specific people, or as occurs when a dbo person creates an object, it is owned by the dbo role.

In Snowflake, objects in the database also have owners, but the owners can't be users, they are always roles.

USING ROLES

When you create a database, there are four pre-defined roles:

- ACCOUNTADMIN
- SECURITYADMIN
- SYSADMIN
- PUBLIC

The ACCOUNTADMIN role is the closest to what we have in SQL Server with the sysadmin role. The SYSADMIN role in Snowflake is much more limited. For example, if a user in the SALESPERSON role creates a table, a SYSADMIN role member cannot even see that table, unless the SYSADMIN role is added to the SALESPERSON role.

Now it's recommended that the SYSADMIN role is added to all other roles, to avoid this issue but there's no concept of a DDL trigger that would allow you to enforce that.

ONE AT A TIME

Another very curious concept is that your security context can only be in one role at a time. That's quite unlike Windows, and SQL Server where you can do whatever any of the roles that you are in can do.

There is a USE Database like we have in SQL Server but there is also USE WAREHOUSE to decide which virtual warehouse will be used for compute costs, and USE ROLE to decide which role you are operating in. So, commands that you execute are always in a context of:

- Which database
- Which warehouse
- Which role

DDL COMMANDS LOOK ODD

While the concept of granting a permission to a role is familiar, and works much the same way, they also use the words GRANT and REVOKE for role membership.

I find that really peculiar. In my mind, roles are things that have members, and the DDL should relate to adding and removing members from roles.

In Snowflake, roles can be nested, and so if, for example, you want to add the role SALESPEOPLE to the role EMPLOYEES, you say:

GRANT EMPLOYEES to SALESPEOPLE;

So, I can't say that I like the way that they've implemented the DDL commands for roles.

Greg's Wish: I think there should be separate DDL for granting and revoking permissions from roles, and for adding and removing members of roles.

SCHEMA-BASED SECURITY

Another thing that I find odd is that when you grant a role permission to SELECT on a schema, it only applies that to objects that are currently in the schema. So that's just like a shortcut way to avoid granting to every object in the schema. Normally when I grant permissions at the schema level, it's for all items currently in the schema, and all objects that will ever be in the schema.

Curiously, Snowflake has an extra permission for that, where you need to separately tell it "oh, and do that for all future objects too". I think that's a poor default choice, but I can live with it.

TRANSFERRING OWNERSHIP

To transfer ownership of an object, you need to:

- Remove all permission grants
- Change ownership
- Re-establish all permission grants

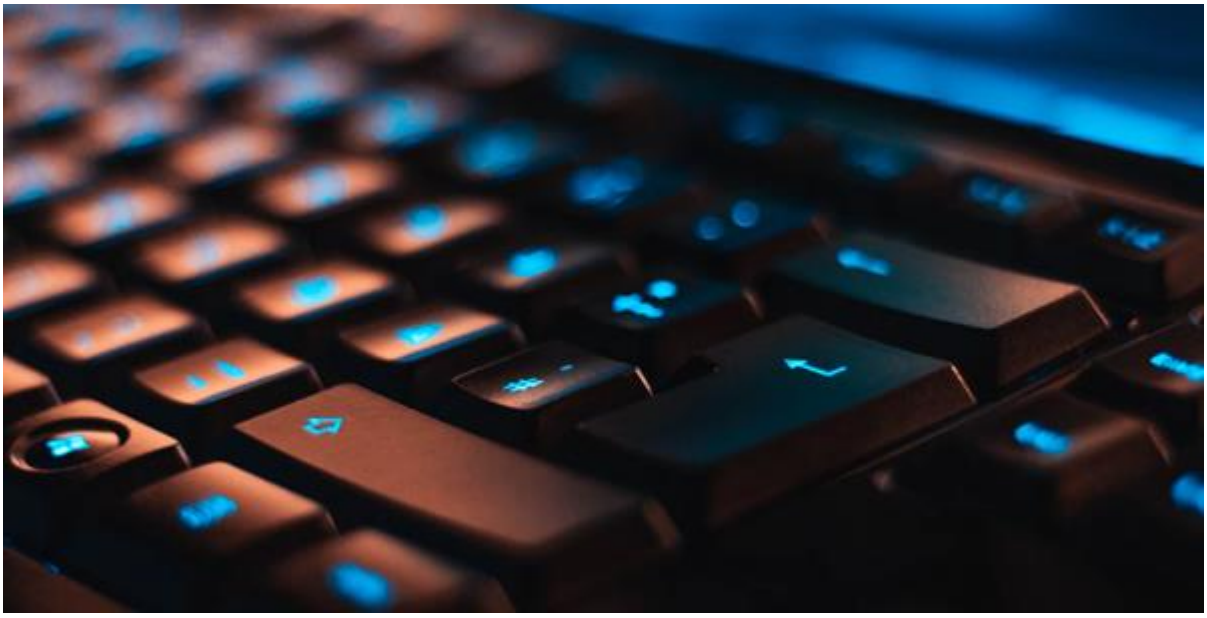
If you delete a role, the role that you are currently executing in when you execute the DELETE, becomes the owning role for all the objects in the deleted role.

