



BEST PRACTICES FOR OPTIMIZING YOUR DBT AND SNOWFLAKE DEPLOYMENT

TABLE OF CONTENTS

Introduction	3	Optimizing dbt	22
What Is Snowflake?	3	Use environments	22
Snowflake architecture	3	Use the ref() function and sources	24
Benefits of using Snowflake	4	Write modular, DRY code	25
What Is dbt?	5	Use dbt tests and documentation	26
dbt Cloud	5	Use packages	27
Customer Use Case	6	Be intentional about your materializations	27
Optimizing Snowflake	6	Optimize for scalability	28
Automated resource optimization for dbt query tuning	8	- Plan for project scalability from the outset	28
- Automatic clustering	8	- Follow a process for upgrading dbt versions	28
- Materialized views	8	Conclusion	28
- Query acceleration services	9	Contributors	29
Resource management and monitoring	10	Reviewers	29
- Auto-suspend policies	10	Document Revisions	29
- Resource monitors	11	About dbt Labs	30
- Naming conventions	12	About Snowflake	30
Role-based access control (RBAC)	13		
- Monitoring	13		
- Monitoring credit usage	15		
- Monitoring storage usage	15		
Individual dbt workload elasticity	17		
- Scaling up for performance	18		
- Scaling out for concurrency	20		
Writing effective SQL statements	20		
- Query order of execution	20		
- Applying filters as early as possible	21		
- Querying only what you need	21		
- Joining on unique keys	21		
- Avoiding complex functions and UDFs in WHERE clauses	22		

INTRODUCTION

Companies in every industry acknowledge that data is one of their most important assets. And yet, companies consistently fall short of realizing the potential of their data.

Why is this the case? One key reason is the proliferation of data silos, which create expensive and time-consuming bottlenecks, erode trust, and render governance and collaboration nearly impossible.

This is where Snowflake and dbt come in.

The **Snowflake Data Cloud** is one global, unified system connecting companies and data providers to relevant data for their business. Wherever data or users live, Snowflake delivers a single and seamless experience across multiple public clouds, eliminating previous silos.

dbt is a transformation workflow that lets teams quickly and collaboratively deploy analytics code following software engineering best practices such as modularity, portability, CI/CD, and documentation. With dbt, anyone who knows SQL can contribute to production-grade data pipelines.

By combining dbt with Snowflake, data teams can collaborate on data transformation workflows while operating out of a central source of truth. Snowflake and dbt form the backbone of a data infrastructure designed for collaboration, agility, and scalability.

When Snowflake is combined with dbt, customers can operationalize and automate Snowflake's hallmark scalability within dbt as part of their analytics engineering workflow. The result is that Snowflake customers pay only for the resources they need, when they need them, which maximizes efficiency and results in minimal waste and lower costs.

This paper will provide some best practices for using dbt with Snowflake to create this efficient workflow.

WHAT IS SNOWFLAKE?

Snowflake's Data Cloud is a global network where thousands of organizations mobilize data with near-unlimited scale, concurrency, and performance. Inside the Data Cloud, organizations have a single unified view of data so they can easily discover and securely share governed data, and execute diverse analytics workloads. Snowflake provides a tightly integrated

analytics data platform as a service, billed based on consumption. It is faster, easier to use, and far more flexible than traditional data warehouse offerings.

Snowflake uses a SQL database engine and a unique architecture designed specifically for the cloud. There is no hardware (virtual or physical) or software for you to select, install, configure, or manage. In addition, ongoing maintenance, management, and tuning are handled by Snowflake.

All components of Snowflake's service (other than optional customer clients) run in a secure cloud infrastructure.

Snowflake is cloud-agnostic and uses virtual compute instances from each cloud provider (Amazon EC2, Azure VM, and Google Compute Engine). In addition, it uses object or file storage from Amazon S3, Azure Blob Storage, or Google Cloud Storage for persistent storage of data. Due to Snowflake's unique architecture and cloud independence, you can seamlessly replicate data and operate from any of these clouds simultaneously.

SNOWFLAKE ARCHITECTURE

Snowflake's architecture is a hybrid of traditional shared-disk database architectures and shared-nothing database architectures. Similar to shared-disk architectures, Snowflake uses a central data repository for persisted data that is accessible from all compute nodes in the platform. But similar to shared-nothing architectures, Snowflake processes queries using massively parallel processing (MPP) compute clusters where each node in the cluster stores a portion of the entire data set locally. This approach offers the data management simplicity of a shared disk architecture, but with the performance and scale-out benefits of a shared-nothing architecture.

As shown in Figure 1, Snowflake's unique architecture consists of three layers built upon a public cloud infrastructure:

- **Cloud services:** Cloud services coordinate activities across Snowflake, processing user requests from login to query dispatch. This layer provides optimization, management, security, sharing, and other features.
- **Multi-cluster compute:** Snowflake processes queries using virtual warehouses. Each virtual warehouse is an MPP compute cluster composed of multiple compute nodes allocated by Snowflake from Amazon EC2, Azure VM, or Google Cloud Compute. Each virtual

warehouse has independent compute resources, so high demand in one virtual warehouse has no impact on the performance of other virtual warehouses. For more information, see “[Virtual Warehouses](#)” in the Snowflake documentation.

- **Centralized storage:** Snowflake uses Amazon S3, Azure Blob Storage, or Google Cloud Storage to store data into its internal optimized, compressed, columnar format using micro-partitions. Snowflake manages the data organization, file size, structure, compression, metadata, statistics, and replication. Data objects stored by Snowflake are not directly visible by customers, but they are accessible through SQL query operations that are run using Snowflake.

BENEFITS OF USING SNOWFLAKE

Snowflake is a cross-cloud platform, which means there are several things users coming from a more traditional on-premises solution will no longer need to worry about:

- **Installing, provisioning, and maintaining hardware and software:** All you need to do is create an account and load your data. You can then immediately connect from dbt and start transforming data.
- **Determining the capacity of a data warehouse:** Snowflake has scalable compute and storage, so it can accommodate all of your data and all of your users. You can adjust the count and size of your virtual warehouses to handle peaks and lulls in your data usage. You can even turn your warehouses completely off to stop incurring costs when you are not using them.

- **Learning new tools and expanded SQL capabilities:** Snowflake is fully compliant with ANSI-SQL, so you can use the skills and tools you already have. Snowflake provides connectors for ODBC, JDBC, Python, Spark, and Node.js, as well as web and command-line interfaces. On top of that, Snowpark is an initiative that will provide even more options for data engineers to express their business logic by directly working with Scala, Java, and Python Data Frames.
- **Siloed structured and semi-structured data:** Business users increasingly need to work with both traditionally structured data (for example, data in VARCHAR, INT, and DATE columns in tables) as well as semi-structured data in formats such as XML, JSON, and Parquet. Snowflake provides a special data type called VARIANT that enables you to load your semi-structured data natively and then query it with SQL.
- **Optimizing and maintaining your data:** You can run analytic queries quickly and easily without worrying about managing how your data is indexed or distributed across partitions. Snowflake also provides built-in data protection capabilities, so you don't need to worry about snapshots, backups, or other administrative tasks such as running VACUUM jobs.
- **Securing data and complying with international privacy regulations:** All data is encrypted when it is loaded into Snowflake, and it is kept encrypted at all times when at rest and in transit. If your business requirements include working with data that requires HIPAA, PII, PCI DSS, FedRAMP compliance, and more, Snowflake's Business Critical edition and higher editions can support these validations.

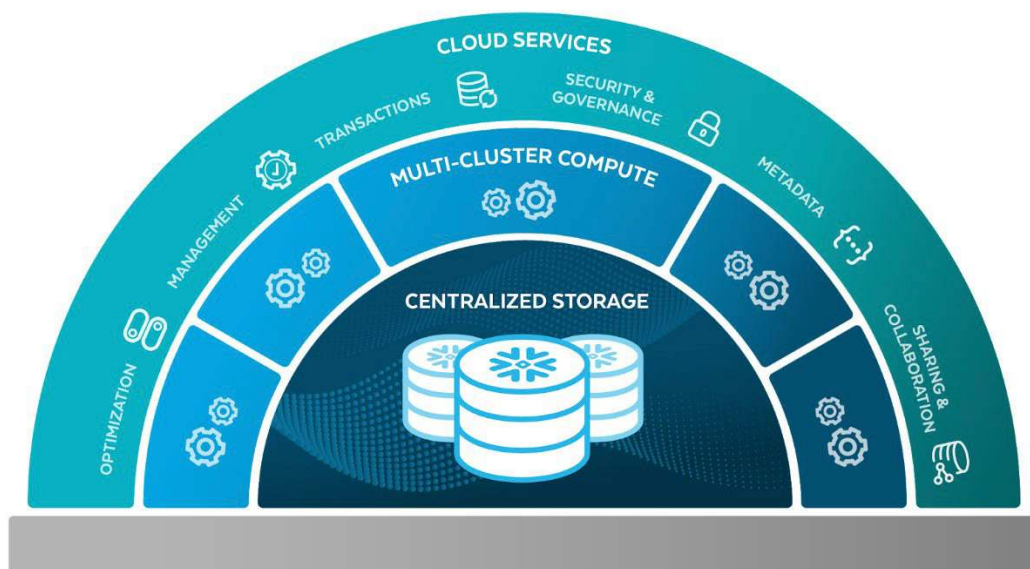


Figure 1: Three layers of Snowflake's architecture

- **Sharing data securely:** Snowflake Secure Data Sharing enables you to share near real-time data internally and externally between Snowflake accounts without copying and moving data sets. Data providers provide secure data shares to their data consumers, who can view and seamlessly combine the data with their own data sources. Snowflake Data Marketplace includes many data sets that you can incorporate into your existing business data—such as data for weather, demographics, or traffic—for greater data-driven insights.

WHAT IS DBT?

When data teams work in silos, data quality suffers. dbt provides a common space for analysts, data engineers, and data scientists to collaborate on transformation workflows using their shared knowledge of SQL.

By applying proven software development best practices such as modularity, portability, version control, testing, and documentation, dbt's analytics engineering workflow helps data teams build trusted data, faster.

dbt transforms the data already in your data warehouse. Transformations are expressed in simple SQL SELECT statements and, when executed, dbt

compiles the code, infers dependency graphs, runs models in order, and writes the necessary DDL/DML to execute against your Snowflake instance. This makes it possible for users to focus on writing SQL and not worry about the rest. For writing code that is DRY (don't repeat yourself), users can use Jinja alongside SQL to express repeated logic using control structures such as loops and statements.

