Build scalable and trustworthy data pipelines with dbt and BigQuery

Rabi Abbasi and Maruti C

1. Abstract 2
2. What is BigQuery 3
   2.1. How does BigQuery Work 3
   2.2 Benefits of using BigQuery 5
3. What is dbt 6
   3.1 Why dbt ? 6
   3.2 dbt Cloud 7
   3.3 Documentation and Dependency Handling 8
   3.4 Making Code Modular Using Macros 9
   3.5 Jumpstarting dbt development with Packages 10
   3.6 Document as you code with dbt 10
   3.7 Centrally defined business metrics 11
   3.8 Resources for Learning More 11
4. Customer Use Case 11
5. Data Architecture for dbt and BigQuery 12
   5.1 Typical data warehousing pipeline for BigQuery 12
   5.2 Organizing BigQuery Resources 14
      5.2.1 Environment Setup 14
      5.2.2 Single BigQuery project 15
      5.2.3 Unified source project & environment conformed projects 16
      5.2.4 Unified source project & business conformed projects 16
   5.3 dbt Projects 17
   5.4 Additional Considerations 18
6. Audit and Security 18
   6.1 Connecting BigQuery to dbt Cloud 18
   6.2 Set up Role Based Access Controls (RBAC) 19
   6.3 Managing Encryption 19
   6.4 Access Management 19
   6.5 Working with Sensitive Data 20
   6.6 Logging 21
   6.7 Labels and Tags 21
7. Optimizing Performance 22
   7.1 Identify Bottlenecks 23
   7.2 Optimising Joins 24
   7.3 Partitioning 24
         7.3.1 Improve query performance and reduce cost on large table joins 25
         7.3.2 Control costs in your dbt development environment 26
         7.3.3 Reduce table storage cost 27

7.4 Clustering 27
7.5 Date Sharded Tables 28
7.6 Denormalization 28
7.7 Merge Behaviour 29
7.8 Materialized Views 31
7.9 Writing effective SQL using DRY principles 31
7.11 Cache queries using BI Engine 32
7.12 Using Bigquery ML and dbt 33
8. Billing & Resource Management 33
8.1 Estimate 34
8.2 Monitor 34
8.3 Optimize 35
9. Appendix 36
9.1 Key Concepts and Terminology 36

1. Abstract

Data is collected by every organization, which gives us the opportunity to do something useful with it: make informed decisions, solve problems and drive business growth. To deliver on these outcomes, It is essential to have high performing data assets which are trusted across the organization.

BigQuery is a petabyte scale data warehouse that empowers us to answer difficult questions expressed in SQL queries. dbt is a transformation workflow tool that helps data practitioners transform, test, document and deploy their data in BigQuery and other data warehouses. With dbt, analytics engineers are able to deliver well-defined, transformed, tested, documented, and code-reviewed data sets in BigQuery, making collaboration and maintenance easier.

This whitepaper presents key concepts of BigQuery and dbt and talks about the best practices for utilizing their combined power to build scalable and trustworthy data pipelines.

2. What is BigQuery

Bigquery is a fully managed, serverless, highly scalable enterprise data warehouse, designed to handle large-scale datasets and analytics workloads with zero infrastructure management. It is well-suited for organizations that need to analyze data quickly and cost-effectively. The query engine allows the execution of ANSI SQL which is a widely adopted data processing language.

BigQuery maximizes flexibility by separating the compute engine that analyzes your data from the underlying storage layer. This allows the distributed analysis engine to query terabytes in seconds and petabytes in minutes.

BigQuery offers an industry leading 99.99% uptime SLA. BigQuery automatically stores copies of your data in two different Google Cloud zones within a single region in the selected location. In addition to storage redundancy, BigQuery also maintains redundant compute capacity across multiple zones. By combining redundant storage and compute across multiple availability zones, BigQuery provides both high availability and durability

BigQuery has robust security measures in place to protect your data, while also providing the controls and compliance certifications required to meet the needs of regulated industries and organizations handling sensitive data.

2.1. How does BigQuery Work

BigQuery's serverless architecture decouples storage and computation and allows them to scale independently. This offers both immense flexibility and cost controls for customers because they don't need to keep their expensive compute resources up and running all the

time.

Under the hood, BigQuery uses a range of multi-tenant services powered by Google's advanced infrastructure technologies such as Dremel, Colossus, Jupiter, and Borg. Let's discuss each of them in more detail.

Dremel: The Execution Engine

Dremel converts SQL queries into an execution tree for efficient data processing. The leaves of the tree are 'slots' for heavy data reading and computation, slot is a unit of compute resource that is equivalent to a single CPU core and the branches are 'mixers' for aggregation, with 'shuffle' for rapid data transfer.

Dremel is widely used across various Google services , so there's great emphasis on continuously making Dremel better. BigQuery users get the benefit of continuous improvements in performance, durability, efficiency and scalability, without downtime and upgrades associated with traditional technologies.

Colossus: Distributed Storage

BigQuery relies on Colossus, Google's generation distributed file system. Colossus also handles replication, recovery (when disks crash) and distributed management (so there is no single point of failure). BigQuery leverages the ColumnIO columnar storage format and compression algorithm to store data in Colossus in the most optimal way for reading large amounts of structured data.

Colossus uses a mix of flash and disk storage to meet different access patterns and frequencies. Hot data is stored on flash to reduce latency whereas Disk-based storage is intelligently managed to avoid overprovisioning and wasted disk IOPs. Data is evenly distributed across all drives and then moved to larger capacity drives as it becomes colder. This strategy maximizes storage efficiency and works well for analytics workloads where data tends to cool off over time.

In addition, Colossus uses Colossus Flash Cache, which is an intermediate cache layer that improves the performance and reliability of Google's services by reducing the number of hard disk reads. It is a cost-effective way to increase IO capacity and reduce latency. The underlying storage infrastructure caches data in Colossus Flash Cache based on access patterns. This way, queries rarely need to go to disk to retrieve data; the data is served up quickly and efficiently from Colossus Flash Cache.

Colossus enables applications and clusters to scale to exabytes of storage and tens of thousands of machines seamlessly, without paying the penalty of attaching much more expensive compute resources — typical with most traditional databases.

Borg: Compute

Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines. Borg is Google's precursor to Kubernetes, which is open-sourced by Google to help enterprises automate the deployment, scaling, and management of containerized applications.

BigQuery is orchestrated via Borg. Borg is responsible for allocating compute resources to BigQuery jobs. When a user submits a SQL query to BigQuery, Borg will first determine how many slots are needed to execute the query. Borg will then allocate the requested number of slots to the BigQuery job.

Jupiter: The Network

The compute and storage in BigQuery are physically separated and the 'shuffle' process, that is between the storage and compute layer, takes advantage of Google's constantly evolving Jupiter network to transfer data rapidly between locations. This high-speed, low-latency network is made up of a distributed network of nodes. Each node is interconnected with other nodes via high-speed links. This network architecture allows for the rapid transfer of data between the compute and storage resources

Jupiter can now handle over 6 petabytes per second of data center bandwidth. Compared to other leading alternatives, the Jupiter network consumes 40% less power, incurs 30% lower costs, and experiences 50 times less downtime, all while reducing the flow completion by 10% and improving throughput by 30%.

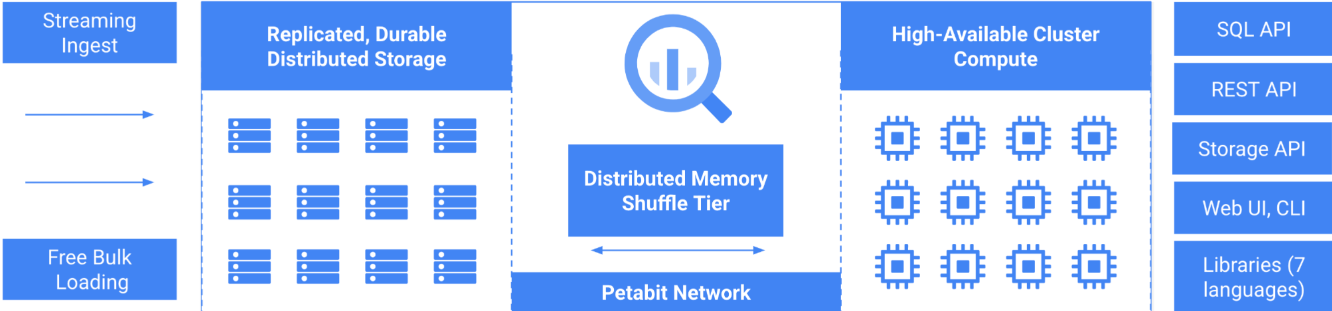


Figure 2.2.1: BigQuery's serverless architecture.

2.2 Benefits of using BigQuery

Some of the benefits for organizations that use BigQuery include:

Easy access: You can load a variety of formats of data including Parquet, Avro, CSV and JSON files into BigQuery from a variety of sources, as well as other Google Cloud services like Cloud Storage. External tables allows access to external data including Amazon S3 and Azure blob storage using BigQuery omni and and federated queries allows querying data in place from databases like cloud spanner and cloud SQL without directly loading it to BigQuery tables, making sourcing data easier.