CHECKING CURRENT GRANTS

You can check the grants that have been made by executing these commands:

```
SHOW GRANTS TO ROLE
SHOW GRANTS TO USER
```

3.3 Encryption



Awesome image by Christian Wiediger

As with most products today, Snowflake has substantial handling of encryption to protect client data.

All editions claim to provide "Always-on enterprise grade encryption in transit and at rest".

DATA IN TRANSIT

Let's start with "in transit". First, for connections, HTTPS and TLS 1.2 are used throughout the system.

Stages are holding locations for data that's being moved in/out of Snowflake. (We'll look at them later). If the customer is using external stages (holding locations in cloud-storage accounts), the data can be encrypted while stored in the stage, and then travel encrypted into Snowflake, to provide a form of end-to-end encryption.

To do that, you create an "encrypted stage" where you tell Snowflake the master encryption key (i.e. the client side key) when creating the stage:

```
create stage webtraffic_stage
   url='s3://corporate-bucket/data/'
   credentials=(aws _key_id='SDWESS22' aws_secret_key='72828382')
   encryption=(master_key='fSyY1jzYfIntsdfsKOEOxq80Au6NbSgPH5r4BDDwhHfd=');
```

DATA AT REST

Regarding "at rest", earlier I described how micropartitions are used to store data in data files. These data files are then encrypted (using AES-256) before being stored. But how it's handled changes with different editions of Snowflake.

There are four levels of keys used:

- The root key
- Account master keys
- Table master keys
- File keys

With Standard and Premier editions, a different file key is used each month (aka key rotation). So, a single "active" key is used to encrypt your first month's data, then a different key is used to encrypt the next month's data, and so on. Previous keys that are no longer "active" (aka retired) are only used for decryption.

ANNUAL REKEY

Enterprise edition and above offer "Annual rekey of all encrypted data". I was a bit puzzled about how that worked at first. I imagined that once a year, all the existing data would get decrypted and rekeyed. I was thinking that would be quite an expensive operation on a large data warehouse.

What happens instead, is that when any key has been retired for a year or more, any data encrypted by it is decrypted and re-encrypted using a new key. If you have those editions (Enterprise or above), if you store data this month, it will have a single key. In a year and one month's time, it will get rekeyed. And again in two years and one month's time, and so on.

BUSINESS CRITICAL

The Business Critical edition takes things further. As well as providing HIPAA and PCI compliance, it offers **Tri-Secret Secure using customer-managed keys**. To implement this, Snowflake combines the key that you provide with a Snowflake-maintained key to create a composite master key.

As well as having the obvious benefits in letting you manage access to the data more closely, you can immediately remove all access if needed (i.e. closing an account, or recovering from a breach).

4 Tables and Programmable Objects

4.1 Types of Tables

Awesome image by Campaign Creators

Snowflake offers a richer set of options for how tables are created, than we have in SQL Server.

CREATE TABLE VARIANTS

As well as a CREATE TABLE statement as you would already be familiar with in SQL Server, Snowflake offers these variants:

```
CREATE TABLE tablename AS SELECT
```

This is basically like a SELECT INTO in SQL Server. (SQL Server DW also has this) It executes the SELECT query and creates a table from the results (including the data).

```
CREATE TABLE tablename LIKE
```

This is an interesting variant. It creates a table with the same schema as an existing table but without any data i.e. it creates a new empty table based upon the design of another table.

```
CREATE TABLE tablename CLONE
```

This is another interesting variant. It clones one table to create another table. It's the same as the LIKE option but also includes all the data.

TABLE DURABILITY

Permanent

Permanent tables are standard table types that are pretty much the same as the equivalents in SQL Server.

