



Looker and BigQuery Important Considerations

Google Cloud Whitepaper
v1

Table of Contents

Introduction — Looker’s In-database Architecture	03
Leveraging BigQuery SQL	03
Nested data	
Partitioned data	
Clustered data	
User Defined Functions and Variable	07
Using BigQuery GIS for Geospatial Data	08
Improving Query Performance and Efficiency	09
Leveraging Looker caching with BigQuery	
Control per-query costs with connection settings	
Accelerate queries with Looker Aggregate Awareness	
Working with EAV data	
Managing and Monitoring Looker and BigQuery	14
BigQuery Information Schema block	
System Activity Analytics	
Managing users and permissions	
Leveraging OAuth with BigQuery and Looker	
Data Privacy and Security	18
Leveraging Google Cloud Data Loss Protection (Cloud DLP)	
Masking sensitive fields with Looker	
Using Looker with PCI data	
Machine Learning: Leveraging BigQuery ML	20
Using existing BigQuery ML models	
Training BigQuery ML models with data from Looker	
Advanced BigQuery Features	23
BI Engine	
BigQuery Omni	
Conclusion/Summary	24

Introduction — Looker's In-database Architecture

Looker is a modern data analytics platform that operates in-database, meaning Looker does not ingest source data. Instead, Looker leverages the power of your database's query engine to drive powerful analysis based on the freshest data. When a question is asked in Looker, it is converted into highly-performant SQL and delivered to your database. As a result, the query experience (speed, concurrency, etc.) is directly related to the power and sophistication of the database technology to which it is connected.

Where Looker is connected to Google BigQuery, data teams can draw on the performance, concurrency, elasticity, and advanced features of BigQuery to provide faster, deeper, more impactful, insights. In an environment where data is stored in, or accessible to, BigQuery's powerful query engine, Looker provides a superior user experience by communicating with the BigQuery cloud data warehouse using BigQuery-specific SQL commands and protocols (both BigQuery Standard SQL and BigQuery Legacy SQL are supported). This allows Looker to take advantage of the power and flexibility of BigQuery and to leverage some of the advanced features available in BigQuery such as machine learning and geospatial data types.

This document includes considerations, best practices, and features of interest for data teams using both Looker and BigQuery. It describes how to use the specialized features in BigQuery SQL, which Looker features to leverage, and how best to deploy Looker with Google BigQuery to get the most out of your data.

Leveraging BigQuery SQL

BigQuery supports two distinct SQL dialects: standard SQL and legacy SQL. Looker natively supports both of these SQL dialects, as well as more than [60 others](#). However, with Looker (and for BigQuery) the preferred query syntax for BigQuery is standard SQL. Looker experts suggest that data teams use standard SQL if possible, as it is more powerful and is actively maintained and enhanced by the BigQuery development team.

Learn more:

[BigQuery Standard SQL](#)

[Connecting Looker to BigQuery](#)

Nested Data

BigQuery is unusual among cloud data warehouses in its support for nesting. Nested records can provide considerable advantages when scanning over a distributed data set by physically co-locating related records. In some cases, properly nested data can help avoid unnecessary JOINS between tables and nested structures to avoid repeating data that would otherwise be repeated in a wide, denormalized table.

BigQuery standard SQL natively supports nested records within tables. With BigQuery, data teams can load and export nested and repeated data from source formats that support object-based schemas, such as JSON and Avro files.

When using BigQuery, queries that do not reference a nested column behave exactly as if the nested column was not present. When referencing a column of nested data, BigQuery can automatically flatten the data and the logic is identical to a co-located JOIN. So, for example, if you have a `web_sessions` table with events nested inside the sessions, when you query for a count of sessions, the presence of events imposes no performance penalty.

In Looker, users have the ability to easily look at a key from the JSON field without needing to constantly write complex queries where data is unnested (flattened). Looker is efficient with data storage because data is only unnested at query time, on an as-needed basis, with query logic defined once in LookML.

A JSON field is made to be its own dimension in Looker by parsing the field in the SQL definition.

```
view: encounter {
  sql_table_name: `looker-private-demo.healthcare_demo_live.encounter` ;;

  ### hospitalization fields ###

  dimension: hospitalization {
    hidden: yes
    sql: ${TABLE}.hospitalization ;;
  }

  dimension: hospitalization__discharge_disposition {
    type: string
    sql: ${hospitalization}.dischargeDisposition ;;
  }

  #nested array, included as its own view below
  dimension: hospitalization__discharge_disposition__coding {
    hidden: yes
    sql: ${hospitalization__discharge_disposition}.coding ;;
  }
}
```



A repeated JSON can be specified as its own view, and UNNEST is used in the join clause to join the flattened column back onto its parent.

```
explore: encounter {
  join: encounter__hospitalization__discharge_disposition__coding {
    sql: LEFT JOIN UNNEST(${encounter.hospitalization__discharge_disposition__coding})
    as encounter__hospitalization__discharge_disposition__coding ;;
    relationship: one_to_many
  }
}
```

Learn more:

[Specifying nested and repeated columns in BigQuery](#)

[Looker guide to nested data in BigQuery \(repeated records\)](#)

[Why nesting is so cool \(Looker blog\)](#)

Partitioned Data

A partitioned table is a special table that is divided into segments, called partitions, that make it easier to manage and query data. By dividing a large table into smaller partitions, data teams can improve query performance and control costs by reducing the number of bytes read by a query. BigQuery gives users the ability to partition their table by a date, by an integer, or by the time the data is ingested.