DBT CLOUD

dbt Cloud is the fastest and most reliable way to deploy dbt. It provides a centralized experience for teams to develop, test, schedule, and investigate data models—all in one web-based UI (see Figure 2). This is made possible through features such as an intuitive IDE, automated testing and documentation, in-app scheduling and alerting, access control, and a native Git integration.

dbt Cloud also eliminates the setup and maintenance work required to manage data transformations in Snowflake at scale. A turn-key adapter establishes a secure connection built to handle enterprise loads, while allowing for fine-grained policies and permissions.

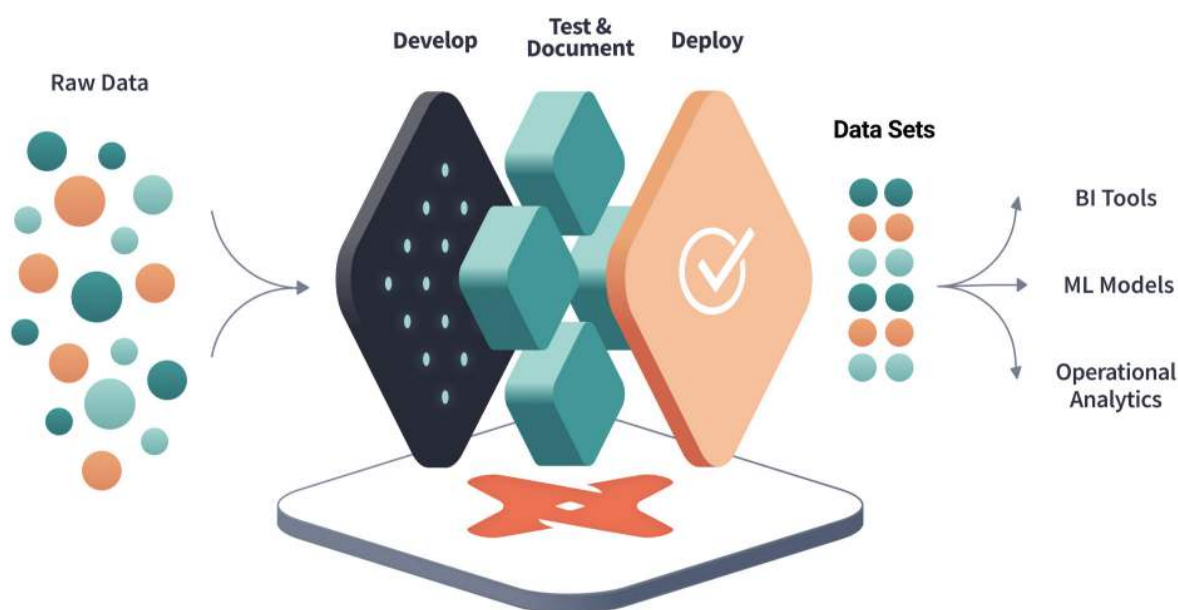


Figure 2: dbt Cloud provides a centralized experience for developing, testing, scheduling, and investigating data models.

CUSTOMER USE CASE

When Ben Singleton joined JetBlue as its Director of Data Science & Analytics, he stepped into a whirlpool of demands that his team struggled to keep up with. The data team was facing a barrage of concerns and low stakeholder trust.

“My welcome to JetBlue involved a group of senior leaders making it clear that they were frustrated with the current state of data,” Singleton said.

What made matters worse was the experts were not empowered to take ownership of their own data due to the inaccessibility of the data stack.


As Singleton dug, he realized the solution wasn't incremental performance improvement but rather a complete infrastructure overhaul. By pairing Snowflake with dbt, JetBlue was able to transform the data team from being a bottleneck to being the enablers of a data democracy.

“Every C-level executive wants more questions answered with data, they want that data faster, and they want it in many different ways. It's critical for us,” Singleton said. All of this was done without an increase in infrastructure costs. To read more about JetBlue's success story, see the JetBlue case study.¹

The remainder of this paper dives into the exact dbt and Snowflake best practices that JetBlue and thousands of other clients have implemented to optimize performance.

OPTIMIZING SNOWFLAKE

Your business logic is defined in dbt, but dbt ultimately pushes down all processing to Snowflake. For that reason, optimizing the Snowflake side of your deployment is critical to maximizing your query performance and minimizing deployment costs. The table on the following page summarizes the main areas and relevant best practices for Snowflake and serves as a checklist for your deployment.



**“Every C-level executive
wants more questions
answered with data, they want
that data faster, and they want
it in many different ways.
It's critical for us.”**

Ben Singleton
*Director of Data Science
& Analytics, JetBlue*

AREA	BEST PRACTICES	WHY
Automated resource optimization for dbt query tuning	Automatic clustering	Automated table maintenance
	Materialized views	Pre-compute complex logic
	Query acceleration services	Automated scale out part of query to speed up performance without resizing warehouse
Resource management and monitoring	Auto-suspend policies	Automatic stop of warehouse to reduce costs
	Resource monitors	Control of resource utilization and cost
	Naming conventions	Ease of tracking, allocation, and reporting
	Role-based access control	Governance and cost allocation
	Monitoring	Resource and cost consumption monitoring
Individual dbt workload elasticity	Scaling up for performance	Resizing warehouse to increase performance for complex workload
	Scaling out for concurrency	Spinning up additional warehouses to support a spike in concurrency
Writing effective SQL statements	Applying filters as early as possible	Optimizing row operations and reducing records in subsequent operations
	Querying only what you need	Selecting only the columns needed to optimize columnar store
	Joining on unique keys	Optimizing JOIN operations and avoiding cross-joins
	Avoiding complex functions and UDFs in WHERE clauses	Pruning

AUTOMATED RESOURCE OPTIMIZATION FOR DBT QUERY TUNING

Performance and scale are core to Snowflake. Snowflake's functionality is designed such that users can focus on core analytical tasks instead of on tuning the platform or investing in complicated workload management.

Automatic clustering

Traditionally, legacy on-premises and cloud data warehouses relied on static partitioning of large tables to achieve acceptable performance and enable better scaling. In these systems, a partition is a unit of management that is manipulated independently using specialized DDL and syntax; however, static partitioning has a number of well-known limitations, such as maintenance overhead and data skew, which can result in disproportionately sized partitions. It was the user's responsibility to constantly optimize the underlying data storage. This involved work such as updating indexes and statistics, post-load vacuuming procedures, choosing the right distribution keys, dealing with slow partitions due to growing skews, and manually reordering data as new data arrived or got modified.

In contrast to a data warehouse, Snowflake implements a powerful and unique form of partitioning called micro-partitioning, which delivers all the advantages of static partitioning without the known limitations, as well as providing additional significant benefits. Snowflake scalable, multi-cluster virtual warehouse technology automates the maintenance of micro-partitions. This means Snowflake efficiently and automatically executes the re-clustering in the background. There's no need to create, size, or resize a virtual warehouse. The compute service continuously monitors the clustering quality of all registered clustered tables. It starts with the most unclustered micro-partitions and iteratively performs the clustering until an optimal clustering depth is achieved.

With Snowflake, you can define clustered tables if the natural ingestion order is not sufficient in the presence of varying data access patterns. Automatic clustering is a Snowflake service that seamlessly and continually manages all reclustering, as needed, of clustered tables. Its benefits include the following:

- You no longer need to run manual operations to re-cluster data.
- Incremental clustering is done as new data arrives or a large amount of data is modified.
- Data pipelines consisting of DML operations (INSERT, DELETE, UPDATE, MERGE) can run concurrently and are not blocked.
- Snowflake performs automatic reclustering in the background, and you do not need to specify a warehouse to use.
- You can resume and suspend automatic clustering on a per-table basis, and you are billed by the second for only the compute resources used.
- Snowflake internally manages the state of clustered tables, as well as the resources (servers, memory, and so on) used for all automated clustering operations. This allows Snowflake to dynamically allocate resources as needed, resulting in the most efficient and effective reclustering. The Automatic Clustering service does not perform any unnecessary reclustering. Reclustering is triggered only when a table would benefit from the operation.

dbt supports table clustering on Snowflake. To control clustering for a table or incremental model, use the `cluster_by` configuration. Refer to the Snowflake configuration guide for more details.

Materialized views

A [materialized view](#) is a pre-computed data set derived from a query specification (the SELECT in the view definition) and stored for later use. Because the data is pre-computed, querying a materialized view (MV) is faster than executing a query against the base table of the view. This performance difference can be significant when a query is run frequently or is sufficiently complex. As a result, MVs can speed up expensive aggregation, projection, and selection operations, especially those that run frequently and that run on large data sets. dbt does not support MVs out of the box as materializations; therefore, we recommend using custom materializations as a solution to achieve similar purposes. The [dbt materializations](#) section in this white paper explains how MVs can be used in dbt via a custom materialization.

MVs are particularly useful when:

- Query results contain a small number of rows and/or columns relative to the base table (the table on which the view is defined)
- Query results contain results that require significant processing, including:
 - Analysis of semi-structured data
 - Aggregates that take a long time to calculate
- The query is on an external table (that is, data sets stored in files in an external stage), which might have slower performance compared to querying native database tables
- The view's base table does not change frequently

In general, when deciding whether to create an MV or a regular view, use the following criteria:

- Create an MV when all of the following are true:
 - The query results from the view don't change often. This almost always means that the underlying/base table for the view doesn't change often or at least the subset of base table rows used in the MV doesn't change often.
 - The results of the view are used often (typically, significantly more often than the query results change).
 - The query consumes a lot of resources. Typically, this means that the query consumes a lot of processing time or credits, but it could also mean that the query consumes a lot of storage space for intermediate results.
- Create a regular view when any of the following are true:
 - The results of the view change often.
 - The results are not used often (relative to the rate at which the results change).
 - The query is not resource-intensive so it is not costly to re-run it.

These criteria are just guidelines. An MV might provide benefits even if it is not used often—especially if the results change less frequently than the usage of the view.

There are also other factors to consider when deciding whether to use a regular view or an MV. One such example is the cost of storing and maintaining the MV. If the results are not used very often (even if they are used more often than they change), the additional storage and compute resource costs might not be worth the performance gain.

Snowflake's compute service monitors the base tables for MVs and kicks off refresh statements for the corresponding MVs if significant changes are detected. This maintenance process of all dependent MVs is asynchronous. In scenarios where a user is accessing an MV that has yet to be updated, Snowflake's query engine will perform a combined execution with the base table to always ensure consistent query results. Similar to Snowflake's automatic clustering with the ability to resume or suspend per table, a user can resume and suspend the automatic maintenance on a per-MV basis. The automatic refresh process consumes resources and can result in increased credit usage. However, Snowflake ensures efficient credit usage by billing your account only for the actual resources used. Billing is calculated in one-second increments.