Working with large datasets: BigQuery is highly scalable and can work with datasets either they are megabytes or petabytes in size, making it a cost effective choice for data warehousing and big data analytics. The capacity pricing model which takes advantage of BigQuery editions and the on-demand pricing model gives you the flexibility to choose between ad-hoc or predictable capacity workloads.

Machine learning: BigQuery ML lets you create and execute machine learning models in BigQuery using standard SQL queries. BigQuery democratizes machine learning by letting SQL practitioners build models using existing SQL tools and skills. It increases development speed by eliminating the need to move data. Furthermore, BigQuery closely integrates with Vertax AI, and solves the need for customers which prefers a unified data and AI platform.

Securing data and complying with international privacy regulations: access control mechanisms including roles-based access control allows organizations to fine-tune access to data based on user needs. Bigquery encrypts all data at rest and transit using AES-256 encryption. If your organization requires meeting regulatory compliance, BigQuery meets standards for HIPAA, ISO 27001, PCI DSS, SOC 1 Type II, and SOC 2 Type II, among others.

Sharing data: Authorized view lets you share query results with particular users and groups without giving them access to the underlying source data. An authorized dataset lets you authorize all of the views in a specified dataset to access the data in the shared dataset. Additionally, Analytics Hub allows efficient exchange of data across the organization either internally and externally.

3. What is dbt

dbt is a transformation workflow tool that is used to modularize and centralize your analytics code, while also providing your data team with guardrails typically found in software engineering workflows. It allows you to collaborate on data models, version them, and test and document queries before safely deploying them to production, with monitoring and visibility.

You can choose dbt as a Core or Cloud offering. dbt Core is an open source command line framework that enables data teams to transform data while maintaining the infrastructure themselves. dbt Cloud is a managed service from dbt Labs. dbt Cloud provides a web-based

IDE, job orchestration, and data observability features for the fastest and most reliable way to deploy dbt, it runs in all cloud providers including Google Cloud..

3.1 Why dbt ?

Collaboration: dbt creates a common ground for everyone in the data team to work together more efficiently, providing:

- **Built-in Git integration** so teams can work in parallel and manage code changes efficiently
- A **testing framework** to help developers validate assertions about their models during and after development.
- **Hosted, auto-generated documentation** to enable business users to discover and understand data assets created in a dbt project
- Support for **code modularity** via Jinja and macros, so teams can share, reuse, and build upon existing code more efficiently, reducing redundant work.

Orchestration/Automation: dbt Cloud allows teams to automate the execution of their pipeline, making it easier to schedule and run data pipelines on a schedule. dbt webhooks can be used to build continuous integration pipelines, allowing pull requests to be tested before merging, while Slim CI allows for further refinement by allowing you to run and test models based on the state and selection criteria.

Monitoring:
dbt Cloud provides detailed execution logs, auditing and alerting features, making it easy to keep track of your data pipeline's performance and troubleshoot any issues that may arise.

Security:
dbt uses a push down approach to analytics, doing all the calculations at the database level, making the entire transformation process faster, more secure, and easier to maintain. All connections are encrypted by default. dbt meets all modern compliance standards including SOC2 Type II, GDPR, ISO 27001:2013, ISO 27701:2019.

3.2 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 3.2.1). dbt Cloud also eliminates the setup and maintenance work required to manage data transformations in BigQuery at scale. A turn-key adapter establishes a secure connection built to handle enterprise loads, while allowing for fine-grained policies and permissions.

To get started with dbt and BigQuery, you can install the BigQuery adapter for local development on the command line or create a free trial dbt Cloud account to leverage the

in-platform IDE and scheduler. Note that adapters define how dbt connects with BigQuery or various supported data platforms.

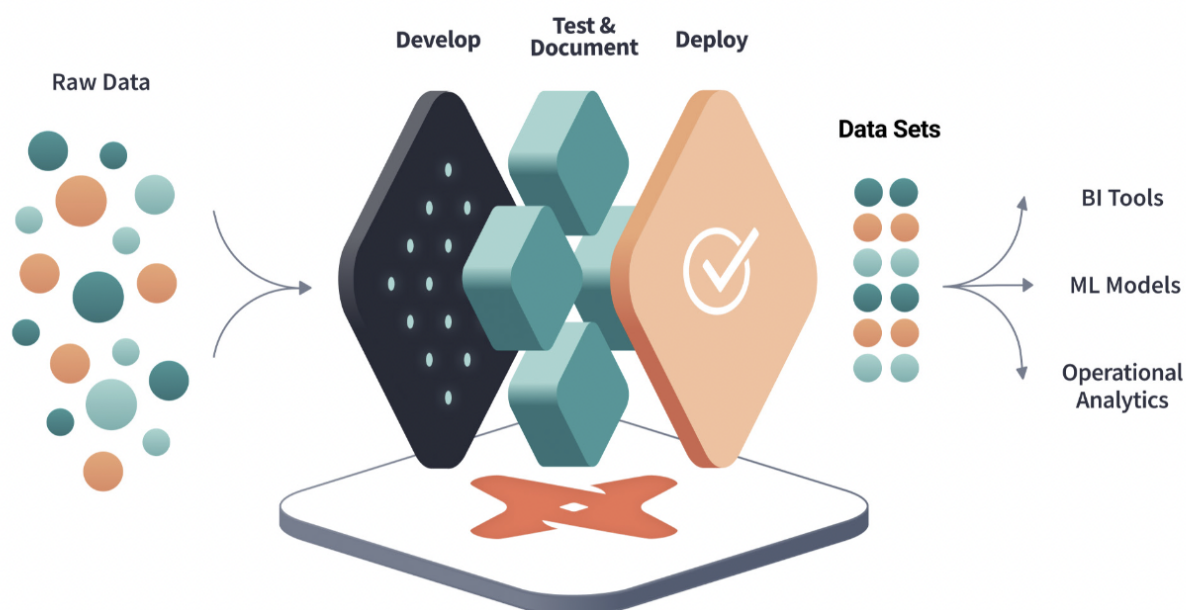


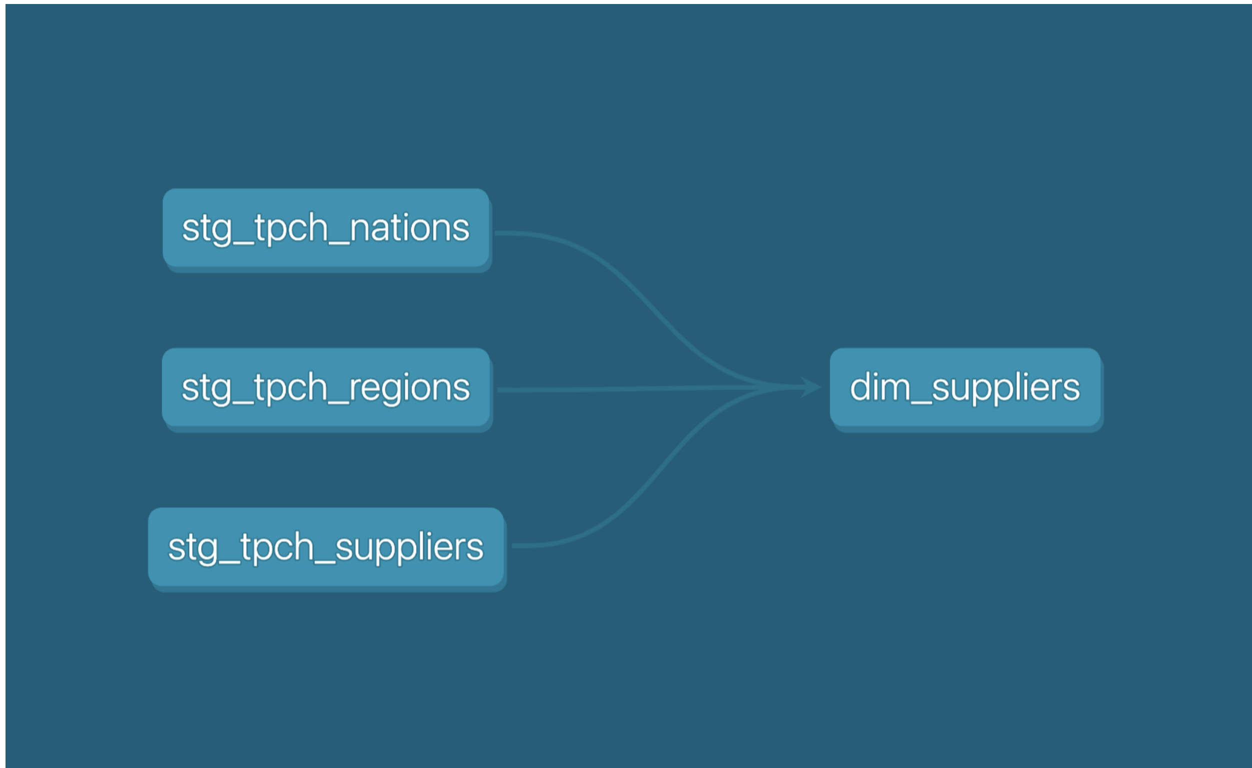
Figure 3.2.1: dbt Cloud provides a centralized experience for developing, testing, scheduling, and investigating data models.

