Best Practices for Leveraging Amazon Redshift and dbt™

aws

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided "as is" without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

Data has become an essential part of every business, and its volume, velocity, and variety continue to increase. More and more organizations have found it challenging to meet their needs with traditional on-premises data analytics solutions, and are looking at modernizing their data and analytics infrastructure by moving to the cloud.

To best leverage their investment in a cloud data infrastructure, companies need to work in a way that enables collaboration with business users and automation of key workflows. For this reason, many data teams use dbt™ to manage data transformations in Amazon Redshift. This allows teams to create trusted data pipelines that are version controlled, tested, and documented, thus increasing stakeholder trust and removing common technical barriers that limit time to production. This whitepaper outlines the benefits and best practices of using the scale of Amazon Redshift data warehouse and the agility of dbt™ to build trusted data pipelines.

Introduction

Amazon Web Services (AWS) is the world's most comprehensive and broadly adopted cloud platform, offering over 200 fully-featured services from data centers worldwide. Millions of customers—including the fastest-growing startups, largest enterprises, and leading government agencies—trust AWS to power their infrastructure, improve their agility, and lower their costs.

Amazon Redshift is a fast, easy and widely used cloud data warehouse that makes it simple and cost-effective to analyze data efficiently using existing business intelligence tools. It is optimized for data sets ranging from a few hundred gigabytes to petabytes or more, and is designed to cost less than other data warehousing solutions.

dbt helps to manage data transformation on the data platform by enabling teams to deploy analytics code following software engineering best practices such as modularity, continuous integration and continuous deployment (CI/CD), and embedded documentation. dbt is completely pushdown, utilizing the database engine compute capabilities for execution. More specifically, dbt allows data teams to implement business logic in SQL in dbt data models, create and automate code-based tests, and version-control their code. At its core, dbt will:

- Compile your project's code into Amazon Redshift DML, and DDL SQL statements

- Infer dependencies to run transformations (in the form of dbt models) in the correct order and generate a project DAG (Directed Acyclic Graph) automatically

- Facilitate reusable code with macros and ref statements.

To get started with dbt and Amazon Redshift, follow the getting started guide. dbt Labs also offers a variety of on-demand training courses to help data teams understand the power and functionality of dbt.

Amazon Redshift Deployment Options

Amazon Redshift is a fast, simple, cost-effective data warehousing solution that enables customers to scale their data analytics. Tens of thousands of customers rely on Amazon Redshift to analyze exabytes of data and run complex analytical queries, making it one of the most widely used cloud data warehouses today. Amazon Redshift is a fully managed service and offers both provisioned and serverless options.

Redshift Serverless

Amazon Redshift Serverless makes it easy to run and scale analytics in seconds without the need to set up and manage data warehouse infrastructure. With Redshift Serverless, your data warehouse service can auto pause, auto resume, and auto scale up and down. You simply set the base data warehouse Redshift Processing Units (RPU) capacity, based on your price and performance requirements. You can create multiple workgroups, and namespaces in Redshift Serverless to allow you for workload isolation or segregate workload based on data domains.

Spinning up Amazon Redshift Service takes minutes, with instructions for the two options provided below. Find instructions on setting up Redshift Serverless data warehouse here and Redshift Provisioned Data warehouse here. Follow instructions provided here to Connect to Serverless and here to connect to Provisioned to start querying Amazon Redshift.

Provisioned Clusters

When using the provisioned deployment option, your Amazon Redshift environment consists of a cluster of nodes of a certain node type. While creating your data warehouse, you can choose the right node type and quantity to meet your requirements. A provisioned cluster is most suited for workloads where you need granular control over the configuration, deployment, and management. You can scale your cluster as needed using the resize option, and leave administrative tasks like back-up, node or drive failure management, and query tuning to Amazon Redshift service. With Redshift, any user—including data analysts, developers, business professionals, and data scientists—can get insights from data by simply loading and querying data in the data warehouse.

How Amazon Redshift works

Amazon Redshift uses SQL to analyze structured and semi-structured data across data warehouses, operational databases, and data lakes, using AWS-designed hardware and machine learning to deliver the best price performance at any scale.

Amazon Redshift achieves efficient storage and optimum query performance through a combination of massively parallel processing, columnar data storage, and efficient, targeted data compression encoding schemes.

Use cases for Amazon Redshift

Popular use cases for Amazon Redshift include:

Business Intelligence

Analyze terabytes to petabytes of structured and semi-structured data using SQL and power queries, reports, and dashboards.

Innovation through machine learning

Drive faster decision-making across your business, with predictive insights to inform new products and services using Amazon Redshift ML.

Data sharing and collaboration

Query, share, and analyze live data among organizations and partners with Amazon Redshift Data Sharing and AWS Data Exchange.

Real-time operational insights

Process high volumes of data from multiple sources including streaming sources simultaneously, with unlimited concurrent users and queries, all with consistent high performance.

Typical data warehouse flow for Amazon Redshift
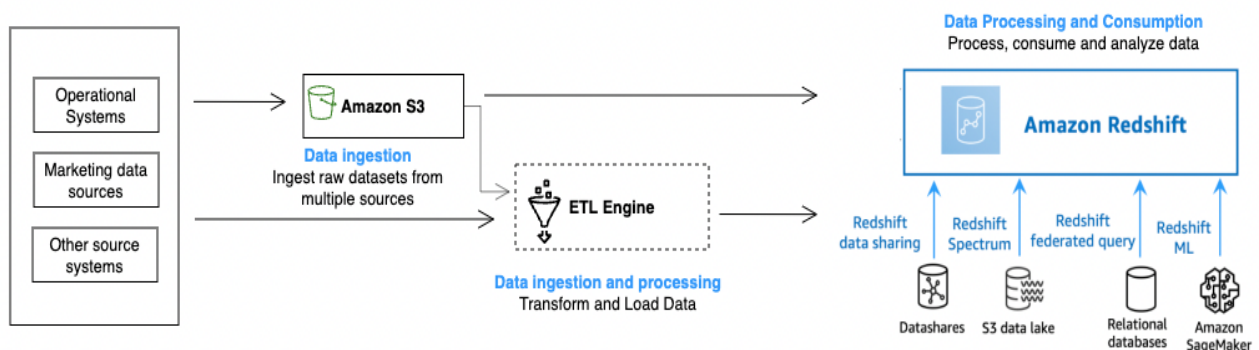
An Amazon Redshift data warehouse is an enterprise-class relational database management system (RDBMS).

At the ***data ingestion*** **layer**, different types of data sources continuously upload structured, semi-structured, or unstructured data to the data storage layer. This data storage area serves as a staging area that stores data in different states of consumption readiness. An example of storage might be an Amazon Simple Storage Service (Amazon S3) bucket.

At the optional ***data processing*** layer, the source data goes through preprocessing, validation, and transformation using extract, transform, load (ETL). These raw datasets are then refined by using ETL operations. This layer can also read sources and write to Amazon Redshift Service directly. An example of an ETL engine is AWS Glue.

At the ***data consumption* layer**, data is loaded into your Amazon Redshift cluster, where you can run analytical workloads. BI tools like Amazon QuickSight are used to build Reporting and Dashboards on top of Amazon Redshift Data warehouse to provide insights to the business community.