You can control the cost of maintaining MVs by carefully choosing how many views to create, which tables to create them on, and each view's definition (including the number of rows and columns in that view).

You can also control costs by suspending or resuming a MV; however, suspending maintenance typically only defers costs rather than reducing them. The longer that maintenance has been deferred, the more maintenance there is to do.

If you are concerned about the cost of maintaining MVs, we recommend you start slowly with this feature (that is, create only a few MVs on selected tables) and monitor the costs over time.

It's a good idea to carefully evaluate these guidelines based on your dbt deployment to see if querying from MVs will boost performance compared to base tables or regular views without cost overhead.

Query acceleration services

Sizing the warehouse just right for a workload is generally a hard trade-off between minimizing cost and maximizing query performance. You'll usually have to monitor, measure, and pick an acceptable point in this price-performance spectrum and readjust as required. Workloads that are unpredictable in terms of either the number of concurrent queries or the amount of data required for a given query make this challenging.

Multi-cluster warehouses handle the first case well and scale out only when there are enough queries to justify it. For the case where there is an unpredictable amount of data in the queries, you usually have to either wait longer for queries that look at larger data sets or resize the entire warehouse, which affects all clusters in the warehouse and the entire workload.

Snowflake's Query Acceleration Service provides a good default for the price-performance spectrum by automatically identifying and scaling out parts of the query plan that are easily parallelizable (for example, per-file operations such as filters, aggregations, scans, and join probes using bloom filters). The benefit is a much reduced query runtime at a lower cost than would result from just using a larger warehouse.

The Query Acceleration Service achieves this by elastically recruiting ephemeral worker nodes to lend a helping hand to the warehouse. Parallelizable fragments of the query plan are queued up for processing on leased workers, and the output of this fragment execution is materialized and consumed by the warehouse workers as a stream. As a result, a query over a large data set can finish faster, use fewer resources on the warehouse, and potentially, cost fewer total credits than it would with the current model.

What makes this feature unique is:

- It supports filter types, including joins
- No specialized hardware is required
- You can enable, disable, or configure the service without disrupting your workload

This is a great feature to use in your dbt deployment if you are looking to:

- Accelerate long-running dbt queries that scan a lot of data
- Reduce the impact of scan-heavy outliers
- Scale performance beyond the largest warehouse size
- Speed up performance without changing the warehouse size

Please note that this feature is currently managed outside of dbt.

This feature is in private preview at the time of this white paper's first publication; please reach out to your Snowflake representative if you are interested in experiencing this feature with your dbt deployment.

RESOURCE MANAGEMENT AND MONITORING

A virtual warehouse consumes Snowflake credits while it runs, and the amount consumed depends on the size of the warehouse and how long it runs. Snowflake provides a rich set of resource management and monitoring capabilities to help control costs and avoid unexpected credit usage, not just for dbt transformation jobs but for all workloads.

Auto-suspend policies

The very first resource control that you should implement is setting auto-suspend policies for each of your warehouses. This feature automatically stops warehouses after they've been idle for a predetermined amount of time.

We recommend setting auto-suspend according to your workload and your requirements for warehouse availability:

- If you enable auto-suspend for your dbt workload, we recommend setting a more aggressive policy with the standard recommendation being 60 seconds, because there is little benefit from caching.
- You might want to consider disabling auto-suspend for a warehouse if:
 - You have a heavy, steady workload for the warehouse.
 - You require the warehouse to be available with no delay or lag time. While warehouse provisioning is generally very fast (for example, 1 or 2 seconds), it's not entirely instant; depending on the size of the warehouse and the availability of compute resources to provision, it can take longer.

If you do choose to disable auto-suspend, you should carefully consider the costs associated with running a warehouse continually even when the warehouse is not processing queries. The costs can be significant, especially for larger warehouses (X-Large, 2X-Large, or larger.).

We recommend that you customize auto-suspend thresholds for warehouses assigned to different workloads to assist in warehouse responsiveness:

- Warehouses used for queries that benefit from caching should have a longer auto-suspend period to allow for the reuse of results in the query cache.
- Warehouses used for data loading can be suspended immediately after queries are completed. Enabling auto-resume will restart a virtual warehouse as soon as it receives a query.

Resource monitors

Resource monitors can be used by account administrators to impose limits on the number of credits that are consumed by different workloads, including dbt jobs within each monthly billing period, by:

- User-managed virtual warehouses
- Virtual warehouses used by cloud services

When these limits are either close to being reached or have been reached, the resource monitor can send alert notifications or suspend the warehouses.

It is essential to be aware of the following rules about resource monitors:

- A monitor can be assigned to one or more warehouses.
- Each warehouse can be assigned to only one resource monitor.
- A monitor can be set at the account level to control credit usage for all warehouses in your account.
- An account-level resource monitor does not override resource monitor assignment for individual warehouses.

- If either the warehouse-level or account-level resource monitor reaches its defined threshold, the warehouse is suspended. This enables controlling global credit usage while also providing fine-grained control over credit usage for individual or specific warehouses.
- In addition, an account-level resource monitor does not control credit usage by the Snowflake-provided warehouses (used for Snowpipe, automatic reclustering, and MVs); the monitor controls only the virtual warehouses created in your account.

Considering these rules, the following are some recommendations on resource monitoring strategy:

- Define an account-level budget.
- Define priority warehouse(s) including warehouses for dbt workloads and carve from the master budget for priority warehouses.
- Create a resource allocation story and map.

Figure 3 illustrates an example scenario for a resource monitoring strategy in which one resource monitor is set at the account level, and individual warehouses are assigned to two other resource monitors:

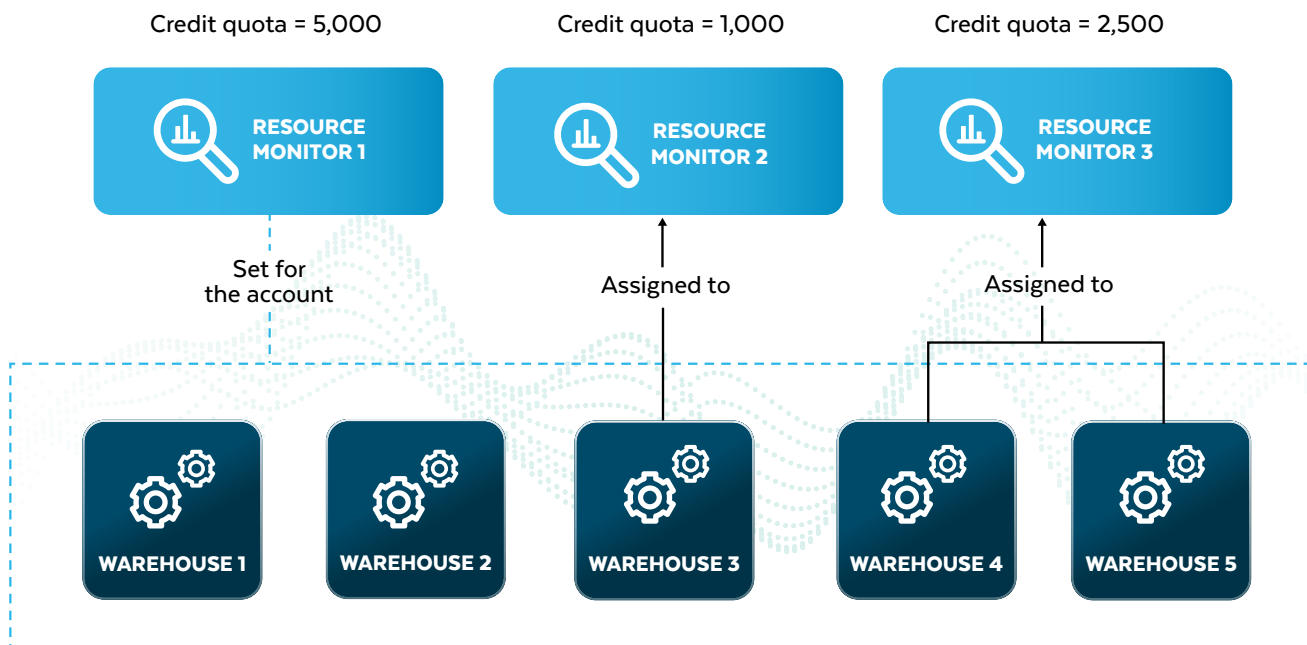


Figure 3: Example scenario for a resource monitoring strategy

In the example (Figure 3 on the previous page), the credit quota for the entire account is 5,000 per month; if this quota is reached within the interval, the actions defined for the resource monitor (Suspend, Suspend Immediate, and so on) are enforced for all five warehouses.

Warehouse 3 performs ETL including ETL for dbt jobs. From historical ETL loads, we estimated it can consume a maximum of 1,000 credits for the month. We assigned this warehouse to Resource Monitor 2.

Warehouse 4 and 5 are dedicated to the business intelligence and data science teams. Based on their historical usage, we estimated they can consume a maximum combined total of 2,500 credits for the month. We assigned these warehouses to Resource Monitor 3.

Warehouse 1 and 2 are for development and testing. Based on historical usage, we don't need to place a specific resource monitor for them.

The credits consumed by Warehouses 3, 4, and 5 may be less than their quotas if the account-level quota is reached first.

The used credits for a resource monitor reflect the sum of all credits consumed by all assigned warehouses within the specified interval. If a monitor has a Suspend or Suspend Immediately action defined and its used credits reach the threshold for the action, any warehouses assigned to the monitor are suspended and cannot be resumed until one of the following conditions is met:

- The next interval, if any, starts, as dictated by the start date for the monitor.
- The credit quota for the monitor is increased.
- The credit threshold for the suspended action is increased.
- The warehouses are no longer assigned to the monitor.
- The monitor is dropped.

Resource monitors are not intended for strictly controlling consumption on an hourly basis; they are intended for tracking and controlling credit consumption per interval (day, week, month, and so on). Also, they are not intended for setting precise limits on credit usage (that is, down to the level of individual credits). For example, when credit quota thresholds are reached for a resource monitor, the assigned warehouses may take some

time to suspend, even when the action is Suspend Immediate, thereby consuming additional credits.

If you wish to strictly enforce your quotas, we recommend the following:

- Utilize buffers in the quota thresholds for actions (for example, set a threshold to 90% instead of 100%). This will help ensure that your credit usage doesn't exceed the quota.
- To more strictly control credit usage for individual warehouses, assign only a single warehouse to each resource monitor. When multiple warehouses are assigned to the same resource monitor, they share the same quota thresholds, which may result in credit usage for one warehouse impacting the other assigned warehouses.