TEMPORARY

This is like a temporary table in SQL Server. The table and its data are retained until the end of the user's session. The syntax supports a concept of LOCAL TEMPORARY and GLOBAL TEMPORARY but these options have no effect. A standard TEMPORARY table is created.

Note that TEMPORARY can be abbreviated to TEMP and has a synonym of VOLATILE.

TRANSIENT

These tables aren't dropped at the end of a user session and stay until someone drops them. They are also visible to all users.

A key difference with them though is that they don't offer the same level of protection as standard tables. They are more like an eventually consistent table where you might lose data if the system fails. They should only be used for data that can easily be recreated if needed.

4.2 Constraints

Awesome image by Jeffrey George

THE USUAL SITUATION

In general database terminology, a primary key is a value that can be used to identify a particular row in a table. The key needs to be unique, and it can't be null. An example would be a CustomerID or CustomerKey in a Customers table. It's called a key rather than a column because it might involve multiple columns i.e. a single primary key might include StateCode and CustomerID.

A foreign key is one or more columns that refer to a key in another table. A common example would be a CustomerID column in an Orders table. Foreign keys can be nullable and are checked when they contain a value.

FOREIGN KEYS IN DATA WAREHOUSES

I like to see foreign keys in data warehouses. The most common objection to them is performance. And yet when I ask those same people if they've ever tested it, I'm usually told that they haven't but that their brother's friend's cousin read it somewhere on the Internet.

Don't be that person.

I also hear **the app does that**, and other excuses.

Yet, almost every time I check a substantial data warehouse for consistency, when it's run for a while without foreign keys being checked, **I invariably find problems**. When I show them to people, I then hear "oh yep, we had that bug a while back…" and so on, but there's almost always an issue.

Even if the app checks the data, other apps touching the same data might not. And what if your ETL (or ELT) processes have bugs? You need to find out about it immediately.

PRIMARY AND FOREIGN KEYS IN SNOWFLAKE

I wish it wasn't so, but while you can define primary and foreign keys on tables in Snowflake, note this from the documentation:

> **Note**
>
> Snowflake supports defining and maintaining constraints, but does not enforce them, except for NOT NULL constraints, which are always enforced.

Yep, they are ignored. They are not checked at all.

Same deal for unique constraints. (Mind you, unique constraints are really a broken concept in SQL Server as well. While they are checked, SQL Server only allows a single row where the key is null. That's not good either).

And that's why there are people complaining in the Snowflake forums about issues caused by duplicate primary keys.

Greg's Wish: I really think this aspect of the product needs to be reconsidered and I encourage the Snowflake team to do so.

For many sites I work at, this single aspect would be a showstopper.

4.3 Data Clustering in Tables

Awesome image by Pierre Barman

In SQL Server, most tables benefit from having a clustering key i.e. the column or columns that the table is logically sorted by.

Note: much old SQL Server training material and documentation used to say "physical order" and that's never been true.

To do that, SQL Server stores the data within the pages in a sorted way and maintains a doubly-linked logical chain of pages. In addition, it maintains pointers to the data by creating an index on it.

CLUSTERING IN SNOWFLAKE

By comparison, Snowflake's design tends to encourage you to avoid creating clustering keys on tables. For most tables, the data will automatically be well-distributed across the micropartitions.

You can assess how well that's working by checking if there are multiple partitions with overlapping values. You can see an image of that in the documentation here:

https://docs.snowflake.net/manuals/user-guide/tables-clustering-micropartitions.html#clustering-depth-illustrated

The more overlap there is, the greater the clustering depth. You can see that by querying the **SYSTEM$CLUSTERING_DEPTH** system function. It's also possible to see all the most important clustering details (including the depth) by querying the **SYSTEM$CLUSTERING_INFORMATION** system function.

SOMETIMES YOU NEED TO CLUSTER

This is all good, but sometimes you need to cluster a table, particularly if you've had a lot of INSERT, UPDATE, DELETE style operations that have been executed.

Note: this is only an issue if it's changed the columns that were involved in the auto-clustering and has introduced skew and additional clustering depth.

The aim of the clustering (which, as noted, isn't for all tables) is co-locate the related table data the same micro-partitions, and to minimize clustering depth.

Generally, you'll only do this for very large tables. And you'll know it's time to do it when your queries have slowed down markedly, and the clustering depth has increased.

MICROPARTITION PRUNING

Micropartition pruning is eliminating micropartitions that aren't needed for a query. It's somewhat like partition elimination and clustered columnstore segment elimination in SQL Server. And it's important for query performance.