Amazon Redshift can also consume data for analytical workloads from many sources:

- **Amazon Redshift *data sharing*** can be used to access datashares from other Amazon Redshift environments, either provisioned or serverless, for read purposes securely and easily. You can share data at different levels, such as databases, schemas, tables, views (including regular, late-binding, and materialized views), and SQL user-defined functions (UDFs). For more information about data sharing, see Getting started accessing data in other Amazon Redshift clusters.

- **Amazon Redshift Spectrum** can be used to query data in Amazon S3 files without having to load the data into Amazon Redshift tables. Amazon Redshift provides SQL capability designed for fast online analytical processing (OLAP) of very large datasets that are stored in both Amazon Redshift clusters and Amazon S3 data lakes. When accessing your S3 data lake via Amazon Redshift Serverless, you do not pay for Amazon Redshift Spectrum separately. You have a unified serverless experience and pay for data lake queries also in RPU-seconds. For more information about Redshift Spectrum, see Getting started querying your data lake.

- **Amazon Redshift federated query** can be used to join data from relational databases, such as Amazon Relational Database Service (Amazon RDS) and Amazon Aurora, or Amazon S3, with data in your Amazon Redshift database. You can use Amazon Redshift to query operational data directly (without moving it), apply transformations, and insert data into your Amazon Redshift tables. For more information about federated queries, see Getting started querying data on remote data sources.

Key Concepts for Understanding dbt™

dbt Architecture and Benefits

dbt is a transformation workflow that lets teams quickly and collaboratively deploy analytics code following software engineering best practices such as modularity, CI/CD, and documentation.

Users will write their business logic with just one SQL select statement or Python DataFrame (not available on Redshift) as a dbt model. At time of execution, dbt compiles the model and takes care of materialization and writing the DML/DDL needed to manage transactions, dropping tables, and managing schema changes. Code is reusable and modular and can be referenced in subsequent dbt models, especially with the use of macros, hooks, and package management.

By having dbt take care of the DML/DDL, environment management is as simple as running the dbt model in the desired environment, thus removing common errors that happen from copy and pasting SQL. Version control allows for mature source control processes like branching, pull requests, and code

reviews. To increase stakeholder trust, data quality tests can be written and applied on the underlying data, catching any edge cases that might apply in development, testing, and deployment.

dbt Cloud

dbt Cloud is a hosted service that helps data teams productionize dbt deployments. dbt Cloud offers turnkey support for job scheduling, CI/CD integrations, serving documentation, native git integrations, monitoring and alerting, and an Integrated Developer Environment (IDE) all within a web-based user interface (UI).

To get started with dbt and Redshift, you can install the Redshift adapter for local development on the command line or create a free trial dbt Cloud account to leverage the in-platform IDE and scheduler.

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.

In dbt, Models, sources, tests, and snapshots are all examples of nodes in dbt projects. 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 means you can run based on what models have changed from the last run or restart from failure alongside running models in parallel.
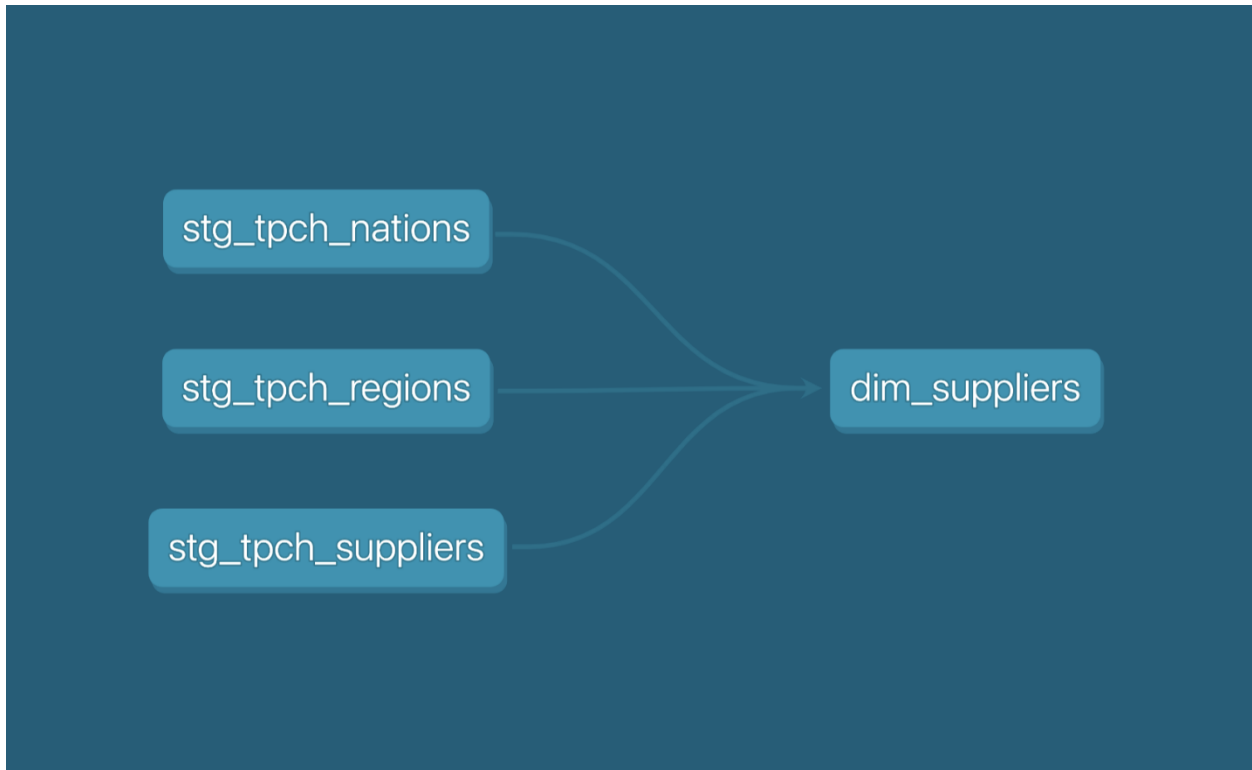
A sample dbt DAG: In this example, the user has declared in model dim_suppliers a reference to stg_tpch_nations, stg_tpch_regions, and stg_tpch_suppliers. At time of a dbt run, if a user has declared 3 threads, dbt would know to execute the first three staging models prior to running dim_suppliers. By 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.

Sources work similarly to refs, with the key difference 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 opens the door to more extensive project testing and documentation involving your source data.

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

Making Code Modular Using Macros

In dbt, alongside SQL, you can also 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 the dbt project are called **macros**. 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. In addition to being helpful for environmental logic, macros can help operationalize Redshift administrative tasks like grant statements, or systemically remove deprecated objects. Basically any code you find yourself writing/executing over and over, you should consider putting it into a macro.

Jumpstarting dbt development with Packages

dbt packages are libraries of open source models and/or macros for a particular purpose or data source. Instead of having to rewrite the same code that the community have perfected, use packages so you can focus on implementing your organization's custom models and business logic.