When a resource monitor reaches the threshold for an action, it generates one of the following notifications, based on the action performed:

- The assigned warehouses will be suspended after all running queries complete.
- All running queries in the assigned warehouses will be canceled and the warehouses suspended immediately.
- A threshold has been reached, but no action has been performed.

Notifications are disabled by default and can be received only by account administrators with the ACCOUNTADMIN role. To receive notifications, each account administrator must explicitly enable notifications through their preferences in the web interface. In addition, if an account administrator chooses to receive email notifications, they must provide (and verify) a valid email address before they will receive any emails.

We recommend having well-defined naming conventions to separate warehouses between hub and spokes for tracking, governance (RBAC), and resource monitors for consumption alerts.

Naming conventions

Your resource monitor naming conventions are a foundation for tracking, allocation, and reporting. They should follow an enterprise plan for the domain (that is, function/market + environment). They should also align to your virtual warehouse naming convention when more granularity is needed.

The following is a sample naming convention:

<domain>_<team>_<function>_<base_name>

<team>: The name of the team (for example, engineering, analytics, data science, service, and so on) that the warehouses being monitored have been allocated to. When used, this should be the same as the team name used within the names of the warehouses.

<function>: The processing function (for example, development, ELT, reporting, ad hoc, and so on) generally being performed by the warehouses to be monitored. When used, this should be the same as the processing function name used within the names of the warehouses.

<base name>: A general-purpose name segment to further distinguish one resource monitor from another. When used, this may be aligned with the base names used within the names of the warehouses or it may be something more generic to represent the group of warehouses.

An example of applying the naming conventions above might look something like this:

USA_WAREHOUSES: A resource monitor set to monitor and send alerts for all warehouses allocated to the USA spoke.

USA_PRD_DATASCIENCE_ADHOC: A resource monitor set to monitor and send alerts for just the single production data science warehouse for the USA.

USA_PRD_SERVICE_WAREHOUSES: A resource monitor set to monitor and send alerts for all production services (for example, ELT, reporting tools, and so on) warehouses for the USA.

Role-based access control (RBAC)

Team members have access only to their assigned database and virtual warehouse resources to ensure accurate cost allocation.

Monitoring

An important first step to managing credit consumption is to monitor it. Snowflake offers several capabilities to closely monitor resource consumption.

The first such resource is the Admin Billing and Usage page in the web interface, which offers a breakdown of consumption by day and hour for individual warehouses as well as for cloud services. This data can be downloaded for further analysis. Figure 4 through Figure 6 show example credit, storage, and data transfers consumption from the Snowsight dashboard.

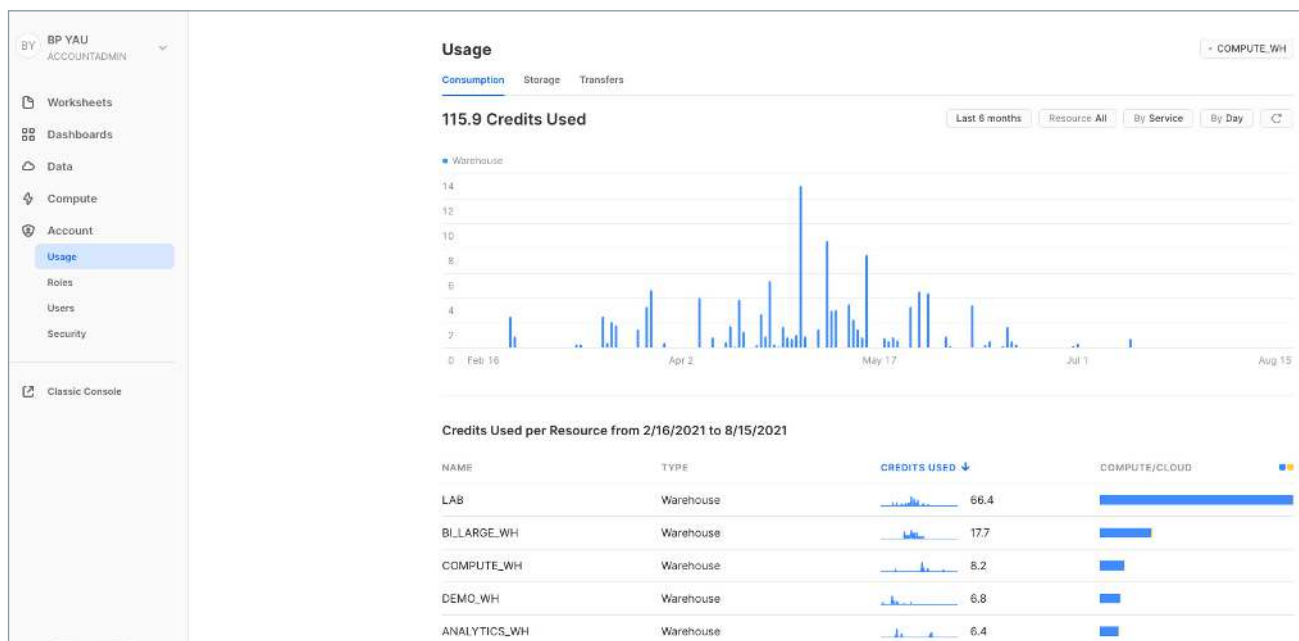


Figure 4: Example credit consumption from the Snowsight dashboard

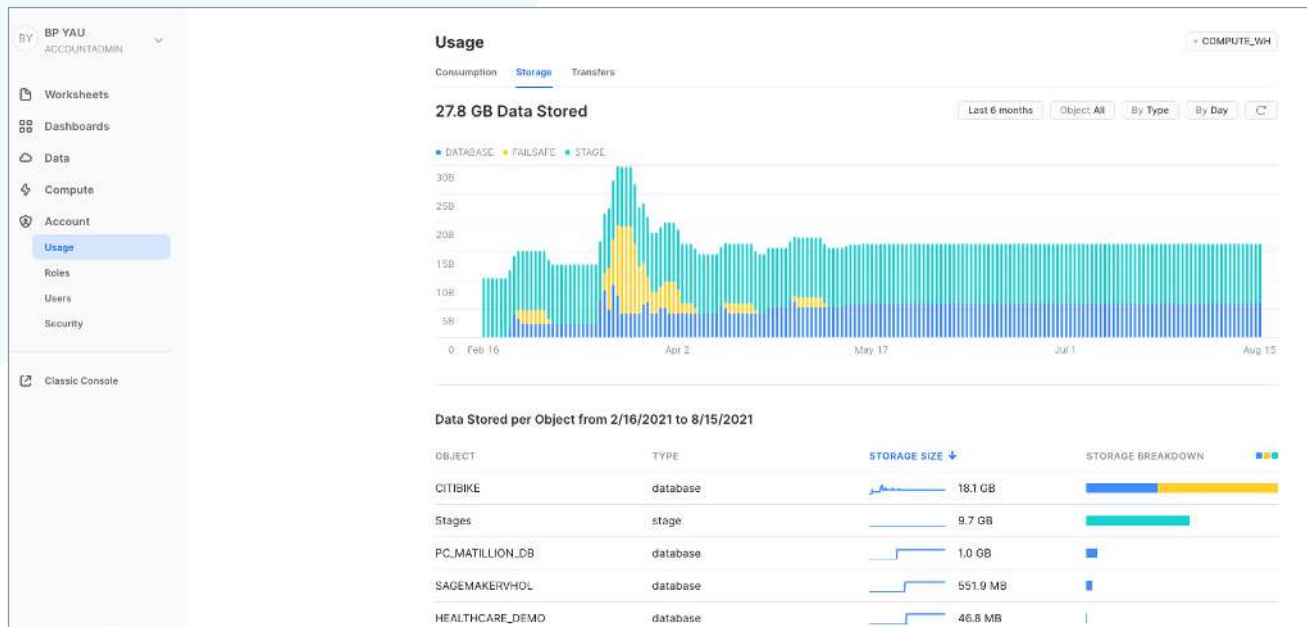


Figure 5: Example storage consumption from the Snowsight dashboard

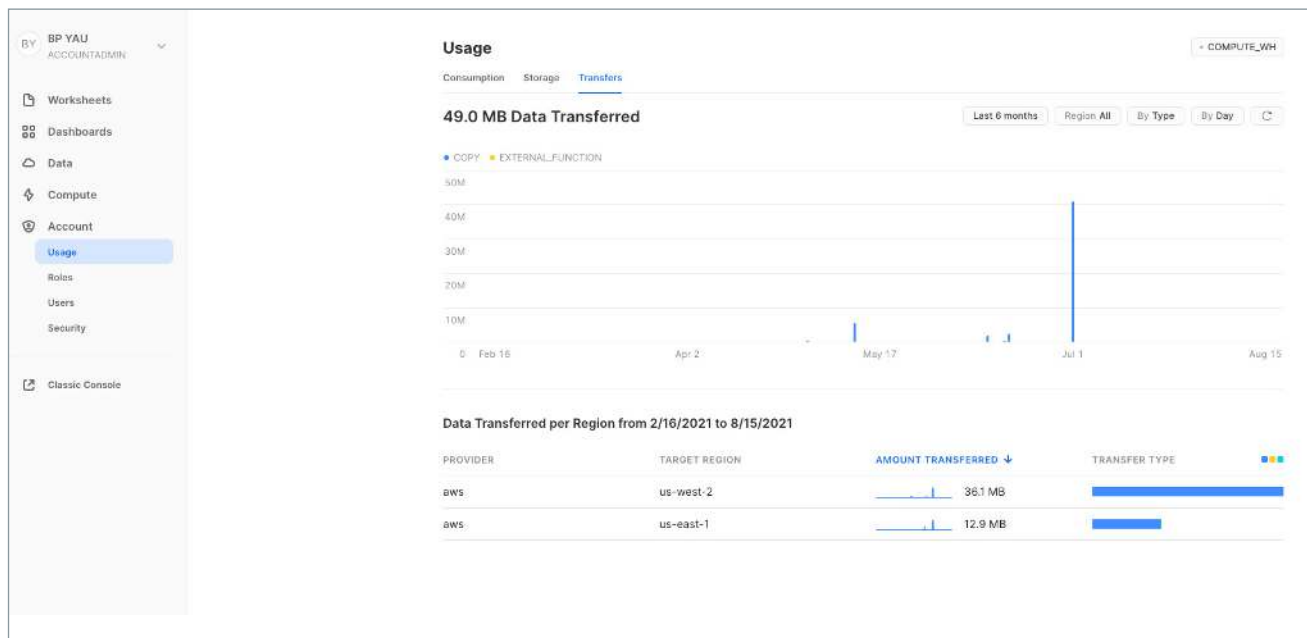


Figure 6: Example data transfers consumption from the Snowsight dashboard

For admins who are interested in diving even deeper into resource optimization, Snowflake provides the account usage and information schemas. These tables offer granular details on every aspect of account usage, including for roles, sessions, users, individual queries, and even the performance or “load” on each virtual warehouse.