But there's a cost

While it might seem obvious to then keep all the tables clustered like we often do in SQL Server, the more frequently the data in the table changes, the more work is involved in maintaining the clustering.

However, if you have tables that don't change much and are queried all the time, clustering could produce a great outcome.

A FEW OTHER NOTES ON CLUSTERING

Unlike applying a clustering key to a SQL Server index (via a primary key or a clustered index), the change isn't applied immediately. It's done in the background.

You also want to avoid having high cardinality columns as clustering keys. As an example, if you have TIMESTAMP columns (somewhat like SQL Server datetime or datetime2 columns), you'd be better off adding an expression (like a computed column) that truncated that to a date, and then clustering on that.

Reclustering a table in Snowflake is automatic and is performed when it becomes quite skewed.

4.4 Time Travel

Awesome image by Andrik Langfield

One important concept in data warehouses is the idea of versioning. Imagine that I have customers broken into a set of business categories. And then I change which customers are in which categories. For so many transactional systems, if I run a sales report for last year, the customers would now appear in the current business categories, rather than the ones they were part of last year.

In Kimbal-based designs, that's part of the rationale for what are called **Slowly-Changing Dimensions** (SCDs).

SQL SERVER TEMPORAL TABLES

SQL Server 2016 introduced temporal tables. Each temporal table has two components: one table that holds the current data, and another table that holds the history for the same data i.e. the previous versions of the rows. While the SQL Server implementation is interesting, in practice, I've found it lacking in many ways.

The biggest challenge is the lack of temporal joins. Even though I can view the contents of a table at one point in time, and I can view all the versions of rows from the table over a period of time, there are real challenges when you need to work across joins. There's no way to get rows from one table, based upon the timeframe of the rows from another table. When I created the WideWorldImporters sample databases for SQL Server 2016, you'll find a whole bunch of cursor-based code in the

procedures that extract data changes. I really didn't want to do that but couldn't see any other way to achieve it.

SNOWFLAKE AND TIME TRAVEL

The standard mechanism for accessing historical data in Snowflake is to use what's called **Time Travel**. The core things that Time Travel lets you do are:

- Run queries to see the previous version of data at a given time
- Create a clone of a table based upon a previous version of its data (you can also do this for schemas and databases)
- Recover previous versions of objects that have been dropped

Because this historical data is available, there is even an UNDROP command that works for tables, schemas, and databases.

When you want to use historical versions of the data, in a SELECT statement (or in a CREATE CLONE statement), you can specify AT for a specific time, or BEFORE to exclude the given time. Then when you specify the time, you have three options:

- Specify a specific time via the TIMESTAMP option
- Specify a relative time from the current time by using the OFFSET option with a number of seconds
- Specify an individual statement via a Query ID

The queries look like this:

```
select *
from customers
at(timestamp => 'Tue, 03 December 2019 06:20:00 +1000'::timestamp);


select *
from customers
at(offset => -60 * 20);


select *
from customers
before(statement => '4f5d0db9-105e-45ec-3434-b8f5b37c5726');
```

QUERY HISTORY AND QUERY ID

This last option is a curious one for SQL Server people. Snowflake keeps track of previous queries that it has executed. Each one is identified by a Query ID.

There are many things you can access via this Query ID like the query plan involved.

RETENTION OF HISTORY

The retention period for Time Travel is based upon the edition of Snowflake that you are using.

- All editions can use a retention period of one day
- Enterprise Edition allows up to 90 days for permanent objects, but only one day for transient and temporary objects.

In Enterprise Edition, you can also specify a specific period for specific objects by using the DATA_RETENTION_TIME_IN_DAYS parameter when you are creating the objects.

If you really don't want to use Time Travel (and the space it will use), you can set DATA_RETENTION_TIME_IN_DAYS to 0 at the account level.

4.5 Programmable Objects

Awesome image by Kevin Ku

Like SQL Server, Snowflake has ways of creating programmable objects. But the way they work, and the way they are created is quite different.

FUNCTIONS

Functions are the most similar. You can create them in two ways:

- Javascript
- T-SQL

I like the idea that you can choose which language to write code in, and that they both end up extending the Snowflake SQL language pretty much the same. Some code is better written in T-SQL and other code is better written in a higher-level language.

Functions are called as expected, in place of an expression.

Both scalar functions and table-valued functions are supported. Scalar functions must return a value.