To specify a specific partition or set of partitions to use in a specific query, BigQuery users leverage a predicate filter (WHERE clause). For example, when the date column is used in the WHERE clause within a query, BigQuery will only scan the data in the partitions designated by that WHERE clause. This gives granular control over how much data is read in any given query, directly impacting query speed and cost.

Looker leverages partitions in BigQuery through filters in both the explore interface and through dashboards. In order to ensure these key fields are included when querying BigQuery data, Looker developers can require (or suggest) filters at both the dashboard and explore level. Leveraging BigQuery partition functionality in Looker does not impact end user analysis — end users do not need to have knowledge of partitions or the underlying data set to experience more efficient, faster queries.

It is strongly recommended that Looker developers leverage partition capabilities in BigQuery, particularly for very large tables. By using partitions, Looker developers can ensure end users query limited data sets where possible, helping eliminate inefficient, costly queries against the full dataset. Common use cases for partitioned data with Looker include querying event logs, machine data, and other easily partitioned large data sets.

Set conditional filters or always filters on explores with partitioned tables to create recommended or required filters for business users.

```
explore: event_logs {  
  label: "(5) Raw Event Logs (short term)"  
  view_label: "Events"  
  conditionally_filter: {  
    filters: [event_logs.timestamp_date: "90 days", event_sessions.session_start_date: "90 days"]  
  }  
}
```

Learn more:

[Introduction to partitioned tables in BigQuery](#)

[Using partition_keys in Looker](#)

Clustered Data

To improve query efficiency, BigQuery also allows users to specify cluster keys that designate the way data is sorted when it is being stored. Clustering provides storage optimization within columnar segments to improve filtering and record colocation. Clustering can improve the performance of certain types of query such as queries that use filter clauses and queries that aggregate data. Essentially, clustering will sort the values within the column so the BigQuery engine can easily find the records it needs to complete the query.

Looker developers can leverage clustered data in BigQuery in the same way they leverage partitions, e.g. through filters in both the explore and the dashboard. In order to ensure that these key fields are

being included in queries sent back to your database, Looker developers can require or suggest filters at both the dashboard and the explore level.

Using clustered data is recommended when you don't need to strictly limit the cost of a given query, you need greater granularity than is allowed with partitioning alone, and when queries commonly use filters or aggregation against multiple particular columns. Clustering is often used in conjunction with partitioning – e.g. partition by date, cluster by user_id.

When using LookML, cluster tables are used when the resulting table is persistent – either a Persistent Derived Table (PDT) or aggregate table – and is accomplished using the `cluster_keys` parameter. Google BigQuery tables can be partitioned on a date, timestamp, `ingest_time`, or integer.

You can use cluster or partition keys in the definition of the persistent derived tables.

```
view: daily_active_accounts {
  derived_table: {
    cluster_keys: ["id"]
    partition_keys: ["activity_date"]
    sql_trigger_value: select current_date();;
    explore_source: opportunity_line_item {
      column: active_arr {}
      column: activity_date { field: activity_calendar.activity_date }
      column: id { field: account.id }
      derived_column: primary_key {
        sql: CONCAT(activity_date,id) ;;
      }
    }
  }
}
```

Learn more:

[Introduction to clustered tables in BigQuery](#)

[Using cluster_keys in Looker](#)

User Defined Functions and Variables

A user defined function (UDF) in BigQuery is a reusable function, defined in SQL, and created by a Looker developer using another SQL expression or JavaScript. UDFs in BigQuery accept columns of input and perform actions, returning the result of those actions as a value. UDFs can be persistent or temporary. Persistent UDFs can be reused across multiple queries.

In Looker you can instantiate and call BigQuery UDFs via SQL expressions. To leverage existing persisted UDFs, the process is the same, simply calling the UDFs in the SQL definitions within your LookML model.

If you are creating a temporary UDF you can leverage the `sql_preamble` parameter in an Explore to define your function. Similarly, you can also define persisted BigQuery variables within the `sql_preamble` and leverage them throughout your model. You can also define persisted variables through Looker's SQL runner interface.

Defining a UDF in `sql_preamble`

```
explore: events {
  label: "(2) Web Event Data"
  sql_preamble:
    CREATE TEMP FUNCTION GET_URL_PARAM(query STRING, p STRING)
    RETURNS STRING
    LANGUAGE js AS """
    ret = null
    try{
    if(query) {
    params = query.split("&").forEach(function(part){
    item = part.split('=')
    if(item[0] == p ) {
    ret = decodeURIComponent(item[1])
    }
    });
    }
    }
    catch(err){}
    return ret
    """;;
```

Learn more:

[Understanding UDFs in BigQuery](#)

```
explore: sf_salary {
  sql_preamble:
    CREATE TEMP FUNCTION MEDIAN(a_num ARRAY<FLOAT64>)
    RETURNS FLOAT64 AS ((
    SELECT
      AVG(num)
    FROM (
    SELECT
      row_number() OVER (ORDER BY num) -1 as rn
      , num
    FROM UNNEST(a_num) num
    )
    WHERE
      rn = TRUNC(ARRAY_LENGTH(a_num)/2)
      OR (
      MOD(ARRAY_LENGTH(a_num), 2) = 0 AND
      rn = TRUNC(ARRAY_LENGTH(a_num)/2)-1 )
    ));
    ;;
}
```

Using BigQuery GIS for Geospatial Data

Location and geographic (geospatial) information can provide powerful insights to businesses, particularly when visualized using Looker. BigQuery GIS (Geographic Information Systems) lets you analyze and visualize geospatial data by using geography data types and standard SQL geography functions. With Looker, geographic information can play an important role in the data model and allows questions to be asked of the data such as “does this location fall within a certain region?”