This historical data can be used to build advanced forecasting models to predict future credit consumption. This trove of data is especially important to customers who have complex multiaccount organizations.

Monitoring credit usage

VIEW	DESCRIPTION
METERING_DAILY_HISTORY	Daily credit usage and rebates across all service types within the last year
WAREHOUSE_METERING_HISTORY	Hourly credit usage per warehouse within the last year
QUERY_HISTORY	A record of every query (including SQL text), elapsed and compute time, and key statistics

Monitoring storage usage

VIEW	DESCRIPTION
DATABASE_STORAGE_USAGE_HISTORY	Average daily usage (bytes) by database
TABLE_STORAGE_METRICS	Detailed storage records for tables

The account usage and information schemas can be queried directly using SQL or analyzed and charted using Snowsight. The example provided below is of a [load monitoring chart](#). To view the chart, click

Warehouses in the web interface. As shown in Figure 7, the Warehouse Load Over Time page provides a bar chart and a slider for selecting the window of time to view in the chart.

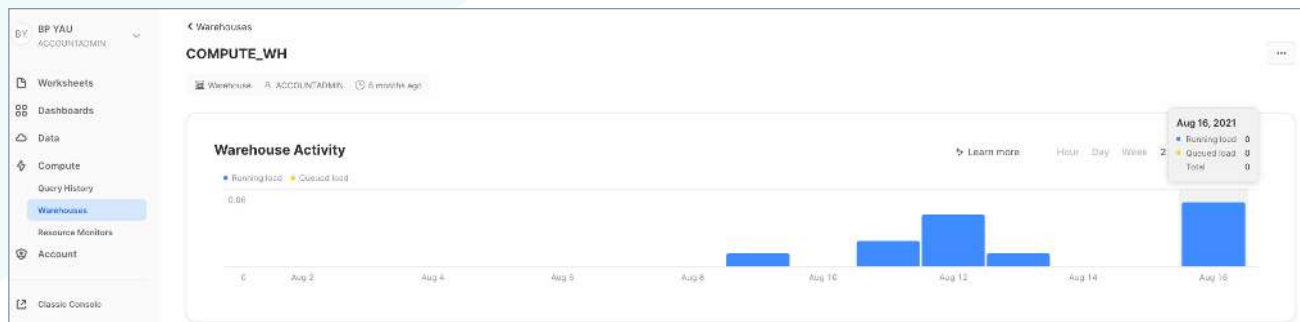


Figure 7: Warehouse Load Over Time page

dbt offers a package called the Snowflake spend package that can be used to monitor Snowflake usage. Refer to the [dbt package section](#) of this white paper for more details.

Many third-party BI vendors offer pre-built dashboards that can be used to automatically visualize this data,

including the ability to forecast future usage. We recommend sharing the account usage dashboards offered by your customers' preferred BI vendors to help them gain visibility on their Snowflake usage and easily forecast future usage. Figure 8 shows an example from Tableau.²

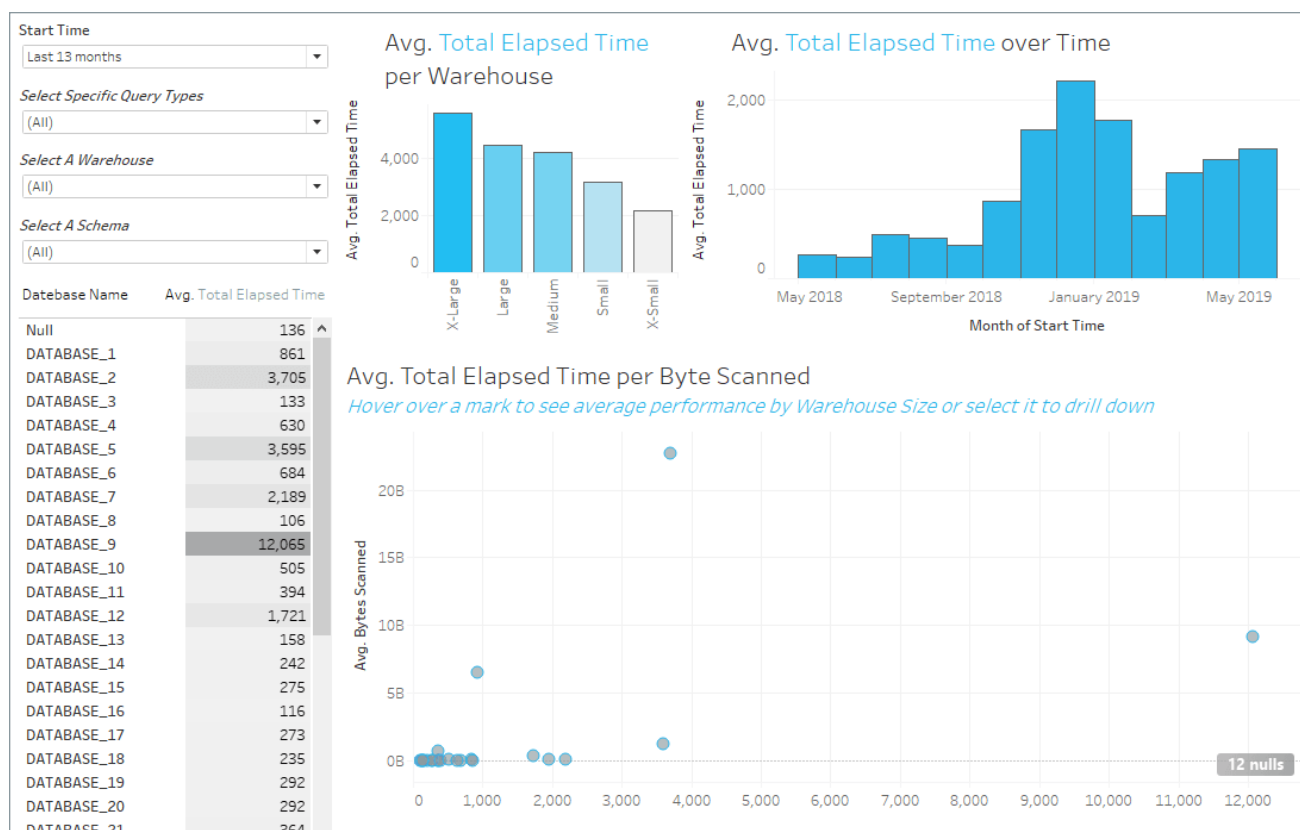


Figure 8: Tableau dashboard for monitoring performance

INDIVIDUAL DBT WORKLOAD ELASTICITY

Snowflake supports two ways to scale warehouses:

- Scale up by resizing a warehouse.
- Scale out by adding warehouses to a multi-cluster warehouse (requires [Snowflake Enterprise Edition](#) or higher).

Resizing a warehouse generally improves query performance, particularly for larger, more complex queries. It can also help reduce the queuing that occurs if a warehouse does not have enough compute resources to process all the queries that are submitted concurrently. Note that warehouse resizing is not intended for handling concurrency issues. Instead, in such cases, we recommend you use additional warehouses or use a multi-cluster warehouse (if this feature is available for your account).

Snowflake supports resizing a warehouse at any time, even while running. If a query is running slowly and you have additional queries of similar size and complexity that you want to run on the same warehouse, you might choose to resize the warehouse while it is running; however, note the following:

- As stated earlier, larger is not necessarily faster; for smaller, basic queries that are already executing quickly, you may not see any significant improvement after resizing.
- Resizing a running warehouse does not impact queries that are already being processed by the warehouse; the additional compute resources, once fully provisioned, are used only for queued and new queries.
- Resizing between a 5XL or 6XL warehouse to a 4XL or smaller warehouse will result in a brief period during which you are charged for both the new warehouse and the old warehouse while the old warehouse is quiesced.

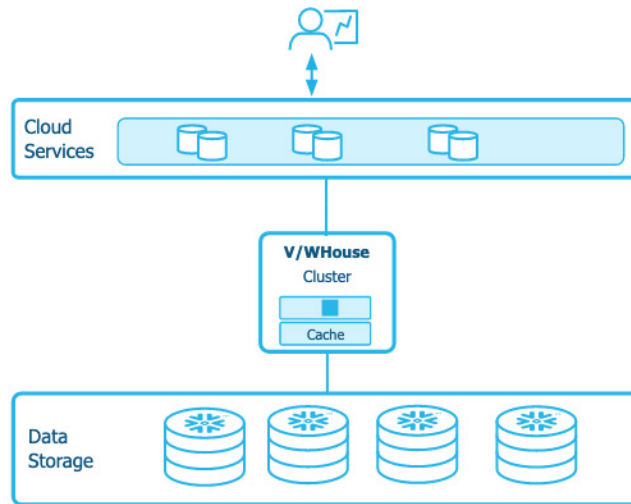


Figure 9: User running an X-Small virtual warehouse

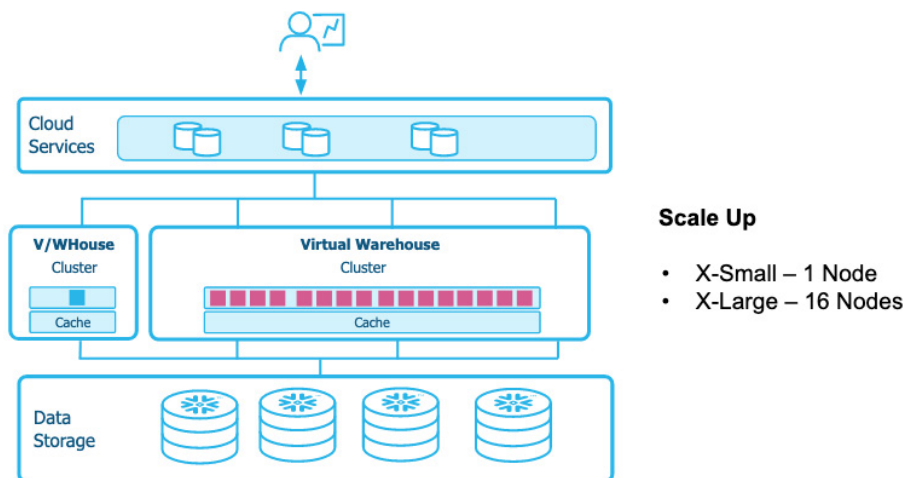


Figure 10: User resizes the warehouse to X-Large

Scaling up for performance

The purpose of scaling up is to improve query performance and save cost. Let's look at an example to illustrate this.