3.3 Documentation and dependency handling

A keystone of dbt's functionality is the ref function. The ref function in dbt automatically establishes a lineage from the dbt model being referenced to the model declared in the reference. By using the ref function, dbt is able to (1) infer dependencies and (2) ensure that the correct upstream tables and views are selected based on your environment. Always use the ref function when selecting from another model, rather than using the direct relation reference (e.g. my_schema.my_table).

When ref() is paired with the use of threads, dbt executes models following an optimal path without requiring manual input. As you increase the number of threads for a run, dbt increases the number of paths in the graph that it can work on at the same time, thus reducing the run time of your project.

Models, sources, tests, and snapshots all represent nodes in dbt projects. By pairing threads with dbt's node selection, you can optimize your dbt run by only running what you need without explicitly declaring the model build order. This makes it possible to limit commands to *only* modified models, or to restart from failure in parallel to running models.



A sample dbt DAG: In this example, the user has declared in model dim_suppliers a reference to stg_tpch_nations, stg_tpch_regions, and stg_tpch_suppliers. At time of a dbt run, if a user has declared 3 threads, dbt would know to execute the first three staging models prior to running dim_suppliers. By specifying 3 threads, dbt will work on up to 3 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.

In dbt, you can name and describe data loaded into BigQuery by declaring it as a source. This allows you to:

- Create dbt models that select from the source object
- Document dependencies between sources and downstream models via dbt's data lineage DAG
- Set up validations, tests, and other quality controls for sources — such as source freshness checks

A sample dbt DAG including a source node: The green node here represents the source table that stg_tpch_nation has a dependency on.

3.4 Making code modular using macros

In dbt, you can combine SQL with Jinja, a templating language similar to Python syntax. Jinja provides a way to apply environment variables and use control structures (like if statements and for loops), extending what is possible with SQL alone.

Macros are pieces of code written with Jinja that can be reused throughout the dbt project. They are analogous to functions in other programming languages, allowing you to define code in one central location and re-use it in other places. The ref and source functions mentioned above are examples of Jinja.

Macros are useful for:

- Defining environmental logic
- Operationalizing BigQuery administrative tasks like grant statements
- Removing deprecated objects systematically
- Reducing code redundancies

3.5 Jumpstarting dbt development with packages

dbt packages are libraries of open-source models and/or macros that help automate or simplify common routines. The package hub is a directory of packages maintained by members of the dbt community, including:

- Modeling methods for common data sources, like Facebook Ads or Netsuite
- Data assertions and validations that go beyond dbt's standard tests
- dbt_ml, to help with training, auditing, and using BigQuery ML models
- dbt_utils, a package that automates data-modeling routines such as creating a date spine, unpivoting columns, and more.

Users can add hub packages to their dbt project by adding the package name and version to the `packages.yml` file and running `dbt deps`. Once done, they can use macros and models from the package throughout their project using the normal ref and macro syntax. For private, local, or git packages not in the dbt package hub, follow package installation directions provided here.

Many teams also create and share internal packages to help standardize logic and definitions across multiple dbt repositories.

3.6 Document as you code with dbt

You can use meta fields to establish owners for the components of your dbt project (sources, models, tests, and macros). You can also use meta fields to flag objects that contain sensitive data like PII, and to implement policies via data governance tools that integrate with dbt.

When defined, exposures represent downstream BI dashboards, applications, or data science pipelines that rely on data transformed in your dbt project. Defining exposures enables you to:

- Use dbt's node selection syntax to run dbt commands that reference those exposures.
- Include the exposures in your dbt documentation & lineage graph. This can help with identifying upstream resources (when debugging data issues) or downstream dependencies (when deprecating a resource).

You can generate dbt documentation by running two CLI commands (dbt docs generate && dbt docs serve) and accessing the results through local host, or by navigating to the "Documentation" section in dbt Cloud. You can also use the dbt Cloud Metadata API to send the information over to partner data catalogs.

3.7 Centrally defined business metrics

The dbt Semantic Layer helps organizations eliminate inconsistencies and duplicate code across downstream analytics tools. It enables teams to define metrics in one central data-modeling layer, and propagate that logic to BI platforms — ensuring that business logic referenced anywhere will be exactly the same everywhere.

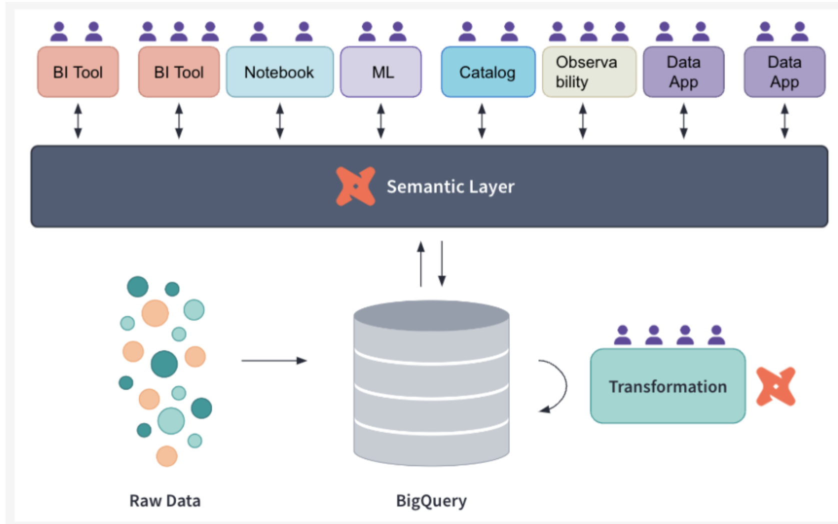


Figure 3.7.1: Consistent metrics across data applications with dbt Semantic Layer

3.8 Resources for learning more

To learn more about using dbt and BigQuery:

- Get started with a guide for connecting to BigQuery and setting up a first dbt project.
- Go deeper with the dbt Fundamental course, featuring video instructions and exercises.
- Use the essential dbt project checklist to compare your project to best practices.

4. Customer example: Viktor Myrvang

Viktor Myrvang is a data platform team lead at Elvia, Norway's largest distribution system operator (DSO). Elvia is responsible for delivering power to over 2M people, or 40% of the country's population.

"We started using BigQuery when we rolled out a million smart meters, which collect data every hour for all our customers. Using that data, we can map the grid load during the day and determine how to optimize around it." With BigQuery, Elvia could store the vast data volume. But after a few years, the team realized the data in their warehouse was becoming unmanageable.

"It was becoming increasingly hard to understand what data was there." After evaluating solutions, Elvia chose dbt Cloud to enable teams to own their code and build models from the

data in BigQuery. “We're heavily invested in infrastructure-as-code—with BigQuery and dbt, we can automate most aspects of project admin, access, and quality.”

BigQuery provides an easy-to-use platform, and dbt layers on development guardrails, dependency tracking, documentation, and lineage. The combination of tools facilitates widespread adoption of their data platform while also embedding best practices across the organization. Teams now reuse data products to build new assets, and frequent analytical questions can be answered with little to no new work.

“Each data engineering, data science, and analytics team leveraging BigQuery and dbt allows us to better understand our customer's consumption and behavior; which ultimately influences decisions on how we will develop Oslo and the surrounding region for the next 20 years.”

The remainder of this paper dives into the dbt and BigQuery best practices that Elvia and thousands of other customers have leveraged to build scalable, high-performing data platforms.

5. Data architecture for using dbt and BigQuery together

Transforming data used to be a slow resource-intensive process. Modern cloud warehousing has taken away these limits, allowing data transformation to happen in-place. We will discuss the steps to build a modern data warehouse using the in-place transformation or ELT approach.

5.1 Typical data warehousing pipeline for BigQuery

Data Ingestion: BigQuery offers multiple methods for data ingestion to help with different use cases , including batch loading a set of data records, streaming individual records in real time or batches of records with microbatches in quasi real-time, using queries to generate new data and append or overwrite the results to a table, and utilizing third-party applications and services.

Choosing the right approach for data ingestion can help ensure optimal performance and efficiency in analyzing and processing data with BigQuery.

When choosing a data ingestion method, consider:

- **File Formats:** Self describing file formats like Avro or Parquet are recommended. They reduce the overhead of maintaining separate schema files and allow for performant data processing.
- **Schema Identification**: Do not use the BigQuery auto-detect schema on source files for long term use as it can introduce uncertainty in your pipeline. If the data source requires a schema, then explicitly provide it.
- **Encoding:** Stick to UTF-8 encoding for both nested and flat source data. This eliminates the need to convert/translate the data encoding later on.
- **Compression:** JSON and CSV files are not natively compressed. Consider compressing them for transfer bandwidth control and long term storage.
- **Pricing:** BigQuery does not charge for loading batched datasets but you will be charged for the amount of data stored and queried.
- **External Data:** By using external tables you are able to use data that is external to BigQuery. However, be mindful of the performance impact.
- **Streaming Ingestion:** Consider using pubsub BigQuery subscriptions for a simpler streaming setup.

dbt supports the external tables package to help create/replace/refresh external tables, using the metadata provided in the .yml file source definitions. This enables you to define GCS files and external raw data as dbt sources without relying on any upstream processes.