Common package functionality includes:

- Modeling methods for common data sources, like Facebook Ads or Netsuite

- Useful tests that fall outside of dbt's standard tests

- Innovative work on pushing dbt transformations to new realms, like dbt_ml

- And more!

dbt Labs currently supports a Redshift package for a provisioned cluster that provides a number of models for Redshift system tables, as well as some useful macros that help you compress tables, unload tables, and even vacuum and analyze tables. These are incredibly helpful if you want to access and analyze Redshift query and permissions data without having to explicitly model it out yourself.

Users can add hub packages to their dbt projects by adding the package name and version they want in their dbt project's `packages.yml` file. After users add them there and run `dbt deps`, they can access the macros and models throughout their project using the normal `ref` and macro syntax. For private, local, or git packages not in the dbt package hub, you can learn more about installing packages to your dbt project here.

Add the dbt_utils package to any dbt project and utilizing the macros there to help stick to DRY (Don't Repeat Yourself) principles in data modeling. `dbt_utils` contains a series of macros and tests that help alleviate common pain points in data modeling, such as the ability to quickly create a date spine (a

table with the relevant time period that has one record for each date in the period), [unpivot columns easily](#), and more.

The `dbt_utils` package also supports an `insert_by_period` materialization type for Redshift, where dbt can insert records into a table one period (i.e. day, week) at a time. This materialization is most helpful for event data that can be processed in discrete periods or when you have incredibly large data loads that fail at specific sections.

As your team and data grows, it is not uncommon for teams to create and share internal packages where they can standardize logic and definitions across multiple dbt repositories. This can help ensure metrics have the same definition, business logic is written consistently, and limit WET (write every time) code.

Documentation as you code with dbt

dbt allows data teams to create robust documentation for their data sources, models, tests, and downstream [exposures](#). At the source and model-level, you can add documentation around models themselves and the fields they contain. dbt docs help everyone in your organization, including business users to answer everyday questions about datasets, and transformation logic. Documenting sources and models locally [via the command line](#). Alternatively, generating documentation through dbt Cloud is a great way to promote data transparency with end business users.

dbt also recommends leveraging [meta fields](#) for sources, models, tests, and macros to help establish clear ownership for dbt project objects. The meta fields are a great way to clearly identify model owner, maturity, and flags if it contains sensitive data like PII. There are several data governance tools, that integrate with dbt by using the `meta` field to implement policies.

[Exposures](#) can be used to represent downstream BI dashboards, applications, or data science pipelines in your dbt project. Documenting exposures in dbt is a great way to clearly identify ownership, dependencies, and descriptions for downstream applications.

All of this dbt documentation is available through a local host after running two CLI commands (`dbt docs generate && dbt docs serve`) or through the "Documentation" section in dbt Cloud. You can also use the dbt Cloud Metadata API to send over the information over to our partner data catalogs.

Resources for Learning More

There is a wide variety of resources publicly available for learning more. If you're new to dbt, start with the [Guide](#) tailored for Redshift users. It will guide through the process of setting up a dbt Redshift project and trying out key dbt features. If you want to jump start using dbt CLI with Amazon Redshift you can use this [Guide](#). Next, try the more in-depth [dbt Fundamental course](#), and use the [essential dbt project checklist](#) to reference a list of best practices.

Optimizing an Amazon Redshift Architecture with dbt
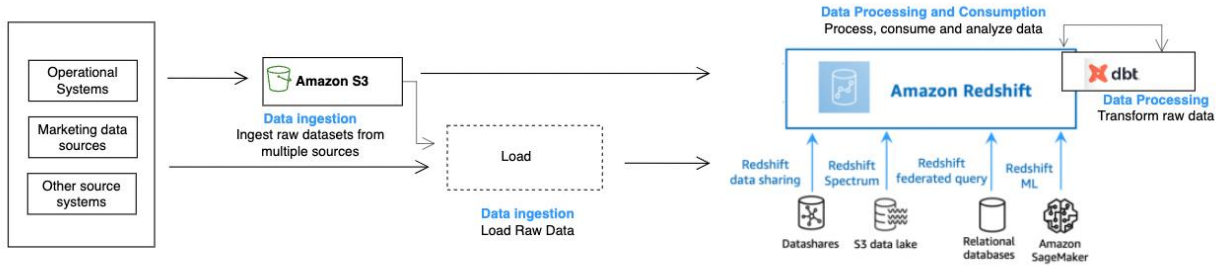
By introducing dbt to an Amazon Redshift architecture, the data team can move data processing inside Redshift, using an ELT (Extract-Transform-Load) framework instead of an ETL (Extract-Load-Transform) framework. In an ELT framework, data is loaded into the data warehouse with little to no pre-processing, and data can be transformed within Redshift. With a shift to an ELT framework, a wider range of people within the organization can contribute data transformations—including practitioners who may have deep business context but no programming skills beyond SQL.

Connecting dbt to Amazon Redshift

The dbt framework uses adapters to "adapt" dbt's functionality to a given data platform. The dbt-Redshift adapter allows dbt to connect to Redshift. After a connection is established, dbt is able to issue Redshift-compliant SQL to Redshift for execution, and conversion of model files into persisted objects.

To use dbt Core with Redshift, the setup process starts with installing the dbt-redshift adapter locally. Once configured with your preferred set of tooling (including text editor, command line interface, etc), you are ready to use dbt with Redshift.

In dbt Cloud, there is no need for installation. Simply create an account, set up a Redshift connection, and you're ready to develop. dbt Cloud is a completely browser-based interface, saving you time on maintaining environments and software updates.

dbt + Redshift Best Practices

Choose A Redshift Cluster or Serverless Endpoint

In Redshift, teams can choose from either an Amazon Redshift provisioned cluster or serverless endpoint. If leveraging a provisioned cluster, we recommend choosing the RA3 node type. Redshift RA3 node types and serverless endpoints separate compute from storage by leveraging Redshift Managed Storage (RMS). They provide superior performance along with features like data sharing, scalable compute and storage, AWS Data Exchange, and cross-database queries. Amazon Redshift serverless has the added benefit where users do not have to worry about sizing aspects and the data warehouse endpoint will automatically pause and resume allowing users to enjoy a pay-as-you-go model. If a customer expects a near zero-administrative cloud data warehouse, then Redshift Serverless is the right choice.

In addition, Amazon Redshift provides a dense compute (DC2) node type, however, we do not recommend them due to their limitations with cross-database querying, and the impact they have on environment configurations and performance.

The correct Redshift set-up and workflow can look different from team to team, however, we have a series of recommendations to build a strong foundation, regardless of organization size. We ultimately recommend implementing this basic structure and then customizing it to work for you and your team.

Plan for Quality Assurance

It is critical to set up distinct production and development environments in dbt and Redshift. This reduces the risk of exposing untested data to business intelligence (BI) platforms, and limits the need for costly rebuilds.

It is also important to create a QA framework. For teams who can access the same raw data in development and production, because the data is the same, you might execute QA-related tasks in your development environment prior to promoting to production. For other teams who might be accessing different versions of the raw data based on if they're in development vs the production environment, a separate QA environment from dev and prod might make sense. This especially makes sense when you are introducing new approvers in the QA environment as opposed to the dev environment.