A user running an X-Small virtual warehouse is illustrated in Figure 9. The user executes an `ALTER WAREHOUSE` statement to resize the warehouse to X-Large, as shown in Figure 10.

As a result, the number of nodes increases from 1 to 16. During the resize operation, any currently running queries are allowed to complete and any queued or

subsequent queries are started on the newly allocated virtual warehouse.

Note that if you start a massive task and amend the warehouse size while the query is executing, it will continue to execute on the original warehouse size. This means you may need to kill and restart a large running task to gain benefits of the larger warehouse.

Also note that it is not possible to automatically adjust warehouse size. However, you could script the `ALTER WAREHOUSE` statement to automate the process as part of a batch ETL operation, for example.

Let's now look at some benchmark data. Below is a simple query, similar to many ETL queries in practice, to load 1.3 TB of data. It was executed on various warehouse sizes.

```
create table terabyte_sized_copy as
select *
from sample_data.tpcds_sf10tcl.store_sales;
```

The table below shows the elapsed time and cost for different warehouses.

T-SHIRT SIZE	ELAPSED TIME	COST (CREDITS)
X-Small	5 hours and 30 minutes	5.5
Small	1 hour and 53 minutes	3.7
Medium	1 hour and zero minutes	4.0
Large	37 minutes and 57 seconds	5.0
XLarge	16 minutes and 7 seconds	4.2
2X-Large	7 minutes and 41 seconds	4.0
3X-Large	4 minutes and 52 seconds	5.1
4X-Large	2 minutes and 32 seconds	5.4
Improvement	132 X	Same

Here are some interesting observations from the table above:

- For a large operation, as the warehouse size increases, the elapsed time drops by approximately half.
- Each step up in warehouse size doubles the cost per hour.
- However, since the warehouse can be suspended after the task is completed, the actual cost of each operation is approximately the same.
- Going from X-Small to 4X-Large yields a 132x performance improvement with the same cost. This clearly illustrates how and why scaling up helps to improve performance and save cost.
- Look at how compute resources can be dynamically scaled up, down, or out for each individual workload based on demand, and also suspend automatically to stop incurring cost, which is based on per-second billing.
- New 5XL and 6XL virtual warehouse sizes are now available on AWS and in public preview on Azure at the time of this white paper's first publication. These sizes give users the ability to add more compute power to their workloads and enable faster data loading,

transformations, and querying. Previously, customers who needed to support compute-intensive workloads for data processing had to do batch processing and use multiple 4XL warehouses to accomplish their tasks. The new 5XL and 6XL virtual warehouse sizes give users the ability to run larger compute-intensive workloads in a performant fashion without any batching.

For a dbt workload, you should be strategic about what warehouse size you use. By default, dbt will use the warehouse declared in the connection. If you want to adjust the warehouse size, you can either [declare a static warehouse configuration on the model or project level](#) or as a dynamic macro such as the [one shared in the Snowflake_utils package](#).

This allows you to automate selection of the warehouse used for your models without manually updating your connection. Our recommendation is to use a larger warehouse for incremental full-refresh runs where you are rebuilding a large table from scratch.

Scaling out for concurrency

Multi-cluster warehouses are best utilized for scaling resources to improve concurrency for users and queries. They are not as beneficial for improving the performance of slow-running queries or data loading; for those types of operations, resizing the warehouse provides more benefits.

Figure 11 illustrates a customer running queries against an X-Small warehouse. The performance is satisfactory, but in this example, the customer

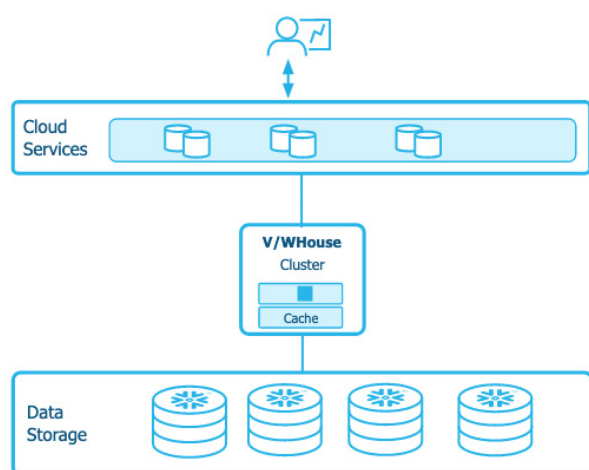


Figure 11: Customer runs queries against an X-Small warehouse

The system will automatically scale out by adding additional clusters of the same size as additional concurrent users run queries. The system also will automatically scale back as demand is reduced. As a result, the customer pays only for resources that were active during the period.

In cases where a large load is anticipated from a pipeline or from usage patterns, the `min_cluster` parameter can be set beforehand to bring all compute resources online. This will reduce the delays in bringing compute online, which usually happens only after query queuing and only gradually with a cluster every 20 seconds.

anticipates that there will soon be a dramatic change in the number of users online. In Figure 12, the customer executes an `ALTER WAREHOUSE` command to enable the multi-cluster warehouse feature. This command might look like:

```
alter warehouse PROD_VWH set  
    min_cluster_count = 1  
    max_cluster_count = 10;
```

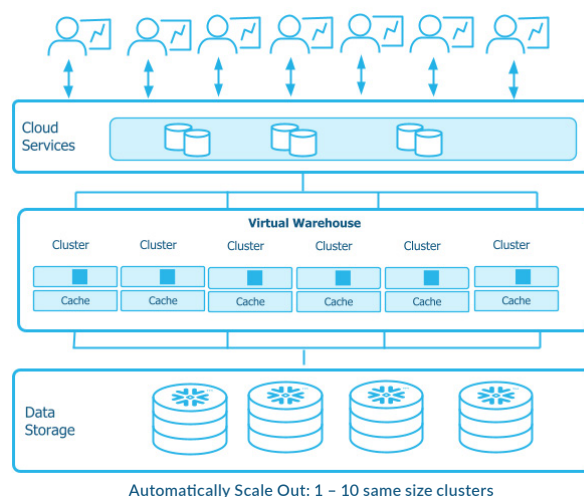


Figure 12: Customer executes an `ALTER WAREHOUSE` command to enable the multi-cluster warehouse feature

WRITING EFFECTIVE SQL STATEMENTS

To optimize performance, it's crucial to write effective SQL queries in dbt for execution on Snowflake.

Query order of execution

A query is often written in this order:

```
SELECT  
FROM  
JOIN  
WHERE  
GROUP BY  
ORDER BY  
LIMIT
```



Figure 13: The order of query execution

The order of execution for this query in Snowflake is shown in Figure 13 above. Accordingly, the example above would execute in the following order:

Step 1: FROM clause (cross-product and join operators)

Step 2: WHERE clause (row conditions)

Step 3: GROUP BY clause (sort on grouping columns, compute aggregates)

Step 4: HAVING clause (group conditions)

Step 5: ORDER BY clause

Step 6: Columns not in SELECT eliminated (projection operation)

SQL first checks which data table it will work with, and then it checks the filters, after which it groups the data. Finally it retrieves the data—and, if necessary, sorts it and prints only the first <X> lines.

Applying filters as early as possible

As you can see from the order of execution, ROW operations are performed before GROUP operations. Thus, it's important to think about optimizing ROW operations before GROUP operations in your query. It's recommended to apply filters early at the WHERE-clause level.

Querying only what you need

Snowflake uses a columnar format to store data, so the number of columns retrieved from a query matters a great deal for performance. Best practice is to **select only the columns you need**. You should:

- Avoid using **SELECT *** to return all columns
- Avoid queries with **SELECT** long string columns or **SELECT** entire variant column

Joining on unique keys

Joining on nonunique keys can make your data output explode in magnitude, for example, where each row in table1 matches multiple rows in table2. Figure 14 shows an example execution plan where this happens, wherein the JOIN operation is the most costly operation.

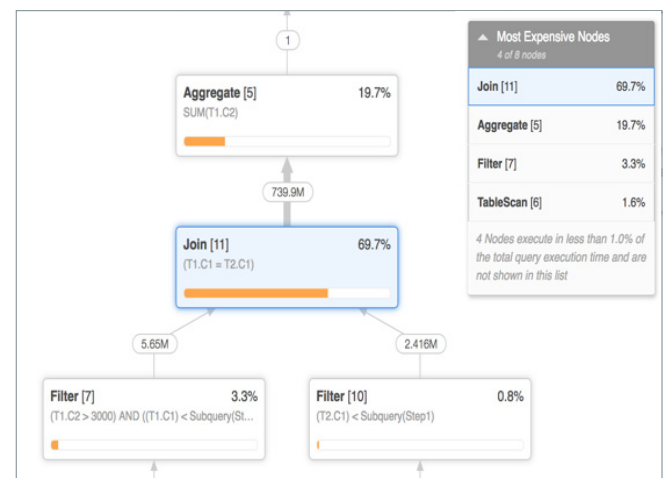


Figure 14: Example execution plan in which the JOIN is costly

Best practices for JOIN operations are:

- Ensuring keys are distinct (deduplicate)
- Understanding the relationships between your tables before joining
- Avoiding many-to-many joins
- Avoiding unintentional cross-joins

Avoiding complex functions and UDFs in WHERE clauses

While built-in functions and UDFs can be tremendously useful, they can also impact performance when used in query predicates. Figure 15 is an example of this scenario in which a log function is used where it should not be used.

```
select l_orderkey
from lineitem l, orders o
where l_orderkey=o_orderkey and
      log(10, l_extendedprice) > 4.5 and
      log(10, o_totalprice - l_tax) > 4.5
```

Figure 15: Example of using a log function inappropriately

OPTIMIZING DBT

dbt is a transformation workflow that lets analytics engineers transform data by simply writing SQL statements. At its core, the way it operates with Snowflake is by compiling the SQL for Snowflake to execute. This means you can perform all of your data transformations inside of your data warehouse, making your process more efficient because there's no need for data transference. You also get full access to Snowflake's extensive analytics functionalities, now framed by the dbt workflow. In this section, we discuss specific dbt best practices that optimize Snowflake resources and functionalities. For broader dbt best practices, [check out this discourse post](#).

Use environments

Mitigate risk by defining environments in Snowflake and dbt. Making use of distinct environments may not be new in the world of software engineering but definitely can be in the world of data. The primary benefit of using clearly defined production and development environments is the mitigation of risk: in particular, the risk of costly rebuilds if anything breaks in production. With dbt and Snowflake, you can define cohesive environments and operate in them with minimal friction. Before even beginning development work in dbt, you should create and strictly implement these environments.