Data Processing: Data processing refers to the transformation of raw data into usable information. This can include operations like cleaning, enriching, and aggregating data to ease analysis.

dbt helps teams collaborate on this process, with safeguards like version control, testing and documentation of your queries, allowing you to safely deploy them to production, with monitoring and visibility.

Data Visualization/Consumption: After data is cleaned and prepared, it can be:
- Visualized in products like Google's Looker and Looker Studio
- Fed back to upstream applications via Reverse ETL, to enrich user experience
- Used to train machine-learning models

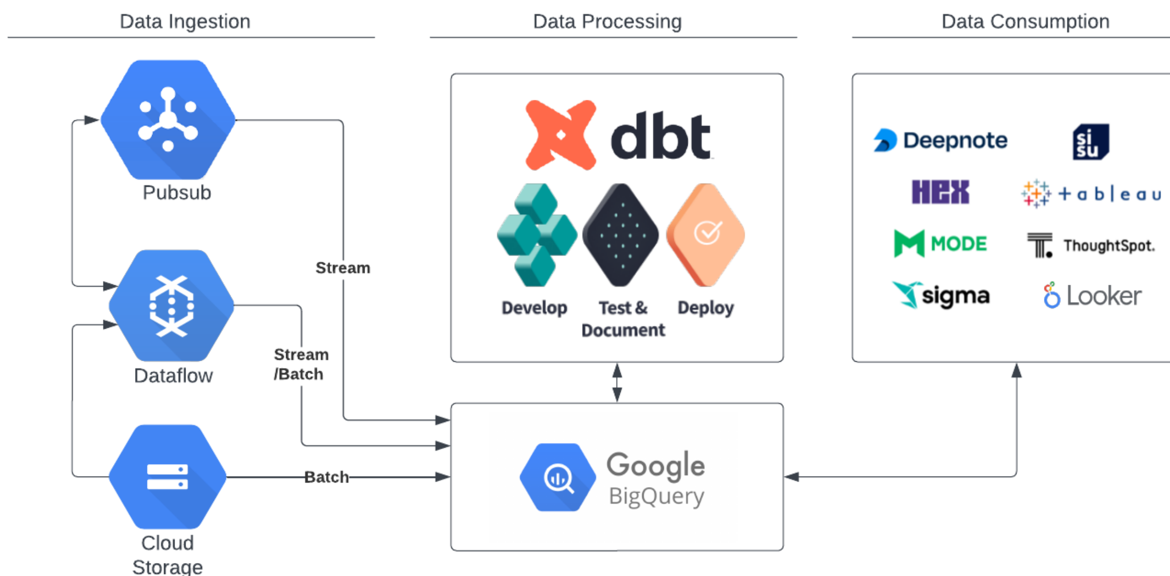


Figure 5.1.1: Ingestion, Process and Consumption stages of a typical data warehouse pipeline and where dbt fits in the life cycle.

5.2 Organizing BigQuery resources

BigQuery resources are organized in a hierarchy. You can use this hierarchy to manage aspects of your dbt + BigQuery workloads such as environments, permissions, quotas, slot reservations, and billing.

When deciding on which setup is best for you, consider:

- Current team size and expected growth
- Number and growth of dbt models, and BigQuery quota limitations
- Access Management overhead
- Billing Requirements (Unified or Per Vertical/Function)

5.2.1 Environment Setup

You can use environments in BigQuery and dbt to test data before exposing it to end-users and the applications they interact with.

Teams typically take one of these two approaches to quality assurance (QA):

1. Test in a development environment before promoting code to a production environment.

2. Test in two environments — QA and development — before promoting code to a production environment. Teams with integration tests, multiple reviewers, or other complex QA requirements may prefer this workflow.

For the purpose of this writeup, we will consider the simpler approach of managing QA workflows inside of the development environment.

5.2.2 Single BigQuery project

A simple monolithic project structure is used to contain all dbt and Bigquery artifacts. Billing is unified for development and production workloads but it can be difficult to predict costs, storage and compute. Access management, while easy to manage, can be difficult to audit, and can result in over provisioned access. Project might be at the risk of frequently hitting quota limits. Figure 5.2.2.1, shows how to structure all datasets under a single project structure:

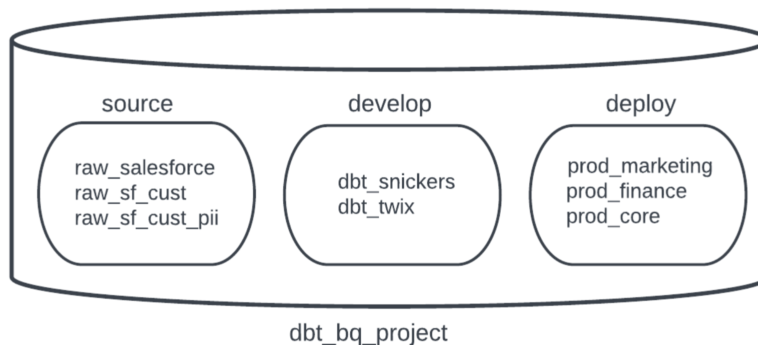


Figure 5.2.2.1: Environment setup with a single BigQuery project

- Raw Datasets: stages all raw source data. We suggest the datasets containing PII are segregated from clean datasets. Clean or obscured datasets can be consumed for development purposes while PII data is only exposed for production use.
- Dev Datasets: we suggest that dbt developers have their own sandbox/development dataset, prefixed with their name (ex. dbt_<username>), for development. In this space, dbt developers can create, update, and delete models without the risk of impacting others' development spaces
- Prod Datasets: organize your production data objects by functions or verticals. To do this, you can put your dbt models into different production datasets. We recommend using custom schemas. For example, you would call the schema that holds your production marketing models prod_marketing. If it's a core model needed by multiple teams, you can name the schema prod_core

5.2.3 Unified source project & environment conformed projects

A single project is dedicated to hosting all raw source data. Additional projects per environment, such as testing and production are used to execute function/vertical aligned data pipelines. This setup features simple access management and unified billing. As a downside the production project can hit BigQuery project quotas depending on demand, so consider quota limit increase or consider a more granular approach as per section 5.2.4. Figure 5.2.3.1, depicts a typical three project setup:

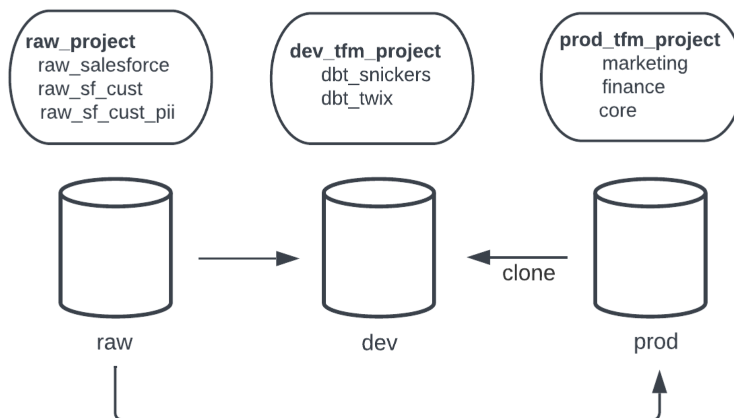


Figure 5.2.3.1: Environment setup with unified source and environment conformed projects

- Raw Project: stages all raw source datasets.
- Dev Transformation Project: segregated sandbox/development environment for all dbt developers where each dataset is prefixed with their name (ex. dbt_<username>)
- Prod Transformation Project: segregated production environment where each dataset is organized by function or vertical. No need to prefix the dataset names by prod_ as we already made that distinction at the project level.
- Use zero copy cloning to clone production datasets/tables into the development environment. A table clone is a lightweight copy of another table and you are only charged for storage of data that differs from the original table. Creating table clones not only saves costs but speeds up the onboarding process, not requiring you to rebuild all models from scratch. You can clone datasets by creating a custom macro: use the dbt_utils.get_relations_by_prefix function to iterate over the tables (see example) and clone the dataset using dbt run-operation
- For compatibility with dbt, ensure datasets across projects are either all in one multi-regional location (e.g. EU, US), or all in a specific regional location

5.2.4 Unified source project & business conformed projects

A single project is dedicated to hosting all raw source data. Each function/vertical has a dedicated project for building and executing data pipelines. This approach features modular project level access control management and function/vertical based billing: each department is billed separately based on on-demand or flat-rate usage of that project.