BigQuery GIS is available only in BigQuery standard SQL. With BigQuery GIS, Looker developers and users can aggregate geopoints into more meaningful polygons like states or zip codes and then easily plot them for visual reference.

For polygons in BigQuery GIS, Looker developers can define their own geographic features using `ST_AsGeoJSON(geom)` to export the data in GeoJSON format, and convert that to a TopoJSON [using a custom script](#). Free tools such as [mapshaper.org](#) and others can provide simple ways to convert GIS data.

You can use GIS functions, coupled with BQ public datasets, to find the zipcode for each lat/lon point.

```
view: card_transactions_with_zip {
  derived_table: {
    sql_trigger_value: select max(trans_date) from `looker-private-demo.retail_banking.card_transactions`;
    sql: SELECT
      card_transactions.trans_id,
      zip_code AS merchant_zip,
      city as merchant_city,
      county as merchant_county,
      state_name as merchant_state
    FROM
      `looker-private-demo.retail_banking.card_transactions` AS card_transactions,
      `bigquery-public-data.geo_us_boundaries.zip_codes` AS zip_codes
    WHERE ST_Within(ST_GEOGPOINT(card_transactions.merchant_lon,card_transactions.merchant_lat)),
      zip_codes.zip_code_geom);
}
```

Learn more:

[Introduction to BigQuery GIS](#)

[Example script for building custom geographic ranges \(GitHub\)](#)

[Mapshaper.org](#) – a free tool for use in GIS conversions

Improving Query Performance and Efficiency

Leveraging Looker Caching with BigQuery

Looker can reduce database load and improve query response time by using the cached results of prior queries. Database concurrency is improved, too, due to reduced need to access source data. This also reduces query costs by reducing the amount of data queried in BigQuery. Caching policy can be defined in several ways in Looker: at the explore level, at the model level, or by using the datagroups. It is often easiest to set caching policy by applying datagroups within explores, using the `persist_with` parameter.

When Looker performs a query, it looks first in the cache for existing, valid results of that same query. If no valid results are found in the cache, a normal query is executed. Looker experts recommend optimizing your caching policies to sync with ETL policies wherever possible to reduce database query traffic. (i.e. If your ETL only updates data every hour and the exact same query was run 10 minutes ago, there's no sense rerunning it.) You can control caching policy and sync Looker data refreshes with your ETL process to integrate more closely with the backend data pipeline, so cache usage can be maximized without the risk of analyzing stale data. Named caching policies can be applied to an entire model or to individual explores and persistent derived tables (PDTs). Refer to the aggregate awareness section (below) for more information on PDTs.

You can customize the caching logic for models or explores using datagroups and the `persist_with` parameter.

```
datagroup: ecommerce_etl {  
  sql_trigger: SELECT max(created_at) FROM ecomm.events ;;  
  max_cache_age: "24 hours"  
}  
  
persist_with: ecommerce_etl
```

Learn more:

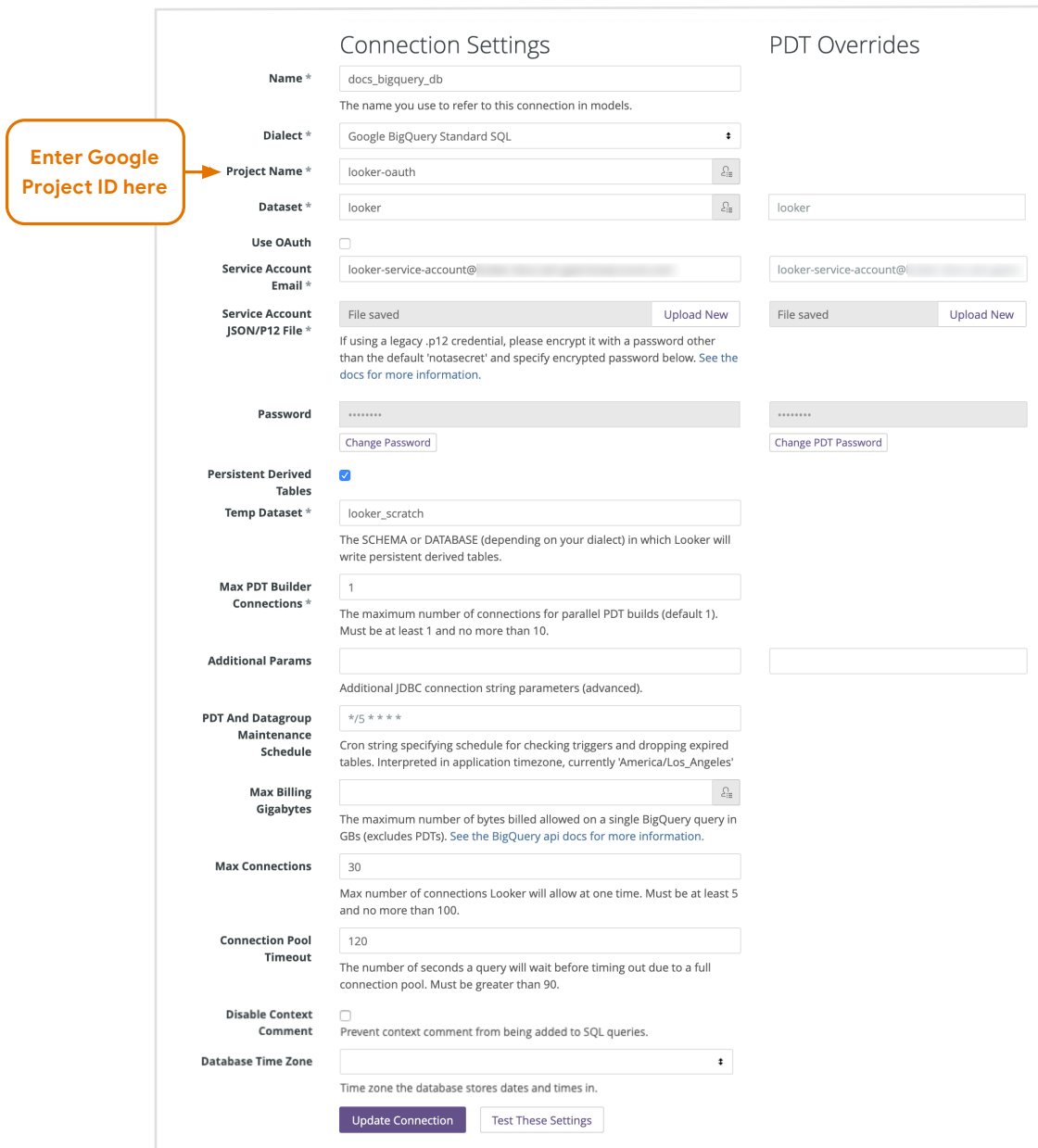
[Caching queries with datagroups](#)

[Best practices: Optimizing Looker performance](#)

Control Per-Query Costs with Connection Settings

BigQuery natively supports a number of convenient ways to control query costs, including a database connection setting called maximum bytes billed. With this setting any query that exceeds the maximum bytes billed value will be prevented from running, without incurring any usage costs. This provides a quick way to eliminate very large, expensive queries.

This setting is defined when Looker is connected to BigQuery through the database connections dialog in Looker. Because the maximum bytes billed setting is a parameter for the JDBC connection between Looker and BigQuery (defined in GB), simply using the setting will control costs. Leaving this setting blank will eliminate limits on bytes billed.

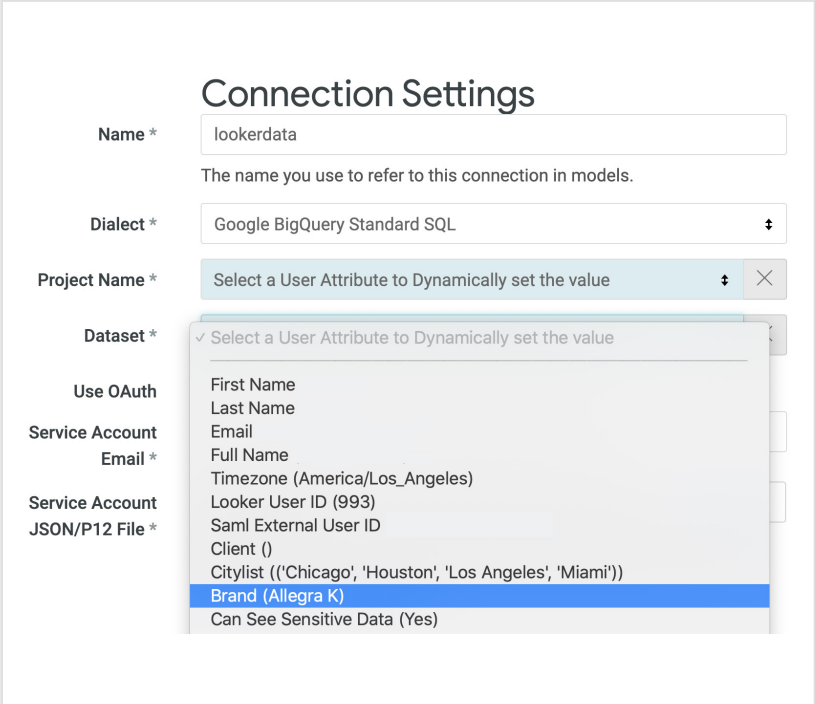


The screenshot shows the 'Connection Settings' dialog in Looker. A callout box on the left contains the text 'Enter Google Project ID here' with an arrow pointing to the 'Project Name' field. The 'Project Name' field contains the value 'looker-oauth'. Other fields include 'Name' (docs_bigquery_db), 'Dialect' (Google BigQuery Standard SQL), 'Dataset' (looker), 'Service Account Email' (looker-service-account@...), 'Service Account JSON/P12 File', 'Password', 'Persistent Derived Tables' (checked), 'Temp Dataset' (looker_scratch), 'Max PDT Builder Connections' (1), 'Additional Params', 'PDT And Datagroup Maintenance Schedule' (*/*S * * *), 'Max Billing Gigabytes', 'Max Connections' (30), 'Connection Pool Timeout' (120), 'Disable Context Comment' (unchecked), and 'Database Time Zone'. The 'Max Billing Gigabytes' field is currently empty, which is the setting being discussed in the text. Buttons for 'Update Connection' and 'Test These Settings' are at the bottom.