STORED PROCEDURES

There is the ability to write stored procedures, but curiously, you can only do that in JavaScript.

I have to say I've never worked with a SQL database engine before that supports stored procedures but won't let you write stored procedures in SQL. I think this is quite a shortcoming in the product.

Stored procedures are called using the CALL statement (not EXEC as in SQL Server). Another curious aspect is that even though the stored procedures support a return value, the syntax for calling stored procedures via CALL doesn't support retrieving a return value. I have to say, that's quite bizarre.

You can pass values back from stored procedures by using temporary tables. Or if the returned data is small enough, you might be able to stuff it into a variant data type object and return that.

Stored procedures can be nested.

Greg's Wish: I hope the product adds support for creating stored procedures in SQL and handles return values better.

TRIGGERS ARE MISSING

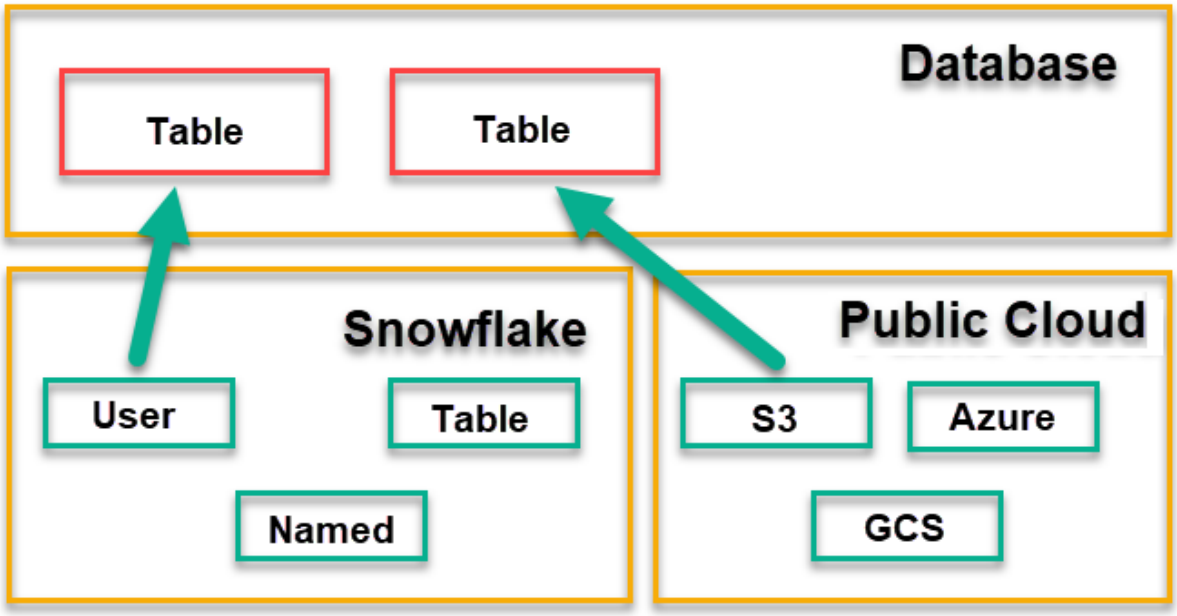
At the time of writing, there is currently no concept of a trigger in Snowflake. And that means neither DML (INSERT/UPDATE/DELETE) triggers nor DDL (CREATE/ALTER/DROP/LOGON) triggers.

As triggers are often a necessary evil in some applications, this again is a significant shortcoming of the product right now.

Greg's Wish: I'd like to see support for triggers added, and preferably both DML and DDL triggers.

5 Loading and Exporting Data

5.1 Stages



Snowflake has the normal options in its SQL language for using INSERT statements to add data to a table. But it also has bulk options for rapidly adding a lot of data to a table via the COPY command. The same command can be used for bulk export.

A curious aspect of this though, is that because it's a cloud-only database system, you can't just use COPY to get data to/from a local file system. COPY works to/from what's called a **stage**.

A stage is a cloud-based storage location. It's called that as it's used as a staging location for data.

TYPES OF STAGES

There are two basic types of stages: the ones provided within Snowflake itself, and the ones that use storage locations in public cloud providers. For the public cloud providers, you can currently choose one of:

- AWS S3 bucket
- Azure storage location
- Google cloud storage (GCS)

When you work with these, you can specify them as an external location (where you put the full address) each time, or you can create an external stage. An external stage is just a name that you give to details of the location. That lets you just use a name, and not have to repeat details of where the location is, all through your code.