We always recommend a QA workflow in order to promote code from development to production, irrespective of where you actually execute those checks. For the purpose of this writeup, we will have the simpler approach of managing QA workflows inside of the development environment.

Configure your Data Warehouse Environment (Multiple Redshift Clusters)

To begin, create three Redshift clusters or serverless endpoints named *raw*, *dev* and *prod*. If you choose to create them in separate Redshift clusters, you will need to enable data sharing (available only on Redshift RA3 and Serverless). Data sharing enables one cluster to share a dataset with another cluster without copying or moving data. As an alternative, you can utilize [cross-database querying](#) to create all three databases in one cluster or endpoint using RA3 instances or serverless.

Raw Database

To avoid duplicating the raw data used by your dbt models, we recommend housing raw data in a separate database accessible to both development and production environments. The exception to this is if you have PHI or PII data that cannot be exposed to development environments. If you come across this need, we recommend placing the cleaned version of the raw data in your dev database while having PHI/PII data in prod. The goal here is to minimize how much duplicate data you have while still meeting data security requirements.
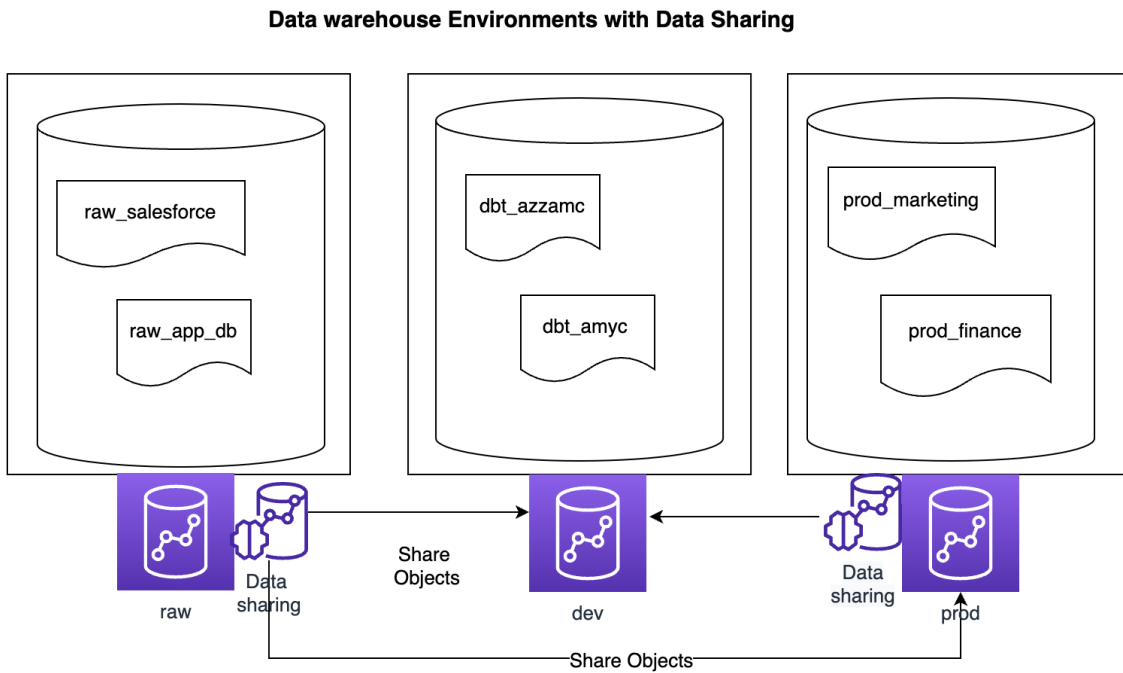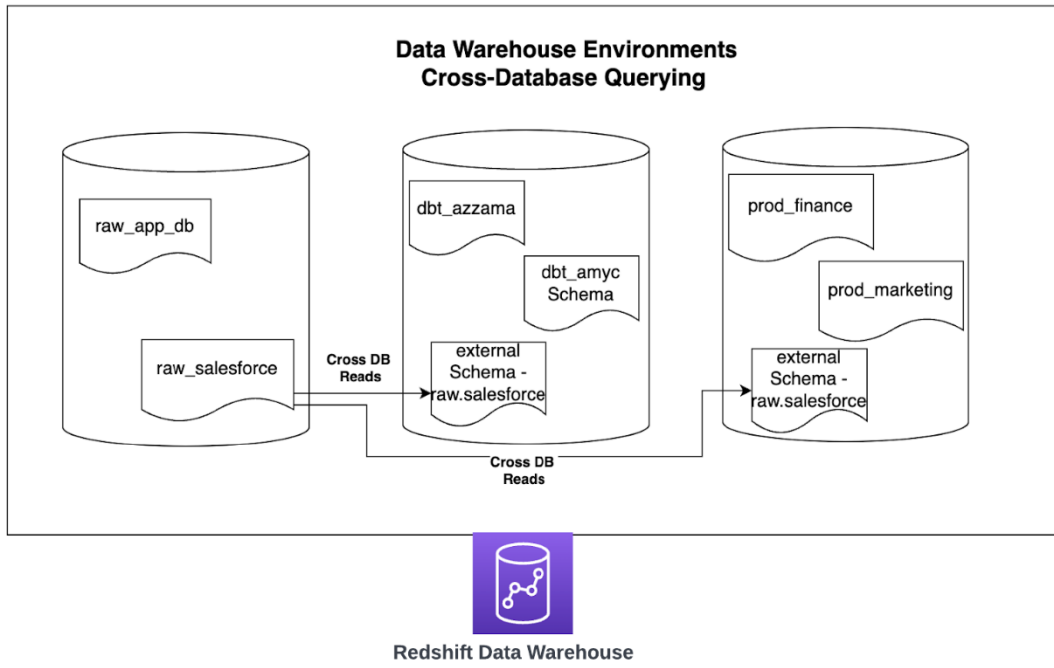
Once the raw database is created and data is ready to be loaded, create `raw` schemas for the unmodeled data from each data source. For raw source data, we suggest naming schema objects with a prefix of the ELT tool that loaded that data (ex. `airbyte_facebook_ads`).

Dev Database
In the dev database, we suggest that dbt developers have their own sandbox/development schema, prefixed with their name (ex. `dbt_amyc`), for development. In this space, dbt developers can create, update, and delete models without the risk of impacting others' development spaces.

Prod Database

In the prod database, we suggest organizing your production data objects by functional or verticals. To do this, you can put your dbt models into different production schemas. We recommend using [custom databases/schemas](#). For example, you would call the schema that holds your production marketing models `marketing`. If it's a core model needed by multiple teams, you can name the schema `core`.

Data Warehouse Environments
Cross-Database Querying

Redshift Data Warehouse

Data warehouse Environments with Data Sharing

Configure your Data Warehouse Environment (Single Redshift Cluster or endpoint)

Alternatively, you can choose to deploy only one Redshift Cluster or endpoint, made up of Redshift DC2 with three schemas or RA3 nodes with multiple databases or a serverless endpoint with multiple databases, and use it to host all your environments. To do this, for dc2 create one database with schemas as your organizational dividers between environments, for example, `raw_<data_source_name>` for raw schemas, `dbt_<developer_name>` for development schemas. Prefix your production schemas based on the function with a `prod_` statement to indicate that they are production schemas. For example, you would call the schema that holds your production marketing models `prod_marketing`.