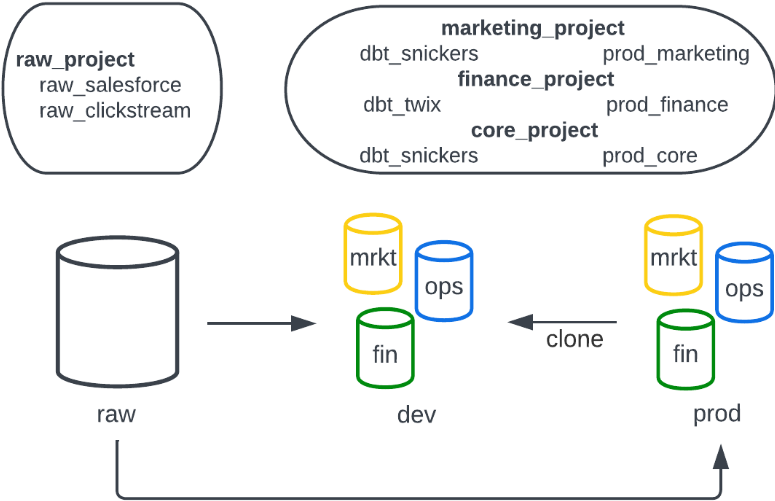


Figure 5.2.4.1: Environment setup with unified source and business conformed projects

We recommend that you unify the development and deployment environments to the same project. Having another set of environment conformed projects can result in a complex architecture, where access control can become increasingly difficult to manage. The development datasets follow the same naming standard ex. dbt_<username>, while the production datasets are named as prod_<function/vertical>.

5.3 dbt projects

A dbt project is a collection of data transformations (stored in .sql or .py files), tests, documentation, and configuration files, linked to one source repository and database connection. You can organize these resources under one project (mono repo) or split them into multiple granular projects (multi repo).

We recommend using a single project for simplicity, unless you need to separate resources to manage scale or meet security requirements.

For more on how to structure your dbt project, see the overview here.
The next big step forwards for analytics engineering, discusses the dbt multi project strategy and what to expect in the near future.

5.4 Additional considerations

We recommend the following practices to maintain data quality, developer experience, and an efficient analytics workflow:

- **Use naming conventions** for projects, tables, datasets, GCS buckets and data files to remove ambiguity and make data resources easier to discover.
- **Use the simplest git branching strategy possible**, to avoid adding complexity to developer workflow.
- **Use a pull request template** to help developers check for missing requirements before submitting code for review. The template can improve developer efficiency and ensure everyone follows a shared standard.
- **Use continuous integration** to automate the process of testing code before it is merged to production.
- **Choose an orchestration tool** like Google cloud's cloud composer, apache airflow or other robust tools available to automate and coordinate with your dbt jobs. These tools make it easier to manage and maintain them over time.

6. Audit and security

In this section we will cover the best practices for ensuring a secure and auditable data platform.

6.1 Connecting BigQuery to dbt Cloud

dbt provides two methods for securely connecting to BigQuery:

1. JSON key file: This method allows non-Enterprise dbt account users to quickly and accurately configure a connection to BigQuery. To use this authentication method:
    - Generate a JSON keyfile using the BigQuery credential wizard
    - Upload the file to dbt Cloud to auto-populate the relevant fields in connection settings of the Project
2. BigQuery OAuth: This method is available to dbt Enterprise accounts and provides an extra layer of security by configuring connections through OAuth. Credential requests are made under the SSL protocol and access is granted through a transitory token. To use this authentication method:
    - Create a client ID and secret for authentication with BigQuery

    - Provide it under the OAuth2.0 Settings section in connection settings

6.2 Set up role based access controls (RBAC)

By using role-based access control (RBAC) to manage database privileges in BigQuery, you can assign different privileges to roles and assign these roles to users to have more granular control of user access. Role-based permissions can be generated dynamically from configurations in an Identity Provider.

RBAC supports row level, and column level security. dbt Cloud offers RBAC for the dbt experience on the enterprise tier, allowing administrators to control who has access to develop the dbt project and alter job orchestration.

6.3 Managing encryption

BigQuery encrypts your content stored at rest. It handles and manages this default encryption without any additional actions. However you may have regulatory or internal requirements requiring you to manage your encryption yourself.

With customer-managed encryption keys (CMEK) you control and manage key encryption keys that protect your data. While working with dbt you can supply the customer managed key for one or group of models by providing the `kms_key_name` config.

It is recommended that you control compliance as an organization policy. This will ensure that CMEK is always required for all resources in your BigQuery project.

6.4 Access management

You can share your data or provide users access to models using Authorized Views and Grants.

An authorized view allows data sharing without giving users access to the underlying objects. It is also a great way to share data across BigQuery projects, making the users from the target project responsible for the query costs. With dbt you can create an authorized view using the grant_access_to configuration.

Grants are a generic way (not BigQuery specific) to manage access for a set of users, groups, or service accounts on models, seed, or snapshots. Service accounts should be granted the permissions to authorize access to services in your automated pipelines. This will minimize concerns about over-permissioning to users in production environments. Grants have two components:

- Privilege: action to be performed, for example, select

- Grantees: the user, group or service account to which the privilege is granted

If you need more custom/complex configuration than what Grants allow, or need to apply configurations to objects other than tables and views, consider using pre and post hooks in dbt.

6.5 Working with sensitive data

BigQuery provides two options for managing sensitive data: column-level access control and dynamic data masking. You can manage PII by creating column-level access enforceable configurations or policy tags.

Start by identifying what needs to be tagged. Consider generating profiles about your data across all BigQuery objects using Cloud Data Loss Prevention. Cloud DLP is able to report where sensitive and high-risk data reside.

Using BigQuery column-level access control, you can create policies that check whether a user has proper access. Data masking is used to obscure column data, while still allowing users access to the column. You can use data masking in combination with column-level access control, to configure a range of access to data, based on the requirements of different groups of users.

The three important steps to enforcing access control are
1. Create policy taxonomies
2. Define data policy and policy tag
3. Attach policy tag to columns

Steps 1, 2 can be automated using an Infrastructure as code (IaC) service like terraform. We can use dbt to attach the policy tags on models. dbt enables this feature as a column resource property as shown in Code snippet 6.6.1.

```
models:
- name: orders
 columns:
  - name: credit_card_number
    policy_tags:
     - 'projects/<gcp-project>/locations/<location>/taxonomies/<organization>/policyTags/<tag>'
```
Code snippet 6.6.1: setting BigQuery policy tags

6.6 Logging

With dbt Cloud you can view and retain execution and audit logs. Execution logs provide model level execution details. The audit log includes details such as who performed an action, what the action was, and when it was performed.

BigQuery also provides audit logs for insight into operational concerns related to your use of Google Cloud services. It reports resource interactions such as which tables were read from and written to by a given query job. Additionally, audit logs can be streamed to BigQuery to be queried with standard SQL.

Consider consolidating the dbt logs in Cloud Logging. You can do this by fetching the logs from a dbt job, using the dbt Cloud API. Since cloud logging is searchable, this can be a useful tool for the operations team to quickly identify issues across your pipeline.

6.7 Labels and tags

A label is a key value pair that can be added to a resource in BigQuery. Labels are searchable and are a great way to organize tables/models (Figure 6.7.1). Common examples include setting:

- Sensitive data indicator: `contains_pii:yes`
- Refresh Interval: `refresh:daily`
- Model Type: `type:incremental`
- Environment: `environment:production`
- Vertical/Function/Team Label: `team:marketing, vertical:finance`

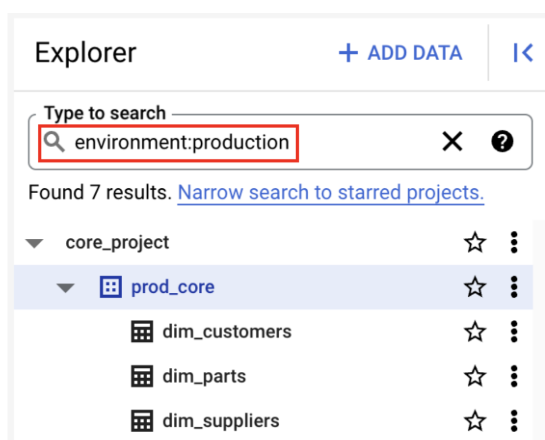


Figure 6.7.1: Labels are searchable in BigQuery Explorer and Dataplex

A tag is a label without any key values. Tags are useful where you are labeling a resource and do not need the key-value format. As an example, if you want to indicate that a table should only be used for testing purposes, you can add a `for_testing_only` tag to it. dbt Models support labels and tags as shown in Code snippet 6.7.1.

```
{{
 config(
  materialized = "table",
  labels = {'contains_pii': '',     # BQ Tag (not to be confused with Policy Tag)
              'refresh': 'daily'},  # BQ Label
  tags=["daily"]                     # dbt Tag (not to be confused with dbt Tag)
 )
}}
```
Code snippet 6.7.1: setting BigQuery label and tag configurations

We recommend pairing dbt's node selection syntax with dbt tags instead of BigQuery labels and tags, to ensure a consistent experience when running subsets of the DAG. Refer to Table 6.7.1 for recommendations on when to use which resource.

BigQuery job labels help with organizing, monitoring performance, and calculating the charges incurred by a group of queries. dbt exposes this functionality through query-comments. It parses the comments and sets them as labels for the executing queries. You can override the default query comment macro and append a wide range of useful audit information to the BigQuery job, enabling you to understand the performance characteristics of your dbt project.