The stages provided by Snowflake are as follows:

- Each user has a default stage when they are connected. A user stage is referred to by the name **@~** and is automatically created for you.
- Each table in the database also has a default stage associated with the name of the table. You refer to a table stage as **@%tablename** and again, these are automatically created for you.
- You can also ask Snowflake to create a stage for you. You create an internal named stage by using the CREATE STAGE command. These are normally permanent but you can also make them session-temporary by adding the word TEMPORARY when creating them.

5.2 File Formats

Awesome image by Pranav Madhu

One thing that I quite like about Snowflake is the way it cleanly works with a wide variety of file formats.

At the time of writing, you could COPY from the following source file formats:

- CSV
- JSON
- AVRO
- ORC
- PARQUET
- XML

There are also many options for configuring how these are used. Apart from the obvious options like record and field delimiters, skipping rows, etc., one of the most important of these options is compression. You can currently choose these options for compression:

- AUTO
- GZIP
- BZ2
- BROTLI
- ZSTD
- DEFLATE
- RAW_DEFLATE
- NONE

I had good outcomes using the AUTO setting. Mostly I'm using zipped input files.

There are a few options for transforming data while loading it, but they are very limited. For example, you can alias a column or change its position.

NAMED FILE FORMATS

As with the way you can avoid specifying full file locations every time by creating an external stage, you can avoid specifying all your file format details every time by creating a named file format.

No surprise, that's done with the **CREATE FILE FORMAT** command.