Data Warehouse Environments on one Redshift Data Warehouse

Redshift Data Warehouse
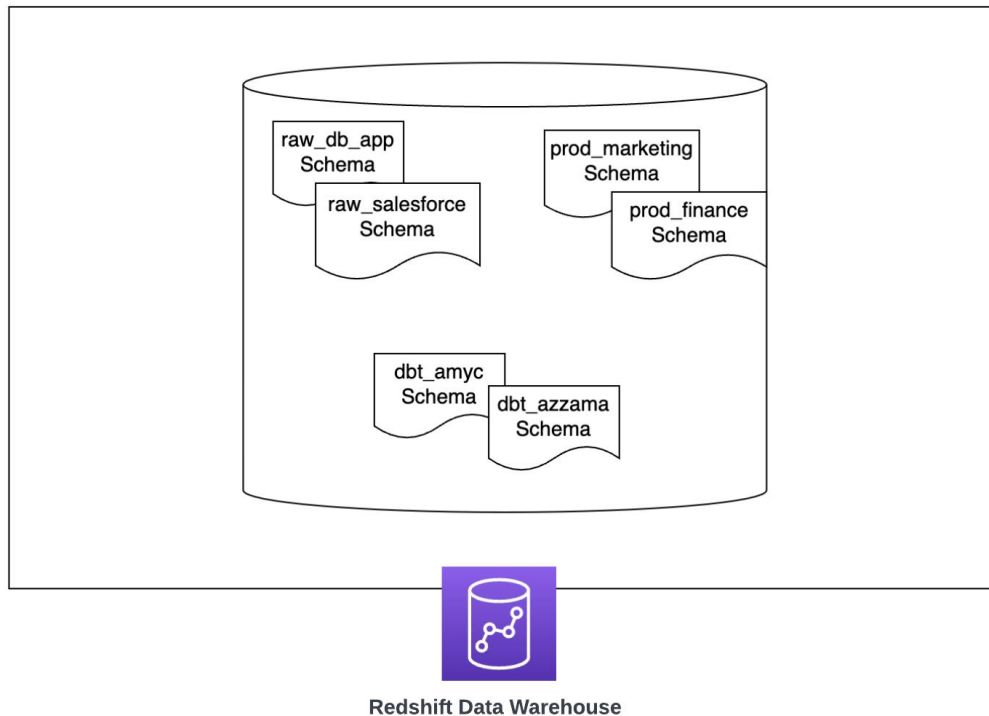
Environments in data transformation are not as siloed as those in general software development. When building data pipelines, there may be a need to reference objects from a prod environment during development or while conducting quality assurance checks. For example, if you are [deferring](#) to an existing object in a prod environment during a dbt Cloud [Slim CI](#) run, you will execute what has changed rather than rebuild the entire project to save on warehouse costs. By deferring, you will select from the

existing object rather than create the object again. A Slim CI run is a dbt Cloud job that allows you to use the state:modified node selector to compare the manifest of the last successful production run to the CI run being executed to see what has changed in the CI run.

Define your Environments as Targets in dbt

To declare environments in dbt, you can use different connection details in targets. [Targets](#) are a dbt property that contain information about your connection to your data warehouse. You can connect your dbt project to Redshift in multiple ways which we will cover later in this paper.t its core, a database name, hostname, username, and port are required to establish a target.

Set up Role Based Access Controls (RBAC), if necessary

By using [role-based access control (RBAC)](#) to manage database privileges in Amazon Redshift, you can assign different privileges to different roles and assign these roles to different users to have more granular control of user access. 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.

Use Redshift's Late Binding Views for Downstream Consolidation

In Redshift, you can create views that are unbounded from their dependencies.This means that if an upstream view or table is dropped with a cascade, the late binding view will not be dropped. Late binding views are useful when a table needs to be dropped and replaced. We recommend using late binding views to build views on external tables.

To physicalize or materialize a late binding view in your project, specify the configuration for dbt at the model or project-level. In a model file, you can explicitly set the materialization for a late-binding view using a [config block](#) like this:

```
{{ config(materialized='view', bind=False) }}
```

You may also want to set the configuration at a project or subfolder-level. To do this, add (and modify the hierarchy) a `bind` variable to your `dbt_project.yml` file. Please see the example below for how to do this.

```
name:

'my_dbt_project'

version: 1.0

…
```

```
#This will make all view models in your project use the late-binding
#modifier. The `bind` configuration can be specified anywhere in the
#configuration hierarchy

models:

     bind: false
```

Performance Tuning your dbt - Amazon Redshift Environment

dbt and Redshift have multiple strategies to make your models more performant and therefore more cost-efficient. These include:

- Using sort and distribution keys on large tables

- Storing data in an efficient manner

- Configuring Amazon Redshift workload management (WLM)

- Choosing appropriate model materializations

- Limiting the amount of data used during development periods

- Implementing dbt packages to stick to DRY practices; and

- Leveraging Redshift Spectrum for large datasets.

Reference to Amazon Redshift best practices for performance tuning: Blog. We discuss each of these strategies below.

Using Sort and Distribution Keys

As your dbt project and datasets expand, your build times will also start to increase. Luckily, Redshift has sort and distribution (dist) keys to optimize performance. When used correctly, you can dramatically improve build times and cost. Paired with dbt's modular modeling mentality and accessible configuration, using sort and dist keys is a matter of adding a few lines to a model `config` block.

Sort keys determine the order in which rows in a table are stored. You can declare any one or more columns on your table as sort keys and Redshift will sort the table accordingly. During processing, metadata like the minimum and maximum values is generated which then Redshift can access these values without iterating over the data during query execution. *Using the right sort key can help Redshift determine how many rows it has to scan and provide a faster query*. If your queries contain range restricted predicates, defining sort key on predicate columns can make queries more efficient. You can

define compound or interleaved sort keys. Compound sort key is more efficient when query predicates use a prefix which is a subset of the sort key columns in order. Interleaved sort key gives equal weight to each column in the sort key.

Distribution keys determine how the rows of a table are stored in relation to the available compute nodes. The rows are distributed to compute nodes according to the values in one column. If you distribute a pair of tables on the joining keys, Redshift collocates the rows on the compute nodes according to the values in the joining columns. This way, matching values from the common columns are physically stored together. There are four types of distribution styles: `auto`, `all`, `even`, and `key`. When you do not specify a distribution style, Amazon Redshift uses `auto,` which means Redshift will assign the style based on table size.

Automatic Table Optimization (ATO) is Redshift's self-tuning capability to optimize table design by applying sort and distribution styles without administrator intervention. ATO continuously observes how queries interact with tables, and uses advanced artificial intelligence methods to choose sort and distribution styles to optimize performance. To enable ATO, create a new table with Diststyle set to Auto and Sort key set to AUTO. You can also alter an existing table with an alter table command and change distribution style and sort key to `auto`.