On the Snowflake layer, the account should be set up with minimal separation of raw and analytics databases, as well as with clearly defined production and development schemas. There are different iterations of this setup, and you should create what meets the needs of your workflow. The goal here is to remove any confusion as to where objects should be built during the different stages of development and deployment.

dbt developers should have control of their own development sandboxes so they can safely build any objects they have permissions to build. A sandbox often takes the form of a personal schema to ensure that other developers don't accidentally delete or update objects. To learn more, [check out this blog post](#).

On the dbt layer, environment definitions consist of two things: the connection details and a dbt concept called [target](#).

When setting up your connection, you provide a data warehouse and schema. Those will be the default Snowflake schema and database you will be building objects into.

Meanwhile, how your target comes into play differs slightly depending on the dbt interface you are using. If you're using the command line, the target is the connection you wish to connect to (and thus the default schema/database). You can also use the target to apply Jinja conditions in your code, allowing you to adjust the compiled code based on the target. If you're using dbt Cloud, the target can be used only to apply conditional logic; the default schema/database will be defined in the environment settings.

As a best practice for development, the default schema should be your sandbox schema, while for production, the default should be a production schema. As a project grows in size, you should define [custom databases/schemas](#) either via hard-coding or via dynamic logic using [targets](#) so that, depending on the environment you're working in, the database/schema changes to the associated Snowflake environment.

When you combine environments with the [ref function](#), code promotion is dramatically simplified. The [ref](#) function dynamically changes the object being

referenced based on the environment, without you having to write conditional logic. This means that when you select from a referenced object, dbt will automatically know the appropriate schema and/or database to interpolate.

This makes it possible for your code to **never have to change** as it's promoted from development to production because dbt is always aware of the underlying environment. Figure 16 (below) shows an example of how a dbt model relates to a Snowflake database. You can configure the dbt model `df_model` to explicitly build into the Snowflake database `df_{environment}` every time or based on conditional logic.

In addition to creating clearly defined environments, there is an additional cost (and time) saving measure that `target` makes possible. During development, you may find that you often need only a subset of your data set to test and iterate over. A good way to limit your data set in this way is to implement [conditional logic to limit data](#) in `dev`.

Such macros can automate when a filter is applied and ensure only a limited amount of data is run. This allows you to do away with the hassle of remembering to apply and remove data limitations through environments.

To more systematically apply this through a project, a good practice is to put the conditional logic into a macro and then call the macro across models. This

allows updating the logic only in one place. You can also implement [a variable](#) in the logic to adjust the time period specified in the WHERE clause (with a default date that can be overridden in a run).

Here is sample code that allows you to call this macro into a dbt model and add the WHERE clause when the target is dev:

```
{% macro limit_in_dev(timestamp) %}

-- this filter will only apply during a dev run

{% if target.name == 'dev' %}

    where {{timestamp}} > dateadd('day',
-{{var('development_days_of_data')}} , current_date)

{% endif %}
```

For larger projects, you can also use macros to limit rebuilding existing objects. By operationalizing the [Snowflake Zero-Copy Cloning](#) feature, you can ensure that your environments are synced up by cloning from another environment to stay up to date. This is fantastic for developers who prefer to simply clone from an existing development schema or from the production schema to have all the necessary objects to run the entire project and update only what is necessary. [By putting this macro into your project](#), you ensure that developers are writing the correct DDL every time because all they have to do is execute it rather than manually write it every time.

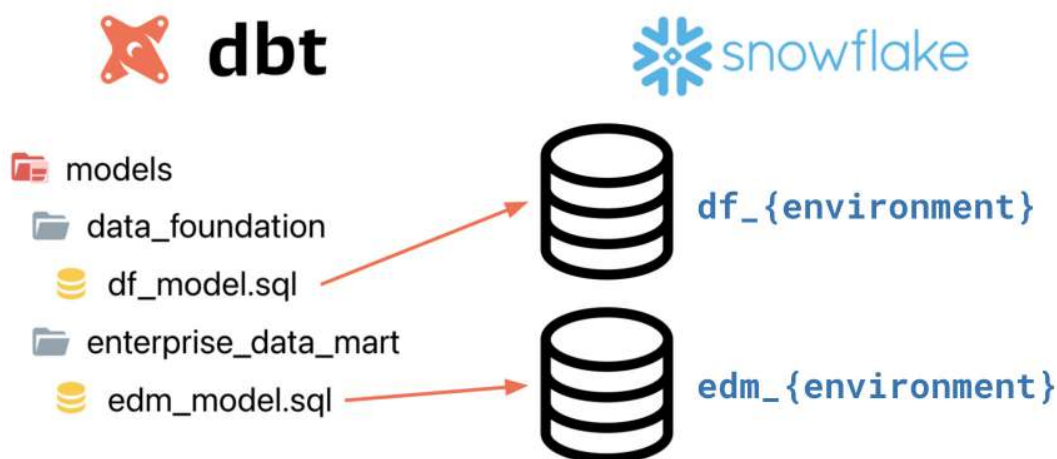


Figure 16: Example of how a dbt model relates to a Snowflake database

USE THE `ref()` FUNCTION AND SOURCES

Always use the `ref` function and sources in combination with threads.

To best leverage Snowflake's resources, it's important to carefully consider the design of your dbt project. One key way to do that is to ensure you are using the `ref()` and `source()` functions in every dbt model, rather than hard-coding database objects.

The `ref` function is a keystone of dbt's functionality. By using the function, dbt is able to infer dependencies and ensure that the correct upstream tables and views are selected based on your environment. Simply put, it makes sense to **always use the `ref` function when selecting from another model**, rather than using the direct relation reference (for example, `my_schema.my_table`).

When you use the `ref` function, dbt automatically establishes a lineage from the model being referenced to the model where that reference is declared, and then it uses it to optimize the build order and document lineage.

After the `ref()` function creates the directed acyclic graph (DAG), dbt is able to optimally execute models based on the DAG and the number of `threads` or maximum number of paths through the graph dbt is allowed to work on. As you increase the number of threads, dbt increases the number of paths in the graph that it can work on at the same time, thus reducing the runtime of your project.

Our recommendation is to start with **eight threads** (meaning up to eight parallel models that do not violate dependencies can be run at the same time), and then increase the number of threads as your project expands. While there is no maximum number of threads you can declare, it's important to note that increasing the number of threads increases the load on your warehouse, potentially constraining other usage.

The number of concurrent models being run is also a factor of your project's dependencies. For that reason, we recommend structuring your code as multiple models, maximizing the number that can be run simultaneously.

As your project expands, you should continue to increase the number of threads while keeping an eye on your Snowflake compute. Hitting compute limitations as you increase the number of threads may be a good signal that it's time to increase the Snowflake warehouse size as well.

Figure 17 shows a sample dbt DAG. In this example, if a user declared three threads, dbt would know to run the first three staging models prior to running `dim_suppliers`. By specifying three threads, dbt will work on up to three models at once without violating dependencies; the actual number of models it can work on is constrained by the available paths through the dependency graph.

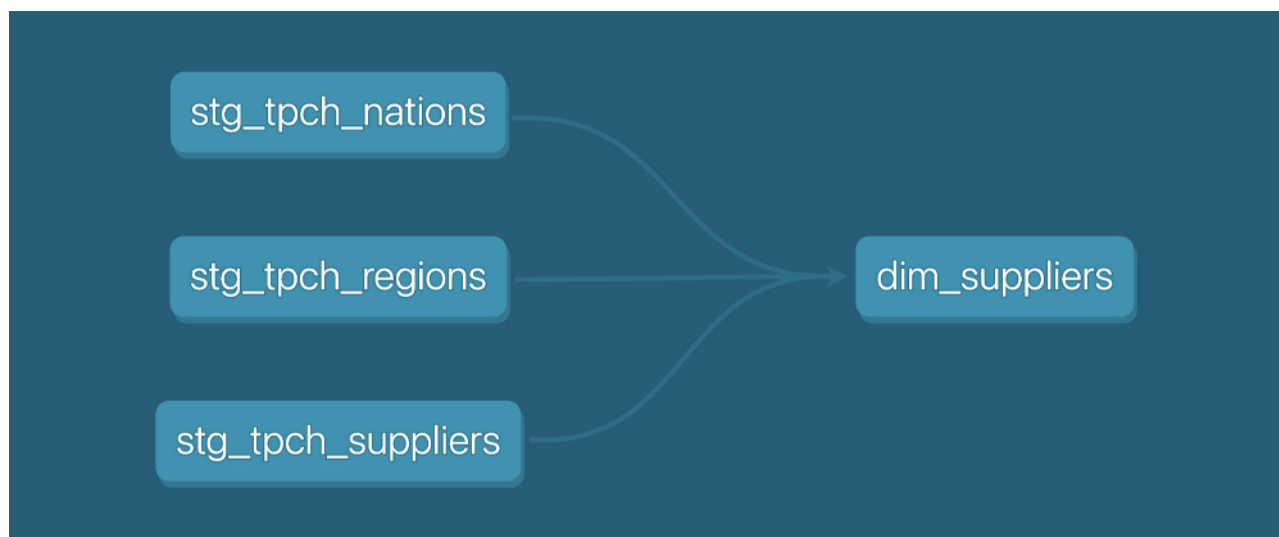


Figure 17: A sample dbt DAG

Sources work similarly to the `ref()` function, with the key distinction being that rather than telling dbt how a model relates to another model, sources tell dbt how a model relates to a *source object*. Declaring a dependency from a model to a source in this way enables a couple of important things: It allows you to select from source tables in your models, and it opens the door to more extensive project testing and documentation involving your source data. Figure 18 (below) shows a sample dbt DAG including a source node. The green node represents the source table that `stg_tpch_nation` has a dependency on.

WRITE MODULAR, DRY CODE

Use Jinja to write DRY code and operationalize Snowflake administrative workflows.

dbt allows you to use [Jinja](#), a Pythonic templating language that can expand on SQL's capabilities. Jinja gives you the ability to use control structures and apply environment variables.

Pieces of code written with Jinja that can be reused throughout a dbt project are called macros. They are analogous to functions in other programming languages, allowing you to define code in one central location and reuse it in other places. The `ref` and `source` functions mentioned above are examples of Jinja.

In addition to being helpful for environmental logic, macros can help operationalize Snowflake

administrative tasks such as grant statements, or they can systemically remove deprecated objects.