Resource	Common Use Case
Label	Organizing and searching resources in Google Cloud. Use as function specific key:value pairs
Job Label	Aggregate billing per label, Auditing BigQuery jobs
Tag	Organizing and searching resources in Google Cloud. Use as cross function string values
Tag (dbt)	Organizing and searching resources in dbt. Use for running parts of the dbt workflow.
Tag Resource / Policy Tag	Column-level access control and masking

Table 6.7.1: Recommendations on using Label and Tag resources

7. Optimizing performance

In this section we explore a number of practices that should be considered during your BigQuery & dbt development journey. Table 7.1, provides a summary of the best practices.

Link to bq:

Best Practice	Why

Identify Bottlenecks	Look for optimization opportunities
Denormalize models	Improve query performance
Optimizing joins	Improve join performance
Partitioning	Predict guaranteed costs and improve performance
Clustering	Reduce costs (unguaranteed) and Improve query performance
Date Sharded Tables	Avoid complex/depreciated configurations
Merge Behaviour	Enhance incremental models
Materialized Views	Pre compute complex logic (~~experimental~~)
Write modular SQL	Create maintainable, modular and efficient SQL
Avoid custom UDFs	Prevent UDF overhead when possible
Data Caching using BI Engine	Improve query performance in BI tools
Using BigQuery ML and dbt	Use ML packages to increase modularity, velocity and reliability for predictive analysis

Table 7.1: Summary of optimization best practices

7.1 Identify bottlenecks

If you have already built your models and are considering performance optimisations, a good place to start is by observing the Model timing tab in dbt Cloud and looking for bottlenecks. In Figure 7.1.1 we can quickly identify the fct_orders model as a great candidate for optimization.

The timeline statistics can then help you understand whether certain stages of a model/sql dominate resource utilization. This is useful information which can help narrow down the best practices which would be most beneficial in optimizing the performance.

Using a custom query comment macro can also help monitor the performance characteristics of your dbt project by making it easier to identify queries in the BigQuery query plan.

dbt Cloud also generates metadata on the timing, configuration, and freshness of models in your dbt project. The dbt Metadata API is a GraphQL service which supports queries on the metadata, via the graphical explorer or the endpoint itself. Teams can pipe this data into their data warehouse and analyze it like any other data source in a business intelligence platform.

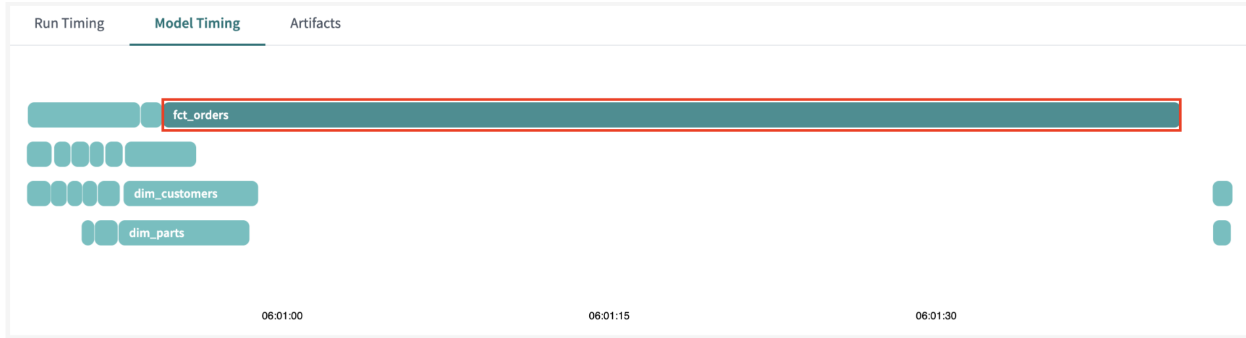


Figure 7.1.1. Job model timing tab

7.2 Optimizing joins

To ensure efficient performance, use the following best practices to avoid anti-query patterns when joining data in BigQuery:

- **Use unique join keys:** Ensure join keys are unique and distinct. Avoid joining on descriptive columns
- **Optimize join order**: When joining a large and a small table, BigQuery creates a broadcast join, which sends the small table to every slot, to process against the large table. Based on this behavior, you can optimize performance by using this logical order in your common table expressions (CTE):
  - Place the largest table first
  - Place the smallest table next
  - Perform subsequent joins in order of decreasing size of the tables
- **Cluster join keys**: Optimize hash data shuffling by co-locating or clustering data. Refer to section 7.3 for more on clustering.
- **Avoid self-joins:** Self-joins square the number of rows, which results in poor performance. Use window (analytic) functions instead.
- **Avoid many-to-many cross joins**: Avoid joins that generate more outputs than inputs. As an alternative, pre-aggregate your data.
- **Unbalanced joins:** Data skew can occur due to uneven table partitions resulting in poor performance. If there is a significant difference between AVG and MAX compute times in the query explain plan, your data is probably skewed. To remedy the issue, consider using the incremental dbt materialization in favor of table/full refresh, ensuring you are pre filtering the data, and reevaluating the table partition type and column. Refer to section 7.3 on choosing the optimal partitions for your tables

7.3 Partitioning

Partitioning allows BigQuery to split a table into multiple segments. Each of these segments can be read selectively, making the data retrieval faster and cheaper. Consider partitioning to:

- Improve query performance and reduce cost on large table joins

- Control costs in your dbt development environment
- Reduce table storage costs

7.3.1 Improve query performance and reduce cost on large table joins

Partitioning can help improve performance when a query requires multiple resource-intensive joins and only a specific range of data is needed. Partitioning only works when data is filtered using literal values: selecting partitions using a subquery or filtering on a non-partitioned column won't improve performance. Approach partitioning with the following recommended steps:

1. Identify and partition upstream dbt models
2. Use a common table expression (CTE) to filter/prune the referenced upstream models
3. Join the filtered tables and perform any data transformations
4. Return the transformed data

Refer to Code snippet 7.3.1.1, on how to write structured and optimized CTEs.

```
with orders as (
  select * from {{ ref('stg_tpch_orders') }}
  where order_date >= DATETIME("2019-01-01")  # filter the referenced partitioned tables early
),
line_item as (
  select * from {{ ref('stg_tpch_line_items') }}
  where commit_date >= DATETIME("2019-01-01") # always filter on the partitioned column
                                              # to prune the partitions
),
final as (
  select
     order_key, ... from orders          # transform and
     inner join line_item               # join the partition pruned tables
        on line_item.order_key = orders.order_key
)
Select                          # return final query expression
  order_key, ...from final
```

Code snippet 7.3.1.1: writing clean & performant CTEs for partition pruning

BigQuery supports three types of partitioning:

- **Time-unit column:** You can partition a table by any column with a data type of TIMESTAMP, DATE, or DATETIME. This strategy is often used for marts models that business users can query.

- **Integer range:** You can partition a table using a range of values on an integer column. This approach is less common, as it can be tricky to configure the correct number of non-skewed buckets without exceeding BigQuery's partition limit (4,000).
- **Ingestion time:** You can partition a table by the ingestion time of the source data.

BigQuery automatically assigns rows to partitions based on the time when BigQuery ingests the data. You can choose hourly, daily, monthly, or yearly granularity for the partitions.

In order to avoid partition skew and consequent performance overhead, do not select partition columns which are at a risk of creating unbalanced partition sizes as data grows.

The dbt partition config can be used to set the partition type and granularity, as shown in Code snippet 7.3.1.2. Use the `require_partition_filter = true` config to require a filter clause on the partitioned column in all queries against a given table. This can help reduce query costs.

```
{{ config(
  materialized='table',
  partition_by={
   "field": "<field name>",
   "data_type": "<timestamp | date | datetime | int64>",
   "granularity": "<hour | day | month | year>"

          # Only required if data_type is "int64"
    "range": {
                    "start": <int>,
                    "end": <int>,
                    "interval": <int>
          }
  },
  require_partition_filter = true
)}}
```

Code snippet 7.3.1.2: dbt model partition configuration

7.3.2 Control costs in your dbt development environment

You can use partitions to reduce the cost of iterating code during development. For example: Consider a table consisting of historical customer orders. The table has ten years of data, with approximately 1 TB of order data per year. Without partitioning, BigQuery will scan all 10 TB of data each time you query that table, at an approximate cost of $50 per query.

Alternatively, you can partition the table by its date column, creating 120 monthly partitions. This allows you to use a WHERE clause to limit scans to individual partitions, as shown in code snippet 7.3.2.1.In the given example, this would reduce the amount of data scanned to 2 partitions or 0.16 TB, bringing cost per query from $50 down to $0.80. This will also reduce job execution time.

```
select *
from {{ source('tpch', 'orders') }}
-- this filter will only apply during a dev run
{% if target.name == 'dev' %}
where created_at >= dateadd('month', -, current_date)
```

```
{% endif %}
```
Code snippet 7.3.2.1: conditional date filter

To apply logic in a DRY (don't repeat yourself) fashion, use a macro that can be called across all models in your project. Code snippet 7.3.1.2 shows how a macro can be used to add the filter clause to models in the development environment.

```
{% macro limit_in_dev(timestamp) %}
{% if target.name == 'dev' %}
    where {{timestamp}} >= dateadd('month', -{{var('development_months_of_data')}}, current_date)
{% endif %}
```
Code snippet 7.3.1.2: a generic date filter macro for reducing development costs

7.3.3 Reduce table storage cost

You can take advantage of cheaper long-term storage by creating partitions. If a partition is not modified for 90 consecutive days, the price of storage drops by 50%.