By leveraging the user attributes capabilities of Looker, you can customize the Looker experience, including connection parameters, for an individual user or group of users. This is particularly useful when you want to provide certain users with unlimited query access and limit access to others. For example, permitting analysts to run queries of any size but limiting the ability of business users to perform massive, expensive queries. This feature can also be useful when enforcing data governance or security policies (see also Managing Users and Permissions later in this document).

From the user management dialog in the admin menu in Looker, a Looker administrator can quickly add constraints to user queries by defining the BigQuery maximum bytes billed setting for that user (or group). This gives user-level granular control over query costs.

You can leverage user attributes in the connection settings.



Learn more:
[BigQuery cost control best practices](#)

[Setting up BigQuery connection parameters in Looker](#)

[Looker user attributes](#)

[BigQuery pricing overview](#)

Accelerate Queries with Looker Aggregate Awareness

When Looker communicates SQL to BigQuery it automatically optimizes the query using [relational algebra](#). This makes queries faster, helps reduce the load on the BigQuery query engine, and lowers cost-per-query. For queries that run frequently, large queries, or expensive queries, Looker developers can create smaller persistent aggregate tables of data, grouped by various attributes. These derived tables are defined in LookML and provide a way for analysts to create new tables without the need to engage data engineering resources and without altering the source data in any way. Persistent derived tables (PDTs) are written into a scratch schema in BigQuery and regenerated at a chosen frequency.

Aggregate tables, defined in the LookML model using the `aggregate_table` parameter, are derived tables that act as summary tables or roll-ups. Looker preferentially uses these aggregates instead of the larger table when performing queries, a behavior we call “aggregate awareness.” Using aggregate awareness logic, Looker always queries the smallest aggregate table possible to answer your users’ questions. If a query span exceeds that of the available aggregate, Looker will automatically `union` fresh data to an aggregate. The original table would be used only for queries requiring finer granularity than the aggregate tables can provide.

LookML example for aggregate awareness

```
explore: event_logs {
  label: "(5) Raw Event Logs (short term)"
  view_label: "Events"
  conditionally_filter: {}
  join: event_sessions {}
  join: event_session_facts {}
  join: account {}
  aggregate_table: event_hourly {
    query: {
      dimensions: [timestamp_hour]
      measures: [count, user_count, event_session_facts.average_session_duration, event_sessions.count]
      timezone: America/Los_Angeles
    }
    materialization: {
      datagroup_trigger: event_trigger
    }
  }
  aggregate_table: event_daily {
    query: {
      dimensions: [timestamp_date]
      measures: [count, user_count, event_session_facts.average_session_duration, event_sessions.count]
      timezone: America/Los_Angeles
    }
    materialization: {
      datagroup_trigger: event_trigger
    }
  }
}
```

Looker aggregate awareness works seamlessly with BigQuery. Once an aggregate is defined, no additional configuration is required and Looker will create and maintain the aggregate in scratch space on BigQuery. As with all materialization in Looker, this is a new table and the original data table is not modified. Aggregates are instantiated as persistent derived tables and have a consistent and persistently addressable name. Using Looker's API, third party tools, such as data science tools, can directly call the data in the aggregate. Accessing a PDT aggregate with third party tools is, at the time of this writing, only supported on BigQuery Standard SQL and not in BigQuery Legacy SQL.

Working with EAV Data

Entity-Attribute-Value (EAV) is a common mechanism to encode key/value pairs in a table and is particularly useful when storing data before it's possible to fully define all necessary attributes (columns). While EAV is simple and flexible, it can result in very long and "skinny" tables and can result in a data schema that's cumbersome to analyze. To perform analysis of EAV data, it is common to define Common Table Expressions (CTEs) to allow EAV data to be addressed using more conventional SQL.

For EAV data, Looker can be used to transform and materialize a more conventional data schema in BigQuery. The EAV table can be parsed once using a LookML query and materialized as a Persistent Derived Table (PDT) in BigQuery. The PDT can be updated incrementally and automatically on a defined schedule to improve query efficiency. Subsequent queries leverage the PDT and are simple and efficient. By building a PDT, the need to write and rewrite cumbersome CTEs can be virtually eliminated.

Open source tools such as PyLookML can be used to autogenerate the LookML to build the resulting PDT.