We recommend keeping to an `auto` style for Diststyle and Sort key until you have a need for more manual distribution configuration. That time may come when you are starting to seek performance improvements and areas to finetune. Before starting, look at data around your model performance to develop a clear understanding of your [dbt project's DAG](#) (directed acyclic graph) and the transformations happening. Answer the following questions:

- What are your largest models?

- What kind of joins are you doing? And between which models? On what key?

dbt has several features to help you answer these questions in a scalable and automated way:

- In dbt Cloud, leverage the **Metadata API** and **Model Timing Tab** to collect data on your jobs and models

- Parse out **dbt artifacts** generated from model runs

- To get artifacts from model runs in Redshift, put the generated **[manifest.json](#)** into an S3 bucket, use the **[external tables package](#)** to stage it, and model the data using inspiration from the **[dbt artifacts package](#)**

The Redshift query optimizer is also a great tool to unpack model performance opportunities. Use the EXPLAIN plan to break down the CTEs of long running models and check if logic can be cleaned up or extracted out to another model. Ultimately, you want to reduce the joins that are most problematic and benchmark by adjusting the sort and dist key.

For the most part, we recommend using dist keys on columns with high cardinality in your largest tables, outside of the automated method. You will likely use them in large fact and dimension tables that are eventually exposed downstream in your BI platform. This can minimize duplicated storage, keep your data on the same node, and improve performance. If you're not able to distribute by one key, we recommend taking a look at our comprehensive blog post about this topic.

Storing Data Efficiently

We recommend the following table design practices, to store data more efficiently, minimize memory usage and disk spill, and improve query performance:

- **Use compression**, a column-level operation that reduces the size of data when it is stored, and improves query performance by reducing disk I/O. You can specify compression encodings when you create a table, but automatic compression usually produces the best results. ENCODE AUTO is the default for tables. Amazon Redshift automatically manages compression encoding for all columns in the table and balances overall performance when choosing compression encodings.

- **Minimize column width,** Consider the largest values you are likely to store in a VARCHAR column

- **Use the appropriate data type,** Store numeric and date/time values as corresponding data types.

Configuring Amazon Redshift Workload Management (WLM)

For the provisioned clusters, while setting up loader, transformer, and reporter groups, take advantage of Redshift's automatic workload management (Auto WLM) capability by creating separate WLM queues for each type of workload. Auto WLM optimizes resource use by adapting the concurrency level based on the runtime of each workload. This can help ensure that short, fast-running queries won't get stuck in queues behind long-running queries.

Having separate WLM queues also enables the use of WLM query monitoring rules (QMR) to control query priorities at a finer grain. Once configured, the rules will promote or demote queries at runtime based on metrics like queue wait, execution time, and CPU usage.

Choosing Appropriate Model Materializations

You may need to adapt your materialization strategy in dbt, if performance starts to lag with increasing project size and data volume.

dbt currently supports four materialization strategies out of the box: view, table, ephemeral, and incremental. By default, dbt will materialize models as views unless otherwise defined in `dbt_project.yml` or in a model `config` block.

We often recommend starting with views and tables to establish a strong foundation for a dbt project, and turning to more complex approaches (like incremental models) to improve efficiency as needed.

An [incremental model](#) is built like a normal table on its first run. On subsequent runs, it will process just the data matching the incremental filter criteria -- typically data that has been added or updated in the source table since the last run -- and insert the changes into the base table built in the first run. By limiting the amount of data transformed per run, incremental models can reduce the runtime and costs of those transformations. We often see incremental materializations applied to large-volume datasets, such as website event data or email CRM platform data.

Advanced users may also be interested in creating their own [custom materializations](#). Custom materializations can be referenced like any other materialization throughout your dbt project.

Materialized Views in Redshift

Materialized views contain precomputed result sets, and can be used to speed up complex queries such as those involving multiple joins and aggregations. To create materialized views in Redshift, use the necessary data definition language (DDL) in a materialization macro.

You can refresh materialized views on a schedule, or configure autorefresh to account for potential changes in the underlying base table components (row-level data). As Amazon Redshift [prioritizes user workload](#), it might stop autorefresh to preserve performance.

Amazon Redshift uses one of two approaches to refresh a materialized view:

- In an **incremental refresh** (not to be confused with [incremental materializations in dbt](#)), Amazon Redshift identifies new records and updates materialized views with only those records. This approach is supported if the underlying SQL for the view contains arithmetic SQL operations such as SUM and AVG.

- Amazon Redshift will perform a **full refresh**, which replaces all of the data in the materialized view, in [cases where an incremental refresh isn't supported](#).

To update a materialized view when its underlying SQL logic has changed, you will need to drop the existing view and rerun the model manually. For more information on Redshift's materialized views, check out the [documentation here](#).