In the past, during object creation, there often needed to be a parallel administrative workflow alongside development that ensured proper permissions were granted on Snowflake objects. Today all of this can be done via Snowflake GRANT statements. dbt adds another layer of functionality here: It allows you to take all your GRANT statements, ensure they are run consistently, and version control them for simple auditability.

See [this example of a macro written to GRANT statements](#). This macro, once implemented as a [dbt hook](#), ensures that the GRANT statements are run after every dbt run, thus ensuring the right roles have access to newly created objects or future objects.

Similarly, as projects grow in maturity, it's common for them to have deprecated or unused objects in Snowflake. dbt allows you to maintain a standardized approach for culling such objects, using macros such as the one mentioned [here](#). This allows you to operationalize how you tidy up your instance and to ensure that it is done on a schedule (via a dbt job).

Macros, in addition to making your SQL more flexible, allow you to compartmentalize your Snowflake administrative code and run it in a systematic fashion.



Figure 18: A sample dbt DAG including a source node

USE DBT TESTS AND DOCUMENTATION

Have at least one dbt test and one model-level description associated with each model.

Robust systems of quality assurance and discovery are key to establishing organizational trust in data. This is where dbt tests and documentation are invaluable.

[dbt tests](#) allow you to validate assumptions about your data. Tests are an integral part of a CI/CD workflow, allowing you to mitigate downtime and prevent costly rebuilds. Over time, tests not only save you debugging time, but they also help optimize your usage of Snowflake resources so you're using them where they are most valuable rather than to fix preventable mistakes.

We recommend that, unless there is a compelling reason not to, every dbt model has a test associated with it. Primary key tests are a good default, as failure there points to a granularity change.

When you implement a CI/CD process, be sure to use [Slim CI](#) builds for systemic quality checks. With Slim CI, you don't have to rebuild and test all your models; you can instead instruct dbt to run jobs on only modified or new resources. This allows you

to use the node selector `state:modified` to run only models that have changes, which is much more resource-efficient.

[dbt Documentation](#) ensures that your data team and your data stakeholders have the resources they need for effective data discovery. The documentation brings clarity and consistency to the data models your team ships, so you can collectively focus on extracting value from the models instead of trying to understand them.

Every dbt model should be documented with a model description and, when possible, a column-level description. Use [doc blocks](#) to create a description in one file to be applied throughout the project; these are useful particularly for column descriptions that appear on multiple models.

If you're interested in documentation for the Snowflake side, apply [query tags](#) to your models. These allow you to conveniently tag in Snowflake's query history where a model was run. You can get as granular as is convenient there, by either implementing model-specific query tags that allow you to see the query run attributed to a specific dbt model or by having one automatically set on the project level, such as with the following macro:

```
{% macro set_query_tag() -%}

  {% set new_query_tag = model.name %} {%# always use model name #}

  {% if new_query_tag %}

    {% set original_query_tag = get_current_query_tag() %}

    {{ log("Setting query_tag to '" ~ new_query_tag ~ "'. Will reset to '" ~
original_query_tag ~ "' after materialization.") }}

    {% do run_query("alter session set query_tag = '{}'.format(new_query_tag)) %}

    {{ return(original_query_tag) }}

  {% endif %}

  {{ return(none) }}

{% endmacro %}
```

USE PACKAGES

Don't reinvent the wheel. Use packages to help scale up your dbt project quickly.

Packages can be described as dbt's version of Python libraries. They are shareable pieces of code that you can incorporate into your own project to help you tackle a problem someone else has already solved or to share your knowledge with others. Packages allow you to free up your time and energy to focus on implementing your own unique business logic.

Some key packages live on the [dbt Package Hub](#). There, you can find packages that simplify things such as:

- [Transforming data from a consistently structured SaaS data set](#)
- [Writing dbt macros that solve the question "How do I write this in SQL?"](#)
- [Navigating models and macros for a particular tool in your data stack](#)

Every dbt project on Snowflake should have at least the [dbt_utils package](#) installed. This is an invaluable package that provides macros that help you write common data logic, such as creating a surrogate key or a list of dates to join. This package will help you scale up your dbt project much faster.

If you're using the Snowflake Dynamic Data Masking feature, we recommend using the [dbt_snow_mask package](#). This package provides pre-written macros to operationalize your dynamic masking application in a way that's scalable and follows best practices.

The [snowflake spend package](#) is another great package that allows you to easily implement analytics for your Snowflake usage. You can use it to model how your warehouses are being used in detail, so you can make sure you use only the resources you actually want to use. We recommend using this package with a job in dbt Cloud, so you can easily set up alerting in case your usage crosses a certain threshold.

At larger organizations, it is not uncommon to create custom internal packages that are shared among teams. This is a great way to standardize logic and definitions as projects expand across multiple repositories (something we discuss in a later

section). Doing this ensures that projects are aligned in companywide definitions of, for example, what a customer is, and it limits the amount of [WET \(write every time\) code](#).

BE INTENTIONAL ABOUT YOUR MATERIALIZATIONS

Choose the right materialization for your current needs and scale.

One of the easiest ways to fine-tune performance and control your runtimes is via materializations. Materializations are build strategies for how your dbt models persist in Snowflake. Four materializations are supported out of the box by dbt: **view**, **table**, **incremental**, and **ephemeral**.

By default, dbt models are materialized as views. Views are saved queries that are always up to date, but they do not store results for faster querying later. If an alternative materialization is not declared, dbt will create a view. View materializations are a very natural starting point in a new project.

As the volume of your data increases, however, you will want to look into alternative materializations that store results and thus front-load the time spent when you query from an object. The next step up is a table materialization, which stores results as a queryable table. We recommend this materialization for any models queried by BI tools, or simply when you are querying a larger data set.

Incremental materialization offers a way to improve build time without compromising query speed. Incremental models materialize as tables in Snowflake, but they have more-complex underlying DDL, making them more complex configurations. They reduce build time by transforming only what has been declared to be a new record (via logic you supply).

In addition to the materializations outlined above, you also have the option of [writing your own custom materializations](#) in your project and then use them in the same way as you would use materializations that come with dbt. This enables you to declare the model to be materialized as a `materialized_view` and grants you the same abilities as maintaining lineage with the `ref` function, testing, and documentation.

OPTIMIZE FOR SCALABILITY

Even when they start lean, dbt projects can expand in scale very quickly. We have seen dbt projects with about 100 models expand, with good reason, to over 1,000 for large enterprises. Because of this, we recommend the following approaches to help you avoid issues down the line.

Plan for project scalability from the outset

Being proactive about project scalability requires that you have a good understanding of how your team members work with each other and what your desired workflow looks like. We recommend reading this [Discourse post](#) as an overview of factors and then considering what options are right for your team.

Generally speaking, we recommend maintaining the mono-repository approach as long as possible. This allows you to have the simplest possible git workflow and provides a single pane through which to oversee your project.

As your project and data team scale, you may want to consider breaking the project up into multiple repositories to simplify the processes of approval and code promotion. If you do this, we recommend you make sure your Snowflake environments are aligned with this approach and there is a continual, clear distinction regarding what project, git branch, and users are building into which Snowflake database or schema.

Follow a process for upgrading dbt versions

One of the ways teams get caught off guard is by not establishing how they plan to go about upgrading dbt. This can lead to teams deciding to forgo upgrading entirely or to different team members having different versions, which has downstream effects on which dbt features can be leveraged in the project. Being on top of upgrading your dbt version ensures you have access to the latest dbt functionality, including support for new Snowflake features.

Our recommended method to upgrading dbt is to use a timeboxed approach. You should start by reading the necessary changelog and migration guides to get a sense of what changes might be needed for your

dbt project. Next, implement a “timebox” for testing the upgrade and, if possible, require either every user or a group of power users to upgrade to the latest dbt version for a set amount of time (such as 1 hour.)

In that time, you should make clear there should be no merges to production and users should develop only on the updated version. If the test goes smoothly, you can then have everyone on your team upgrade to the latest version both in the IDE and locally (or continue with their updated version, as the case may be.) On the other hand, if for some reason the test was not successful, you can make an informed decision on whether your team will stay on the newest release or roll back to the previous dbt version, and then plan for the next steps to upgrade at a later date.

CONCLUSION

Modern businesses need a modern data strategy built on platforms that support agility, scalability, and operational efficiency. dbt and Snowflake are two technologies that work together to provide just such a platform. They’re capable of unlocking tremendous value when used together. Following the best practices highlighted in this white paper allows you to unlock the most value possible while minimizing the amount of resources expended.

CONTRIBUTORS

Contributors to this document include:

- **BP Yau**
Senior Partner Sales Engineer, Snowflake
- **Amy Chen**
Partner Solutions Architect, dbt Labs

REVIEWERS

Thanks to the following individuals and organizations for reviewing this document:

- **Dmytro Yaroshenko**
Principal Data Platform Architect, Snowflake
- **Jeremiah Hansen**
Principal Data Platform Architect, Snowflake
- **Brad Culberson**
Principal Data Platform Architect, Snowflake
- **Azzam Aijazi**
Senior Product Marketing Manager, dbt Labs

DOCUMENT REVISIONS

Date: September 2021

Description: First publication

ABOUT DBT LABS

dbt Labs was founded to solve the workflow problem in analytics, and created dbt to help. With dbt, anyone on the data team can model, test, and deploy data sets using just SQL.

By applying proven software development best practices like modularity, version control, testing, and documentation, dbt's analytics engineering workflow helps data teams work faster and more efficiently to bring order to organizational knowledge. [Getdbt.com](https://getdbt.com).

ABOUT SNOWFLAKE

Snowflake delivers the Data Cloud—a global network where thousands of organizations mobilize data with near-unlimited scale, concurrency, and performance. Inside the Data Cloud, organizations unite their siloed data, easily discover and securely share governed data, and execute diverse analytic workloads. Wherever data or users live, Snowflake delivers a single and seamless experience across multiple public clouds. Snowflake's platform is the engine that powers and provides access to the Data Cloud, creating a solution for data warehousing, data lakes, data engineering, data science, data application development, and data sharing. Join Snowflake customers, partners, and data providers already taking their businesses to new frontiers in the Data Cloud. [Snowflake.com](https://snowflake.com).



©2021 Snowflake Inc. All rights reserved. Snowflake, the Snowflake logo, and all other Snowflake product, feature and service names mentioned herein are registered trademarks or trademarks of Snowflake Inc. in the United States and other countries. All other brand names or logos mentioned or used herein are for identification purposes only and may be the trademarks of their respective holder(s). Snowflake may not be associated with, or be sponsored or endorsed by, any such holder(s).

ENDNOTES

¹ bit.ly/3BqWt3T ² tabsoft.co/2YcdJM3