Learn more:

[Pylookml](#) — an open source tool to autogenerate LookML for EAV data

[Derived Tables](#) — Looker documentation

Managing and Monitoring Looker and BigQuery

BigQuery Information Schema Block

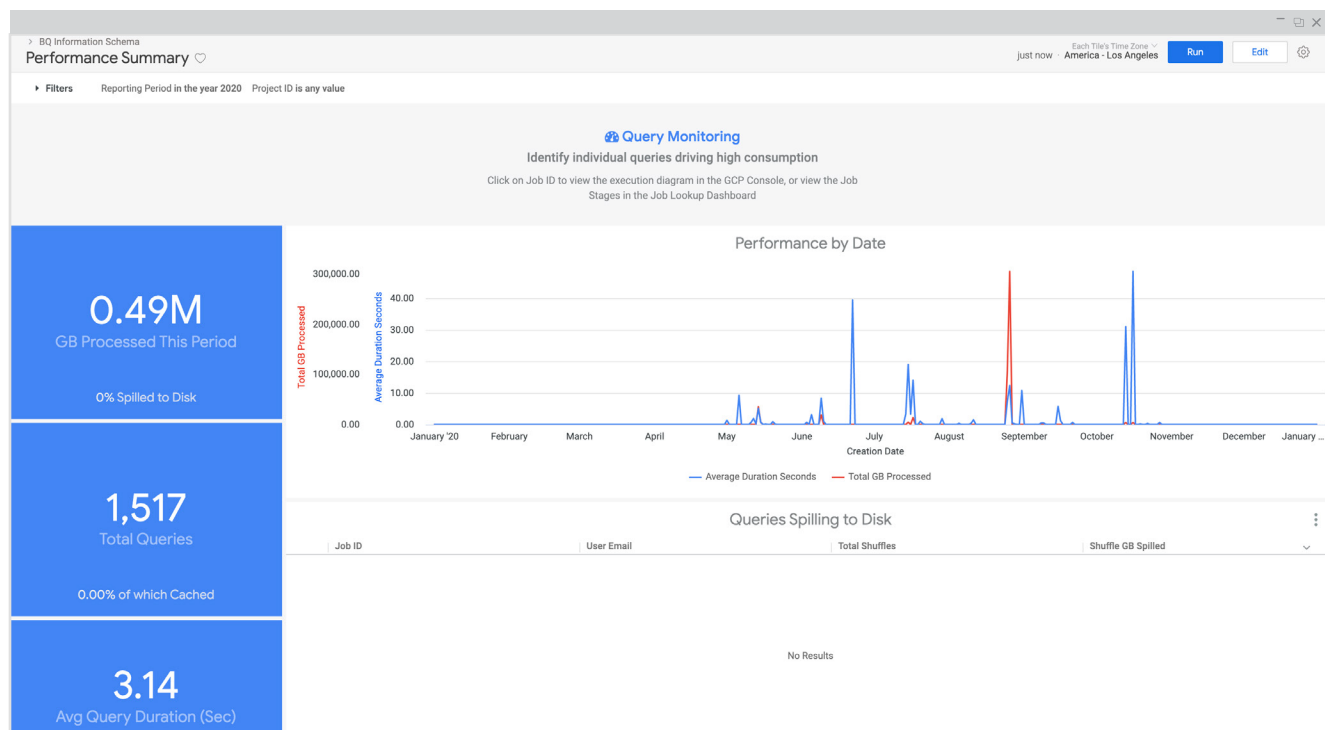
Looker blocks are pre-built data models for common analytical patterns and data sources. BigQuery provides comprehensive data on resource consumption and performance via a native set of metadata tables called the [BigQuery INFORMATION_SCHEMA](#).

Google has developed a [Looker block](#) for the BigQuery Information Schema to help Looker teams operate as efficiently as possible and to provide the highest level of performance while containing costs for [flat-rate](#) BigQuery users.

With the Looker block for BigQuery Information Schema, Looker administrators can:

- Optimize slot capacity distribution by investigating windows and patterns of high and low consumption against capacity.
- Identify troublesome queries and users driving concurrency and average query time across all projects in the organization.
- Monitor overall health and performance of the BigQuery environment, providing a central hub for database administration and auditing.
- Optimize investment in BigQuery by breaking down queries by their individual execution patterns and job stages and reconfiguring query patterns and table partitions or clusters.

Monitoring Queries with the Looker Block for BigQuery Information Schema



The BigQuery Information Schema block is designed to analyze data at the organization level (the By_Organization level of granularity in the Information Schema) to provide the most complete picture of resource consumption and efficiency. Therefore, to use this block requires a [high level of permissions](#) from the Looker Service Account to BigQuery.

The tables from the Information Schema that are used for the Block in its current iteration are:

[Jobs_By_Organization](#)

- This table contains one row for every query that was executed across an entire organization

[Jobs_Timeline_By_Organization](#)

- This table contains one row for every second of execution of every BigQuery job. Each period starts on a whole-second interval and lasts exactly one second, allowing users to view changes in slot usage during the course of a query's [execution timeline](#).

The BigQuery INFORMATION_SCHEMA can be analyzed outside of the Looker block, just as with other data available through BigQuery. Building on the existing Looker block, teams can access views that provide access to:

- dataset metadata
- job metadata
- job timeline metadata
- reservation metadata
- streaming metadata
- routine metadata
- table metadata
- view metadata

Additionally, third party organizations also offer Looker blocks that can help optimize cost and performance in BigQuery, such as the [Google BigQuery Performance block by Datatonic](#).

Learn More

Looker block for [BigQuery Information Schema Performance Monitoring](#)

System Activity Analytics

Looker [system activity](#) is an internal model designed for the analysis of Looker's underlying application database. Explores and dashboards built on the model show information about a Looker instance, including all dashboards, Looks, users, and queries – and can be very helpful for monitoring and auditing. Pre-built system activity dashboards and explores are available under the Admin menu and include User Activity, Content Activity, Database Performance, Instance Performance, and Errors and Broken Content.