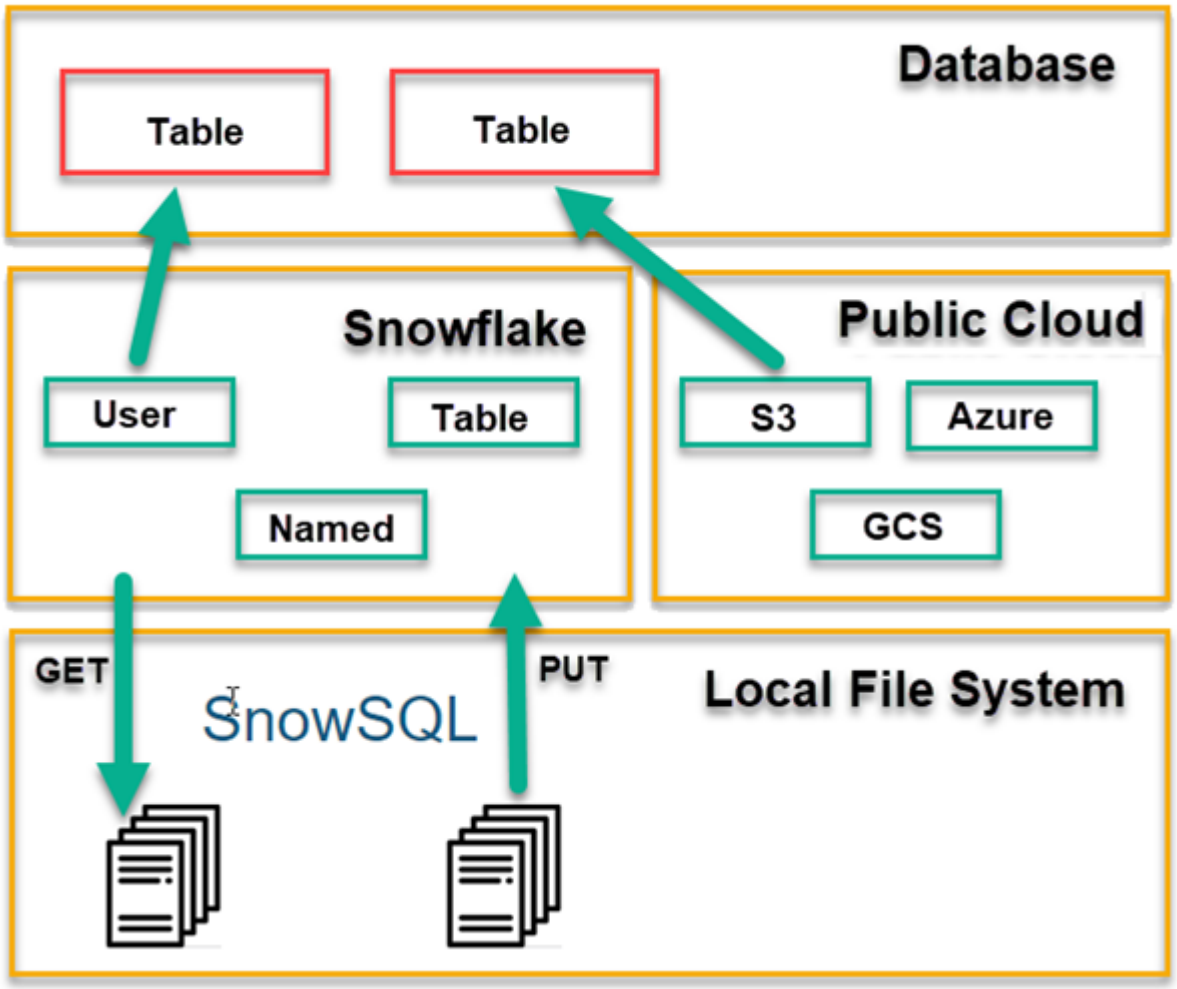
DATA EXPORT

When you export data, you again use the COPY command, but currently there are less file format options available than there are for input. You can use:

- CSV
- JSON
- PARQUET

I would be surprised if the others aren't added soon.

5.3 GET, PUT, and SnowSQL for working with local files



Earlier, I talked about stages. They are internal or external cloud storage locations that you can use the COPY command to copy data into database tables from or use the COPY command to export data from database tables.

Now if you are using external stages, they're just standard storage accounts in AWS (S3), Azure Storage, or Google (GCS). You can use whatever tools you want to get files from other locations (like your local file system) to/from these accounts.

But if you want to get data in or out of the internal stages, you need to talk to the Snowflake API. While you could write your own code to do that, the easiest way to do that is by using a tool called **SnowSQL**.

SNOWSQL

Snowflake has many built-in connectors. SnowSQL runs on 64bit versions of Windows, macOS, and Linux. It's built using the Snowflake Connector for Python. That connector is just straight Python coding. On Windows, for Python v2, you need 2.7.9 or higher. For Python v3, you need 3.5.0 or higher.

SnowSQL is a command line client that you can run interactively as a shell, or you can run in batch mode. Commands can be referenced via a -f parameter (common on Windows) or redirected into it via stdin (common on Linux and macOS).

Using SnowSQL is very much like using OSQL or SQLCMD. Like SQLCMD though, it has a rich command language of its own with variables, auto-complete, command line history, and variable substitution.

SNOWSQL COMMANDS

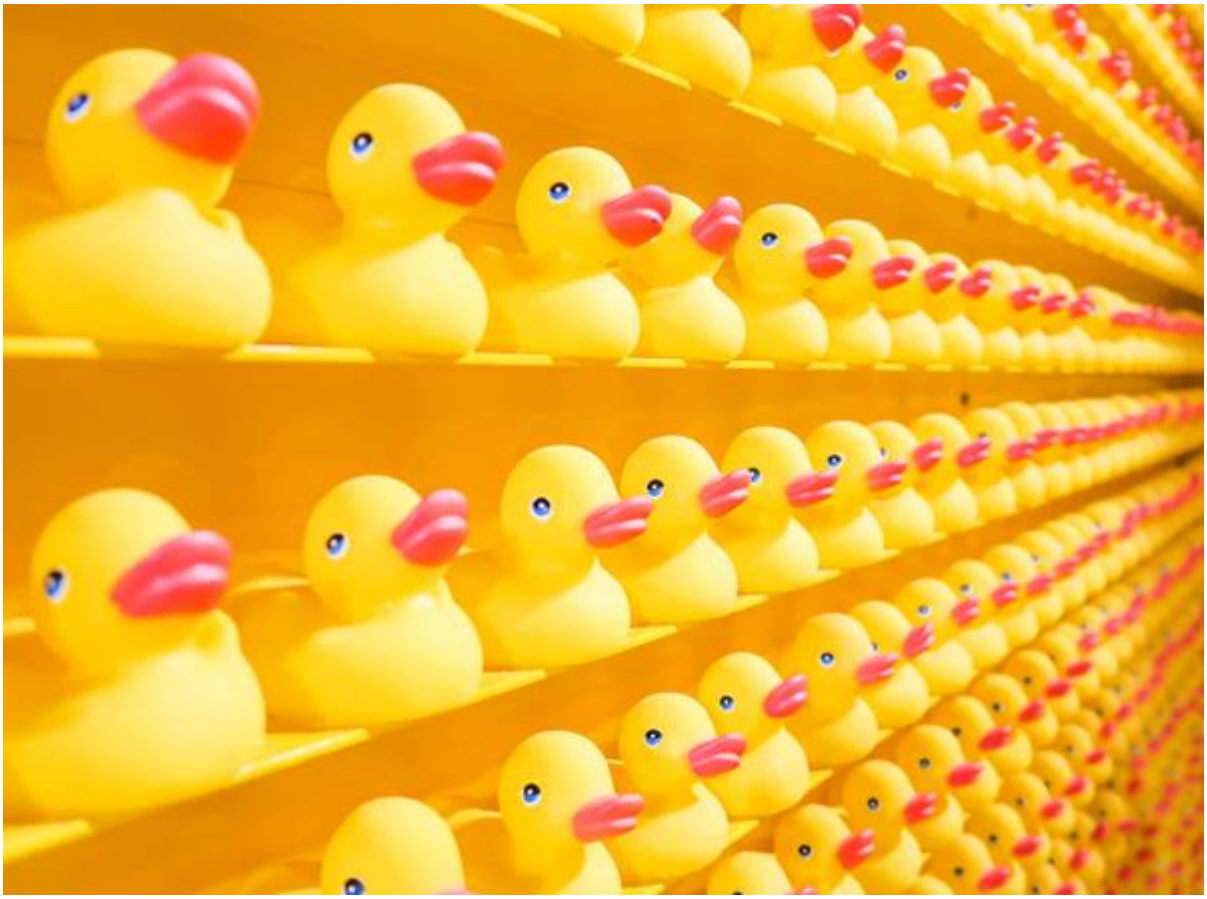
You can run SQL commands through SnowSQL but most people would use other tools for that. The commands that most people will use SnowSQL for are the **GET** and **PUT** commands. These are not SQL commands but SnowSQL commands.