Limiting data used in development

When developing models, you often need to finetune logic before it is ready to apply to an entire dataset. [Conditional logic can help you limit data usage](#) and reduce costs during this phase of work, without needing to manually write and remove LIMIT or WHERE clauses. To apply conditional logic in a DRY (don't repeat yourself) fashion, use a macro that can be called across models throughout your project. The sample code below adds the WHERE clause when the `target` is set to `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 %}
```

Using dbt packages to keep projects DRY

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.

`dbt_utils` supports an `insert_by_period` materialization type, in which dbt can insert records into a table one period (e.g. day, week) at a time. This materialization is most helpful for event data that can be processed in discrete periods or large data loads that fail at specific sections.

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.

Leveraging Redshift Spectrum and External Tables

Using Redshift Spectrum, dbt developers can directly query data stored in Amazon S3. If your team is on DC2 instances, storage and compute costs are coupled. When you need more storage but not more compute, you can reduce costs by offloading infrequently-queried data in Amazon Redshift into Amazon S3, and using Redshift Spectrum.

If your team uses a Serverless or RA3 instance, on the other hand, you can add storage without adding compute. Using Redshift Spectrum is still beneficial as you can directly query open data formats including Parquet, Avro, and more in Amazon S3.

Redshift Spectrum is a good fit for large datasets with billions of rows per day, such as web or email event data that need to be aggregated to produce more meaningful insights. Before using dbt with such data, follow these best practices:

1. [Partition the data in Amazon S3](#) by defining the S3 prefix based on the columns (ie event_date) frequently used in SQL predicate - Multilevel partitioning is encouraged if you frequently use more than one predicate.

2. [Set the table statistics (numRows)](#).

Reference to Amazon Redshift Spectrum performance tuning: [blog](#)

To create and manage external tables in Amazon Redshift, use [dbt's external tables package](#). This functionality allows you to create a YAML-defined source from S3-stored files or an external data source, then execute a [dbt run-operation command](#) to create, refresh, and/or drop the source.

To use dbt's external tables package with Redshift Spectrum, you must have:

- An existing S3 bucket

- Permissions set to create table

- An AWS Glue Data Catalog external database and Redshift external schema in which dbt will populate the external table in Redshift

This approach cuts complexity, and allows you to create and maintain the external table inside of dbt with downstream dependencies taken into account.

As an alternative, use [crawlers on AWS Glue](#) to create and manage external tables in Amazon Redshift. Crawlers will create external tables after extracting required information from your specified Amazon S3 path.

To use crawlers with Redshift Spectrum:

1. Create an external schema referencing an AWS Glue Data Catalog database

2. Create and run crawlers that creates external tables in the Data Catalog database

To make dbt aware of dependencies on an external table created by a crawler, define the external table as a source in your dbt project.
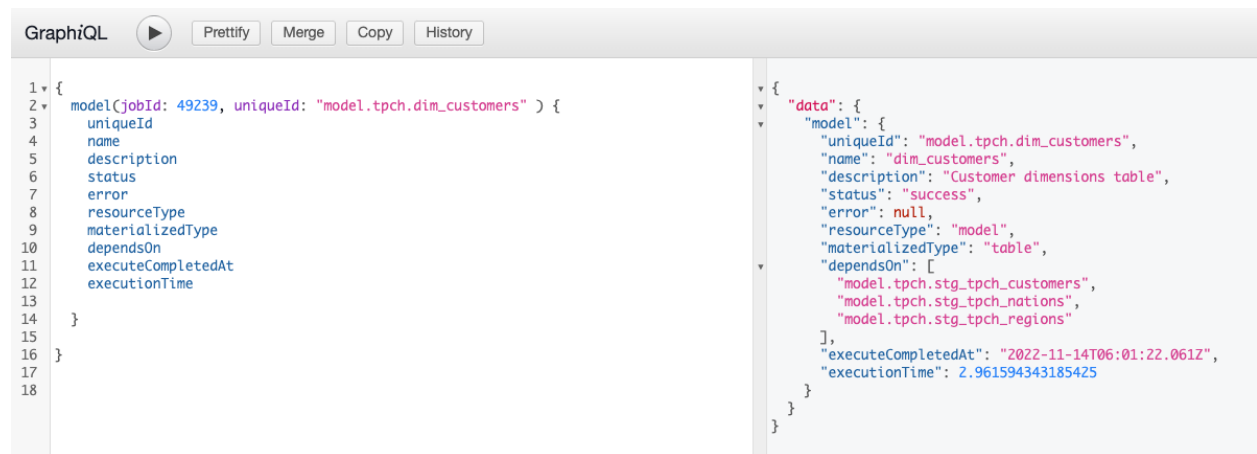
This approach may be preferred when you have little to no information about the external tables, and want to rely on crawlers to extract the information including columns, data types, and partitions. To update external tables with new partitions added over time, rerun the crawlers. You can rerun a crawler manually or on a schedule.

Data-Driven Code Optimization in dbt™ and Amazon Redshift

You can debug and optimize code and models systematically by leveraging dbt-generated metadata and Redshift's EXPLAIN command.

Analyzing dbt Metadata

With each job run, dbt Cloud 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 this 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. dbt Cloud users can also use the data via the Model Timing tab to visually identify models that take the most time and may require refactoring.

Querying the Metadata API: Using the dbt Cloud Metadata API, you access your dbt metadata including information about lineage, execution time, and documentation.
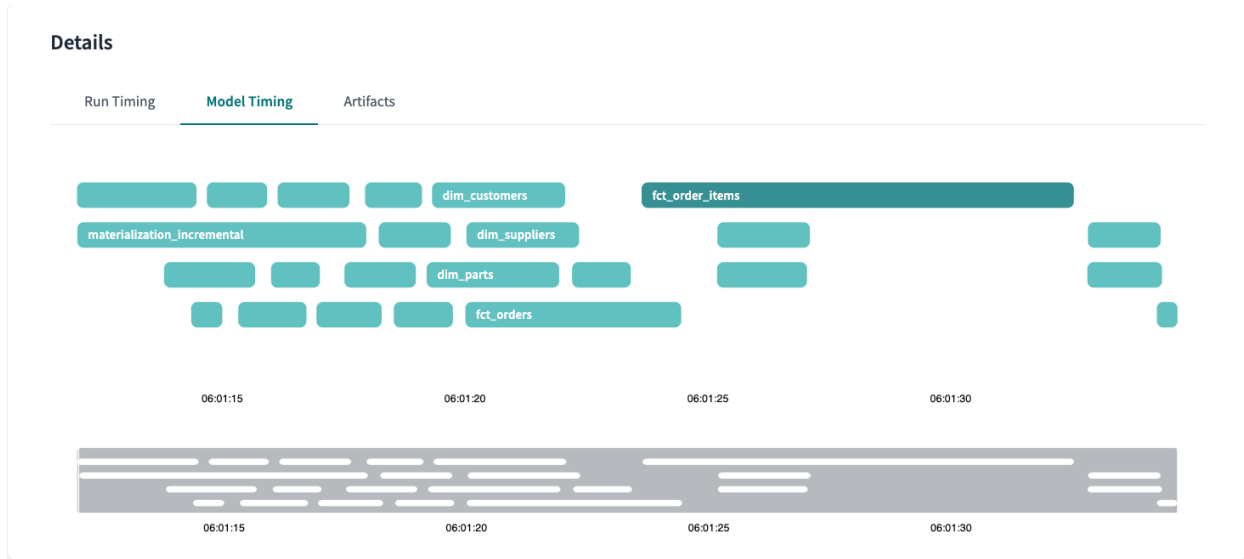
Using the Model Timing tab, this tab is a visual representation of your dbt job run, allowing you identity long running models.

Using Redshift's EXPLAIN and Query Summary

dbt users can also run Redshift's EXPLAIN command to see how a model will run in the query optimizer. By looking at the full query plan for a given model, you can get information on:

- Operations performed by the execution engine

- Tables and fields used per operation

- Volume of data processed in each step

- Operation cost

Note that the EXPLAIN command does not actually compute the query.

You can use this table to help connect the query plan and the query summary. You can execute thie explain plan in the dbt Cloud IDE or in the Redshift Query Explorer. Once you have identified a potential performance improvement, follow the usual dbt workflow by creating a new git branch and update any necessary changes (like materialization, sql command, etc), and test the improvement.

Conclusion

In summary, Amazon Redshift and dbt™ work together to power your data warehousing, and transformation operations. Amazon Redshift offers advanced features like Serverless, Data Sharing, Redshift Spectrum, and Federated query for optimized data ingestion and querying. Meanwhile, dbt™

enables efficient data transformation through Software engineering best practices such as modularity, CI/CD, and SQL-based logic. When combined, Amazon Redshift and dbt™ allows you to be seamless trusted data pipelines that facilitate you to derive quicks insights from your data.

Contributors

Contributors to this document include:

- Amy Chen, Engineering Manager-Partnerships, dbtlabs
- Kira Furuichi, Technical Writer, dbtlabs
- Maneesh Sharma, Sr. Database Engineer Redshift, Amazon Web Services
- Bhanu Pittampally, Sr Analytics Specialist SA, Amazon Web Services
- Jason Pedreza, Analytics Specialist SA, Amazon Web Services
- Harshida Patel, Principal Analytics Specialist SA, Amazon Web Services
- Randy Chng, Analytics Acceleration Lab SA, Amazon Web Services

Appendix: Recommendations for Setting Permissions and Access Controls in Redshift

Given the setup for Redshift we walked through earlier in this paper, we'll outline recommendations to set up permissions and groups for the long-term. The setup we suggest here will not necessarily work for every team, but will provide a starting point to adjust to your internal and regulatory requirements.

Setting Up Group Permissions

Set up Groups

First, set up groups. In Redshift, a group is a collection of users who share the same permissions. We recommend starting off with three primary groups: a `loader`, `transformer`, and `reporter`.

- The `loading` group needs to load data into your database. Extract and load (EL) tools, such as Fivetran or Airbyte, would fall into this group.

- The `transformer` group needs to transform the data. The members in this group may include dbt developers and applications like dbt Cloud.

- The `reporter` group needs to read data in the database. This group may include any ML tools that need to query models in your warehouse.

To establish each group, execute this command: `create group group_name`.

Create Users and Allocate to Groups

Next, create users and assign them into groups.

You will need to create a database user account for each human, as well as each application (such as an EL tool, BI platform, or dbt Cloud), that needs to load, transform, or read data in your warehouse.

ETL tools are generally classified as `loaders`, whereas dbt Cloud will need `transformer` permissions to run scheduled production jobs from your dbt project.

The code example below creates and assigns users to groups:

```
create user your_data_loader

    password '_generate_this_'

    in group loader;
create user your_user_1

    password '_generate_this_'

    in group transformer;
create user dbt_cloud

    password '_generate_this_'

    in group transformer;
create user your_bi_tool

    password '_generate_this_'

    in group reporter;
```

Next, we need to explicitly grant the appropriate permissions to each group.

Loader Group Permissions

The `loader` group will need the ability to create schemas and schema objects. With permission to create a schema, the group will also be able to create tables and views in that raw schema. If your `loader` group includes an ELT tool, make sure the group has some privileges on information schemas.

A note: If you are on a DC2 instance and only have one database, your `loader` group would load into your one database (the `analytics` database in the example below). If you are on a Serverless or RA3

instance and have multiple databases (e.g. `raw` and `analytics`), your `loader` group would need create access only on the `raw` database.

Run the following lines below to grant the `loader` group the correct permissions:

```
grant create on database analytics to group loader;

grant select on all tables in schema information_schema to group
loader;

grant select on all tables in schema pg_catalog to group loader;
```

Transformer Group Permissions

To enable `transformer` group users to create dbt models, generate [dbt docs](#), and conduct analysis, you'll need to grant them:

- access to schema usage

- select privileges on schema objects

- select privileges for future schema objects

- select privileges on information schemas

To grant access to schema usage, run the following commands for each schema:

```
grant usage on schema my_schema to group transformer;

grant select on all tables in schema my_schema to group transformer;

alter default privileges for user my_user in schema my_user

  grant select on tables to group transformer;
```

You will need to rerun these commands whenever you add a new schema to your database via an ELT tool like Fivetran or Airbyte.

To grant access to create objects in the database, run:

```
grant create on database analytics to group transformer;
```

To grant select privileges on information schemas, run:

```
grant select on all tables in schema information_schema to group
transformer;

grant select on all tables in schema pg_catalog to group transformer;
```

Reporter Group Permissions

After tables and views are created via the `transformer` roles, you can grant read access to the `reporter` group, which might include the tools that need to query those objects downstream, such as BI or ML tools.

The best way to manage access to tables and views is to use dbt's [grants configuration](#) to set those permissions directly in dbt. You can apply the configuration on the models config block in a YAML file, or on the config Jinja macro on the model's SQL file.

In `dbt_project.yml`, the configuration will look like this:

```
models:
  +grants:
    select: ['reporter']
```

On the model file, it will look like this:

```
{{ config(grants = {'select': ['reporter']}) }}
```

Note: If you have configured grants in multiple places (`dbt_project.yml`, `resources.yml`, and/or the model's SQL file), dbt will honor more-specific grantees over less-specific ones. For example: if you have different grants configured for a model in its YAML and model SQL files, dbt will only apply the grants applied on the SQL file. You can apply the + (addition) side to signal that you want to apply the new group in addition to the existing group.

This would look like:

```
{{ config(grants = {'+select': ['reporter']}) }}
```

[For more information, check out the documentation.](#)

Authentication Methods

dbt supports several authentication methods for Redshift users, including simple password-based authentication, IAM authentication, and connection via an SSH tunnel in dbt Cloud.

Username and Password Authentication

If you are developing locally, adjust your local `profiles.yml` to explicitly set the connection for a target using a username and password:

```
company-name:
 target: dev
 outputs:
```

```
dev:

  type: redshift

  host: hostname.region.redshift.amazonaws.com

  user: username

  password: password1

  port: 5439

  dbname: analytics

  schema: analytics

  threads: 4

  keepalives_idle: 240 # default 240 seconds

  connect_timeout: 10 # default 10 seconds

  sslmode: [optional, set the sslmode used to connect to the
database (in case this parameter is set, will look for ca in
~/.postgresql/root.crt)]
```

If you're using a secrets manager, rather than providing your password directly, use an environment variable, which will look like the below:

```
Password: "{{ env_var('DBT_USER_PASSWORD') }}"
```

Authentication with IAM

To set up a Redshift profile using IAM Authentication for local development, set the `method` parameter to `iam` in your local `profiles.yml` file. Note that a password is not required when using IAM Authentication.

Your local `profiles.yml` would look a something like this:

```
my-redshift-db:

  target: dev

  outputs:

    dev:

      type: redshift

      method: iam

      cluster_id: [cluster_id]

      host: hostname.region.redshift.amazonaws.com
```

```
user: username

iam_profile: data_engineer # optional

iam_duration_seconds: 900  # optional

autocreate: true           # optional

db_groups: ['analysts']    # optional


# Other Redshift configs:

port: 5439

dbname: analytics

schema: analytics

threads: 4

keepalives_idle: 240 # default 240 seconds

sslmode: [optional, set the sslmode used to connect to the
database (in case this parameter is set, will look for ca in
~/.postgresql/root.crt)]

ra3_node: true # enables cross-database sources
```

[For all potential profiles.yml configurations, check out this document](#)**.**

** IAM Authentication is [not supported](#) for Redshift Serverless as of the time of publication.

SSH Tunnel Connection

If you are using dbt Cloud, you can use an SSH tunnel to connect your dbt project to Redshift. Navigate to the "Projects" section in dbt Cloud, and follow the steps below:

1. Add your hostname, port (usually 5439 for Redshift), and username

2. Save the connection and a public key will be generated and displayed under the "Public Key" section

3. Copy this key to the bastion server which will authorize dbt Cloud to connect to Redshift

Protecting PII: Dynamic Data Masking with dbt

Oftentimes, you will want true column-values to be viewed by only a certain subset of users in our data warehouse. Columns that you may want to hide often contain sensitive PII that only admin users should

be able to view and access. To hide or contort this data for certain users, you can create a [dbt macro](#) to establish data masking logic that can be used across data models. At its core, this macro uses Redshift's `current_user()` value to determine whether the true column value or a hashed value of the column is shown. A deeper-dive into implementing this type of data masking in Redshift can be found in [this discourse](#) and in [this Github repository](#). Amazon Redshift now natively supports [dynamic data masking](#).

Document revisions

Date	Description
March 2023	First Publication