You can also optimize storage by setting the partition expiration at the table or dataset level. The data will not be queryable after the specified interval and will be eventually deleted. You can set it by using the `partition_expiration_days` config in dbt.

7.4 Clustering

BigQuery supports clustering over both partitioned and non-partitioned tables as another way of improving query performance. Clustering organizes data so that similar values are stored next to each other. Use clustering if:

- Your queries often filter on certain columns. BigQuery eliminates scanning unnecessary data reducing cost and increasing performance. However unlike partitioning, clustering does not provide query cost guarantees
- Your queries often aggregate on certain columns. Performance is improved because of colocation of data
- You require more granularity than partitioning provides. Since partitioning can only be done on one column, you can use clustering on four additional columns.
- Target table size exceeds 1GB. You will not see any benefits to clustering for small tables

You can use the cluster_by config as shown in code snippet 7.4.1, to implement clustering for your dbt models. You can specify up to four clustering columns per model. If you need to cluster on additional columns, consider combining clustering with partitioning. The order of

clustered columns determines the sort order of the data. High cardinality and non-temporal columns are better-suited for clustering.

```
{{
 config(
  materialized = "table",
  cluster_by = ["customer_id", "order_id"],
 )
}}
```

Code snippet 7.4.1: setting the dbt model cluster configuration

7.5 Date sharded tables

Sharding, like partitioning, allows you to limit queries to specific dates by creating multiple tables with a date suffix. You can use wildcards to query across a range of dates in a sharded table.

Partitioned tables perform better than sharded tables. With sharding, BigQuery has to maintain a copy of the schema and metadata for each table, adding to querying overhead. As of dbt version 1.0, sharding has been deprecated in favor of column-based partitioning.

You can convert date-sharded tables to ingestion-time partitioned tables following these instructions.

7.6 Denormalization

Denormalization is the process of adding redundant data to a database, to improve query performance by reducing the need for complex JOIN operations. Consider denormalizing a table when it is larger than 10GB and you require additional optimizations over partitioning and clustering. Hower, keep in mind that querying a large denormalized table may be slower than joining it with a small table, so it is important to weigh the tradeoffs.

One common way to denormalize a database is to create a single wide table by joining a fact table with all of its dimensions.We recommend taking advantage of BigQuery's native support for nested and repeated structures by using a combination of ARRAY and STRUCT data types to define the table schema.

As an example, consider the following orders data:

Order Table

order_key	cust_key	total_price	order_date
1	73100	26602	2023-01-19
2	92861	17680	2023-02-03

Line Item Table

order_key	line_number	quantity	extended_price	ship_date
1	1	3	4081	2023-01-29
1	2	18	22521	2023-01-23
2	1	41	7010	2023-02-05
2	2	27	3288	2023-02-05
2	3	42	7382	2023-02-09

Flatten order-lineitems using joins

order_key	cust_key	total_price	order_date	line_number	quantity	price	ship_date
1	73100	26602	2023-01-19	1	3	4081	2023-01-29
1	73100	26602	2023-01-19	2	18	22521	2023-01-23
2	92861	17680	2023-02-03	1	41	7010	2023-02-05
2	92861	17680	2023-02-03	2	27	3288	2023-02-05
2	92861	17680	2023-02-03	3	42	7382	2023-02-09

Flatten order-line items using nested and repeated fields

order_key	cust_key	total_price	order_date	line_number	quantity	price	ship_date
1	73100	26602	2023-01-19	1	3	4081	2023-01-29
1				2	18	22521	2023-01-23
2	92861	17680	2023-02-03	1	41	7010	2023-02-05
2				2	27	3288	2023-02-05
2				3	42	7382	2023-02-09

Figure 7.6.1: using BigQueries native and repeated structures to denormalize data

Flattening data with nested and repeated fields prevents duplication, maintains the normalized nature of the original data, and boosts performance. However, this structure is only valuable if your downstream models or applications can use it.

7.7 Merge behaviour

The incremental_strategy config controls how dbt builds incremental models. dbt uses a merge statement by default to refresh incremental tables on BigQuery. Incremental strategy can be configured with a **merge** or **insert_overwrite** strategy using the configuration block shown in code snippet 7.7.1.

```
{{
 config(
  materialized = 'incremental',
```

```
  incremental_strategy = '<merge | insert_overwrite>'
  incremental_predicates = ''

  # if merge
  unique_key = 'id',
  cluster_by: ['session_start']

  # if insert_overwrite
  partition_by = {'field': 'session_start', 'data_type': 'timestamp'},
  partitions = ['timestamp(current_date)'] #if static partitions
 )
}}
```

Code snippet 7.7.1: setting the dbt model merge behavior configuration

The merge approach automatically updates late-arriving facts in the destination table, but requires scanning all referenced source tables and the destination table, which can be slow and expensive. Clustering the merge keys can help reduce costs. For more on clustering, refer to section 7.4.

The insert_overwrite strategy replaces entire partitions in the destination table, and requires a partition clause. Dynamic partitions can be determined automatically, but static partitions (which rely on user-supplied configuration) can be more efficient.

The advanced incremental_predicates configuration improves performance for large data volumes, and accepts a list of any valid SQL expression(s).

Possible merge configurations are as follows:

- merge: simple
- merge: clustered keys
- Insert_overwrite: static partitions
- Insert_overwrite: dynamic partitions

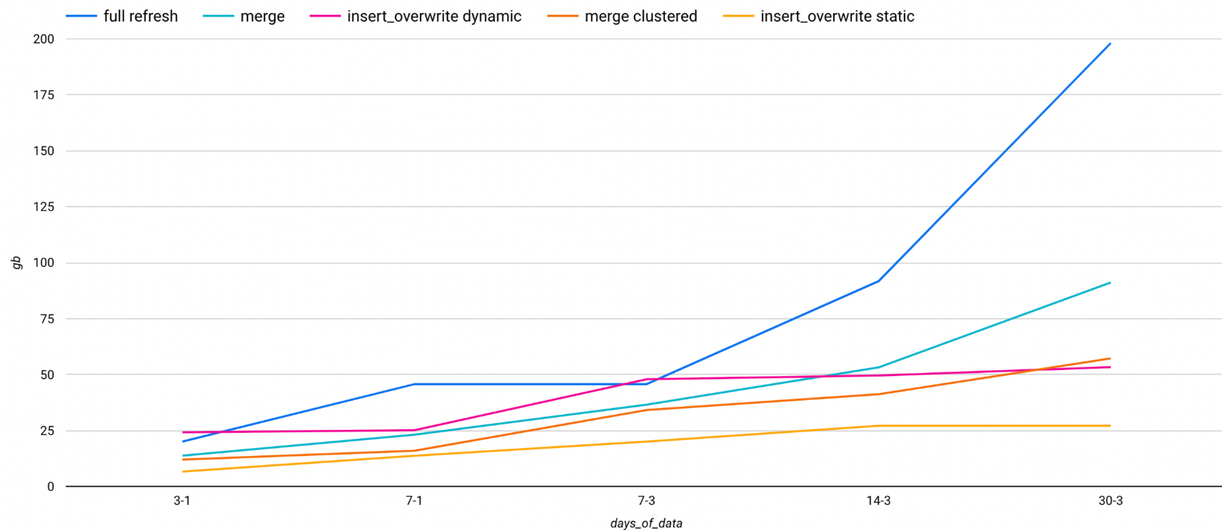


Figure 7.7.1: data processed by BigQuery per incremental merge strategy over increasing number of days

Figure 7.7.1, depicts the average performance for an incremental model with 2.5GB daily partitions over a period of 1-30 days. Static insert_overwrite is most performant, while dynamic insert overwrite strategy is the slowest. Clustered merge is cheaper than simple merge with similar performance. For more on the results refer to : [Benchmarking: Incremental Strategies on BigQuery].

7.8 Materialized views

Materialized views are precomputed views that periodically cache the results of a query for increased performance and efficiency. Consider using materialized views for queries with high computation cost and small dataset results. Processes that benefit from materialized views include online analytical processing (OLAP) operations that require significant processing with predictable and repeated queries.

dbt currently supports [Materialized views] as an experimental feature, with official support coming this year. If you plan on using this feature, experiment with the configuration, and plan for the upcoming release.

7.9 Writing effective SQL using DRY principles

BigQuery uses a columnar format to store data, so the number of columns retrieved from a query impacts performance. Best practice is to *select only the columns you need* by explicitly stating the column names and *avoiding using select all statements*.

Applying filters early allows BigQuery to perform aggregations and joins on a smaller subset of data improving performance. Consider making your pipeline more modular, by splitting your SQL into multiple models or filtering early in your CTE.

Use the dbt_utils package to apply DRY (don't repeat yourself) principles to your data models. This package contains macros and tests that can address common data modeling pain points and expedite tasks like creating a date spine, unpivoting columns, and more.

As your team and data grow, you may want to create and share internal packages to standardize logic and definitions across multiple dbt repositories. This can help limit redundant code, and keep business logic and metric definition consistent.