- GET is used to retrieve files from the internal stages
- PUT is used to upload files into the internal stages

Other useful (and related) commands are:

- LIST is used to list the files in either internal or external stages
- REMOVE is used to delete files from the internal stages

5.4 Parallelism when loading data from files into tables



Awesome image by Joshua Coleman

When you are loading data into Snowflake, it's important to achieve the maximum parallelism that you can. You want as many files loading in parallel as you have. I mentioned earlier that the number of virtual warehouses that you have, and the size of each of those virtual warehouses, will determine the number of processor threads that are available to you.

It would be pointless to have 32 processor threads waiting to load your data, and you provide Snowflake with one large file to load.

Instead, you should consider having at least as many smaller files to load as you have available processor threads. This means that you should favor many small files, rather than a smaller number of larger files.

BREAKING UP LARGE FILES

This means that it can be desirable to break up existing large files. When I was doing Snowflake training, several people asked about good options for breaking up large text files. We couldn't find a good option, so I wrote one.

You can find details of SDU_FileSplit here:

https://blog.greglow.com/2019/08/23/sdu_filesplit-free-utility-for-splitting-csv-and-other-text-files-in-windows/

As well as breaking up the files, it can transfer the header rows into the split files and more.

On Linux, you could just use the split command line utility.

FILE SIZES

In terms of how large the files should be, the Snowflake documentation recommends 10MB to 100MB of compressed data. (Most people load zip files).

That also means that if your files are much smaller, you should aggregate them before loading them, to get them into the target file size.

6 Recovery

6.1 Fail-Safe

Awesome image by John Salvino

In a previous topic, I talked about how historical data in Snowflake can be accessed by a concept called Time Travel. But Time Travel isn't the only way to get to historical data.

In Snowflake, **Fail-safe** is a system that provides protection against system failures and other nasty events like hardware failures or serious security breaches.

Immediately after the Time Travel retention period ends, a Fail-safe period begins. It lasts for a non-configurable period of seven days.

Fail-safe isn't intended for users to access like they do with Time Travel. Instead, Fail-safe data is held so that Snowflake support can recover data that has been lost or damaged.

WHAT ABOUT BACKUP?

With Snowflake, there is no concept of a standard backup. The argument is that their multi-data center and highly redundant architecture almost removes the need for backup. And the thinking is that Fail-safe removes that final risk.

The only issue I see with that logic, is the same as with most cloud-based databases and data warehouses: Users sometimes want to keep point in time backups over long periods. I don't currently see any option available for this within Snowflake. But as I mentioned, they aren't alone in that.