Teams optimizing BigQuery performance can use the existing dashboard to identify users who drive the most queries and are able to track historical performance trends. It's also easy to identify periods of high scheduled activity or high concurrency using the Concurrency Timelines and Hourly Schedules Looks.

System activity database performance dashboard



Learn more:

- [Looker system activity pages documentation](#)
- [Looker system activity database performance dashboard](#)
- [Managing Looker at Scale with System Activity Analytics \(video\)](#)

Managing Users and Permissions

If BigQuery has different accounts with various access restrictions, you can leverage those existing database permissions in Looker. To leverage those restrictions, Looker administrators can set parameters around the user name and password of a connection so each user connects with the appropriate credentials for their database access level. While this will ensure that users do not see data to which they shouldn't have access, this will not affect which Explores, dimensions, and measures are shown to them in Looker.

For example, if a user is configured to connect to the database with an account that prevents them from seeing a credit_card_number column in the user table, any dimension using that database column will still be shown to them in Looker. They will simply receive an error from the database if they attempt to run a query that includes that dimension. Tracking user activity and permissions in Looker is relatively straightforward using the Looker system activity [user activity dashboard](#). With this dashboard you can quickly identify users with administrative or developer permissions, track user activity and engagement, and embedded user statistics.

Learn more:

[Setting up user attributes in Looker](#)

Leveraging OAuth with BigQuery and Looker

Leveraging OAuth: leverage existing permissions within Google Cloud to make sure end users can only see the data that they're supposed to [How to setup OAuth in BQ connection](#).

Example of BQ connection with OAuth

The screenshot shows the 'Connection Settings' form in Looker. The fields are as follows:

- Name ***: lookerdata (The name you use to refer to this connection in models.)
- Dialect ***: Google BigQuery Standard SQL
- Project Name ***: lookerdata
- Dataset ***: thelook
- Use OAuth**:
- OAuth Client ID ***: looker_demo
- OAuth Client Secret ***: [Redacted]
- Additional Params**: [Empty field] (Additional JDBC connection string parameters (advanced).)
- PDT And Datagroup intenance Schedule**: */5 * * * * (Cron string specifying schedule for checking triggers and dropping expired tables. Interpreted in application timezone, currently 'America/Los_Angeles')

Data Privacy and Security

Data teams are often tasked with defining privacy, security, and compliance postures for the data under their care. Although this data is common, in general it is recommended Looker administrators reduce or eliminate access to sensitive data, including PII, PHI, PCI, and other sensitive data types. Sensitive data should only be made available for analysis where it is absolutely necessary and only to authorized users, defined by user and group permissions in Looker and in BigQuery. Google Cloud provides several tools that work in conjunction with Looker to keep sensitive data private and in compliance with relevant regulations.

Leveraging Google Cloud Data Loss Protection (Cloud DLP)

The first step in protecting sensitive data is limiting its presence in BigQuery. Google Cloud Data Loss Protection (Cloud DLP) is a fully-managed service that helps you protect sensitive data on or off Google Cloud. With Cloud DLP you can inspect data to identify and track sensitive data types, tokenize or mask data where necessary, and transform both unstructured and structured data. Cloud DLP is best used as data is moved into or while data is stored in BigQuery, before it is connected to Looker. Once private data is remediated in BigQuery, Looker administrators can share data with confidence.

Learn more:

[Google Cloud Data Loss Protection \(Cloud DLP\)](#)

Masking Sensitive Fields with Looker

While there are several ways to mask sensitive data within Looker, one flexible way is to reference user attributes in SQL using liquid syntax. User attributes are defined by the Looker administrator and can be applied to individual users or to groups. Looker automatically includes some user attributes, but administrators can also define attributes for which users supply values (such as passwords or contact information). By connecting user access to view, then creating a dimension that references the attribute but only returns value when the attribute is properly set, administrators can quickly define, on a very granular level, data visibility.

Access grants are another way to leverage user attribute values to control which end users are able to view specific explores, joins, views or fields. Access grants are defined at the model level with the `access_grant` parameter. As part of the definition, you associate the access grant with a user attribute. You also specify which user attribute values provide access to the access grant.

Learn more:

[Masking sensitive fields for certain user](#)

User Attributes

General

Settings

Labs

Legacy Features

Whitelabel

Users

Users

Groups

Roles

Content Access

User Attributes

Database

Connections

Queries

Persistent Derived Tables

Datagroups

Definition

Name

This is how you reference this attribute in Looker expressions and LookML. Attribute Names can only contain lower case letters, underscores, and numbers. They cannot start with a number.

Label

This is the user-friendly name displayed in the app for lists and filters.

Data Type

User Access

None

View

Edit

If "None", non-admins will not be able to see the value of this attribute for themselves. "View" is required to use this attribute in query filters. If "Edit", the user will be able to set their own value of this attribute, so the user attribute will not be able to be used as an access filter.

Hide Values

Yes

No

If "Yes", the value will be treated like a password, and once set, no one will be able to decrypt and view it. It will only be able to be used in passwords and secure webhook authentication. Once set to "Yes" for an attribute, cannot be unset.

Default Value

Value when no other value is set for the user or for one of the user's groups.