7.10 Avoid custom UDFs

A user-defined function (UDF) lets you create a function in BigQuery using a SQL expression or JavaScript code.
We recommend using dbt macros instead of custom UDFs where possible. Javascript UDFs provide greater flexibility, but can significantly slow down query processing time due to the spin-up overhead required for their instantiation.

The order of performance from highest to lowest is:

- dbt macro
- native (SQL) UDF
- javascript UDF

7.11 Cache queries using BI engine

BI Engine accelerates SQL queries in BigQuery by intelligently caching the data you access most frequently. Reservations manage memory allocation at the project level. You can use the BigQuery Admin Console to reserve up-to 10GB of memory per table, or 250GB when compressed per project, per location for in memory processing. You can partition larger tables to avoid exceeding quota limits.

BI engine is a great way to get extra performance from BigQuery if

- You want to improve the performance of BI tools connected to BigQuery
- Your pipeline includes designated tables that are queried frequently, such as reference or dimension tables

7.12 Using Bigquery ML and dbt

dbt can optimize BigQuery ML models with feature reuse, low overhead, and reliable implementation. It simplifies upstream data and handles dependencies across various steps in the machine learning workflow, including feature engineering, model training, and predicting.

- Feature Engineering is the process of transforming input data into a format that can be efficiently used for machine learning. We recommend making this process as repeatable as possible, by using dbt macros to apply the same transformations to input and prediction data. Also consider using the dbt-ml-preprocessing package to automate data preprocessing tasks and standardize data sets.
- Training & Predicting: The dbt machine learning package allows users to train, audit and use BigQuery ML models, from a select statement and a set of parameters. It also provides helper macros that assist with model audit and prediction

You probably don't want to retrain your models every time the dbt workflow runs, as this can be expensive and time-consuming. We recommend setting the 'enabled' configuration in dbt to an environment variable flag, to retrain the models selectively.

dbt now supports BigQuery python models, starting from v1.3. This enables users to execute Python as PySpark jobs via Google Dataproc service. The Python/PySpark code can read from tables and views in BigQuery, perform all computation in Dataproc, and write the final result back to BigQuery as part of the dbt workflow.This allows for advanced ML and predictive analysis using Python, for use cases beyond BigQuery ML capabilities.

8. Billing & resource management

BigQuery automatically allocates computing resources as you need them. You can reserve compute capacity ahead of time in the form of slots (virtual CPUs) to save costs. In this section we will look at how to monitor resource consumption and optimize billing.

BigQuery provides the following billing plans:

- Free Usage tier: As part of the Google Cloud Free Tier, BigQuery offers some resources free of charge up to a specific limit. These free usage limits are available during and after the free trial. If you go over these usage limits and are no longer in the free trial period, you will be charged according to the pricing on this page.

- On-Demand: You are charged for the number of bytes processed by each query. The first 1 TB of query data processed per month is free

On demand pricing can access up to 2000 slots to execute queries. However as the slots are shared among all queries in a single project,the queries are executed in a best effort pattern . This is the default billing plan.

- Editions: BigQuery provides three editions (Standard, Enterprise, and Enterprise Plus) to support different types of workloads. Each edition provides a set of capabilities at a different price point to match the requirements of different types of customers. You can create a reservation or a capacity commitment associated with an edition. All editions support autoscaling slots by default.

When it comes to storage pricing, BigQuery provides a transparent and flexible model, as outlined in this documentation. The pricing structure is based on two types of storage: active storage and long-term storage. Active storage, designed for frequently accessed data, incurs higher costs, while long-term storage is ideal for less frequently accessed information and comes at a lower cost.

By default, BigQuery calculates storage usage in logical bytes for billing purposes. However, when creating datasets using SQL or the BigQuery API, you can opt to be billed based on physical bytes instead. This enables a tailored billing model that aligns with specific requirements. Moreover, BigQuery facilitates a seamless transition for existing datasets to switch to physical bytes for billing, which can help you optimize your storage strategy and costs within the BigQuery ecosystem.

It is recommended that you start your project with on-demand billing, unless you have prior indication of resource usage. Monitor the billing for a set period of time, identify usage patterns and switch to a more suited billing plan. Use the following three steps to build a billing strategy:

8.1 Estimate

Start by estimating the cost of running BigQuery for your department or organization. You can use the BigQuery pricing estimator to find out the execution and storage costs for your project. This is also a great time to start incorporating best practices:

- Estimate query cost before running large queries: When you enter a query in the Google Cloud console, the query validator provides an estimate of the number of bytes read. You can use this estimate to calculate cost in the Pricing Calculator
- Set billing limit in dbt: Limit query costs by setting the maximum byte billed flag in dbt. Note that if this limit is exceeded, the executing query and job will fail
- Set billing alerts and budgets: By setting alerts and budgets you can avoid surprises and monitor all charges in one place

8.2 Monitor

There are three common approaches for monitoring spend:

- Export billing data to BigQuery and use your reporting tool of choice to build a billing dashboard
- Visualize spend over time with Looker Studio
- Use query labels to categorize and calculate job spend

It is also important to monitor slot usage trends over time. To access slot usage, head over to Cloud Monitoring metrics explorer and search for *Slots used by project, reservation, and job type*. You can also use the billing dashboard, INFORMATION_SCHEMA.JOBS* views, Cloud Logging, the Jobs API, or BigQuery Audit logs to check how many slots are being used by your projects.

If you have identified projects or users with unexpectedly high usage in the metrics explorer, setting custom quotas Is a great way to control costs. Project quotas aggregate usage of all users, while user quota is enforced individually for all users of the project. See Managing your quota using the Google Cloud console.

8.3 Optimize

With on-demand pricing, you only pay for the queries you run, making it a good option for when you need to run occasional queries.

The capacity-based analysis pricing model offers predictable cost for queries. This is ideal for running regular queries or if you need dedicated processing capacity. BigQuery provides three editions (Standard, Enterprise, and Enterprise Plus) to support different types of workloads.

Each edition provides a set of capabilities at a different price point to match the requirements of different types of customers. You can create a reservation or a capacity commitment associated with an edition. Capacity commitments are not required to purchase slots, but can save on costs. The commitment plans are a good option if you are looking to save money by committing to a fixed amount of resources for a specified period.

The general recommendation is to use Standard editions for trail and test projects, Enterprise plus is when you need features like CMEK (customer managed encryption keys) and advanced features like BQML otherwise Enterprise should work for most of the use cases.

As your workload increases, BigQuery dynamically adjusts your slots so that you only pay for what you use as all the Editions support Autoscaling slots. While Enterprise and Enterprise plus lets you set the baseline slots, which is the minimum number of slots that will always be allocated to the reservation, standard doesn't support baseline.

At any given time, some slots might be idle. This can include:

- Slot commitments that are not allocated to any reservation.
- Slots that are allocated to a reservation baseline but aren't currently in use.

By default, queries running in a reservation automatically use idle slots from other reservations within the same administration project as long as they use the same edition as You cannot share idle slots between reservations of different editions. You can share only the baseline slots or committed slots.

In some use-cases where performance is a priority, it may be beneficial to identify the business vertical responsible for the sporadic usage and provision a separate on-demand project or create a reservation with standard edition.

9. Appendix

9.1 Key concepts and terminology

- BigQuery Project: Grouping of all your Google Cloud resources
- Dataset: Also known as a schema in some databases, Datasets are top-level containers that are used to organize and control access to your tables and views.
- Partitioned Table: Dividing a large table into smaller time or range based partitions, query performance and costs can be controlled by reducing the number of bytes read by a query.
- Clustered Table: Columns are used to colocate data. This helps with faster retrieval data and better query performance.
- External Table: Tables that are created, backed by storage that is external to Bigquery
- Temporary Table: Restricted tables used to cache query results and exist only up till 24 hours.
- Materialized View: Precomputed views that periodically cache the results of a query for increased performance and efficiency.
- Authorized View: Lets you share query results without giving users access to the underlying tables
- Slots: A virtual CPU used by BigQuery to execute SQL queries. BigQuery automatically calculates how many slots each query requires, depending on query size and complexity
- Reservations: After you purchase slots, you can assign them to different buckets, called reservations. Reservations let you allocate the slots in ways that make sense for your particular organization.
- Commitments: A capacity commitment is a purchase of BigQuery compute capacity for some minimum duration of time. Commitments are measured in BigQuery slots, which are a unit of computational capacity.
- dbt Project: Grouping of all your dbt resources (Models, Macros, Tests)

- Model: sql or py files where the transformation logic resides in dbt
- dbt DAG: a visual representation of your data models and their connection to each other.
- Jinja: A templating language used by dbt to implement control structures and more
- Packages: Standalone dbt projects, with models and macros that tackle a specific problem area
- Macros: pieces of code that can be reused multiple times – are analogous to "functions" in other programming languages
- dbt Docs: generate documentation for dbt projects and render it as a website
- dbt Tests: ensure data models are of good quality and meet your assertions
- dbt Deployment: promote and deploy code to production and other environments
- dbt Semantic Layer: centrally defined business metrics in the modeling layer for less duplicative coding and more consistency for data consumers