Using Looker with PCI Data

Where possible, it is suggested that data teams eliminate access to PCI data in Looker. This can be accomplished by using tools such as Cloud DLP, transforming data at time of loading into the database, or removing sensitive data once it's loaded in BigQuery. In general, PCI data such as

card numbers holds little analytical data in itself – and using customer IDs or other mechanisms can reduce the need to include that data in analysis. The Looker customer success team is an excellent resource if you are considering making PCI data available in Looker, as they can provide advice and alternatives. Some details on PCI and securing data in Looker are available on looker.com.

Machine Learning: Leveraging BigQuery ML

BigQuery ML allows SQL developers to create and execute machine learning models in BigQuery using standard SQL queries. With LookML & BigQuery ML, developers can build models using existing tools and skills, speeding up development and eliminating the need to move data out of BigQuery or spin up new infrastructure for predictive model development.



Using Existing BigQuery ML Models

Because BigQuery ML natively supports models for regression, clustering, factorization, and more, if you need these types of models you can simply take advantage of them by calling them via SQL commands right in your LookML model, using Looker Derived Tables. When a query is executed, the SQL commands, including calling the BigQuery ML model, are sent to BigQuery for execution and results are returned as part of the query results. Because these results are the same as any other query results, you can then use them in reports or dashboards, build a Looker PDT, and send them via data actions just as with any query results. Because a LookML query with BigQuery ML operates just as any other query you can pull results from BigQuery ML via Looker APIs, seamlessly.

Here is an example of using a simple Derived table to build a full ML model within BigQuery:

```
view: future_purchase_model {
  derived_table: {
    datagroup_trigger: bqml_datagroup
    sql_create:
      CREATE OR REPLACE MODEL ${SQL_TABLE_NAME}
      OPTIONS(model_type='logistic_reg'
        , labels=['will_purchase_in_future']
        , min_rel_progress = 0.005
        , max_iterations = 40
        ) AS
      SELECT
        * EXCEPT(fullVisitorId, visitId)
      FROM ${training_input.SQL_TABLE_NAME};;
  }
}
```

And that's it. Notice that this simply references other LookML objects, so we don't need massive SQL queries to manage these models. We can take each feature and manage it as it's own LookML field, then just select the ones we want to put into our `training_input` table.

Learn more:

[Supported models in BigQuery ML](#)

[BigQuery ML and Looker \(blog\)](#)

Training BigQuery ML Models with Data from Looker

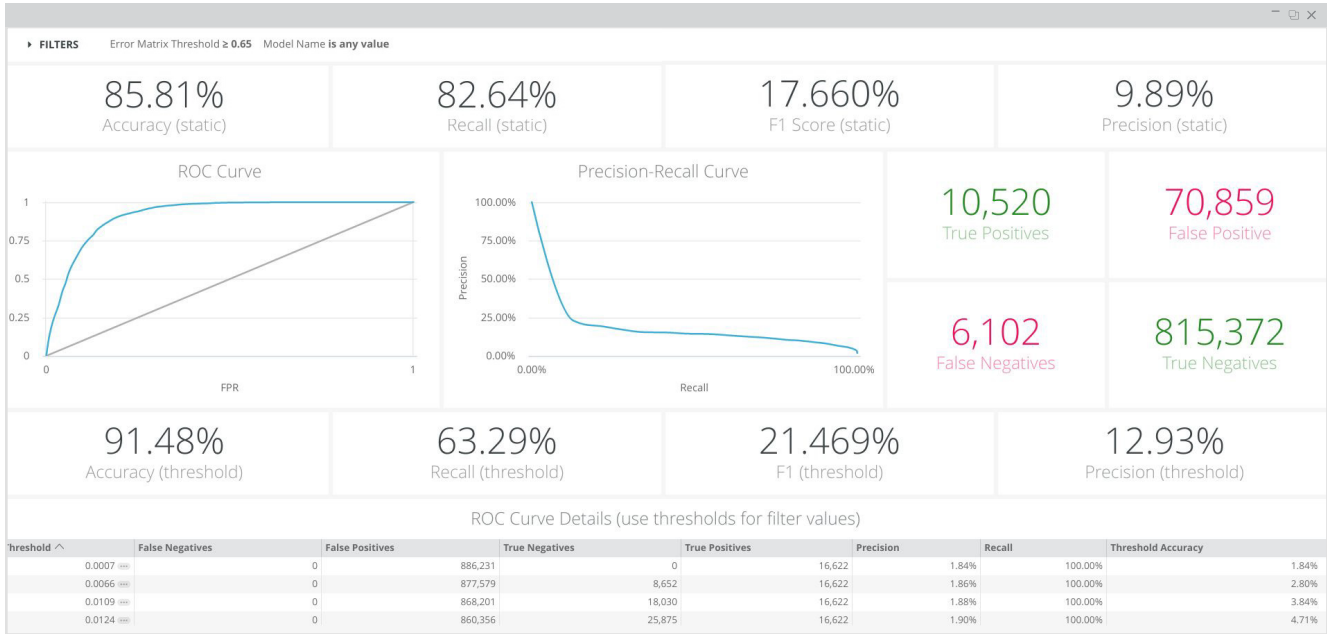
Looker makes it easy to compile training data using NDTs that make use of the logic you've already put into LookML (DRY), then you can use datagroups to trigger how often you want to retrain your model. End users can see the results joined back on with other business-critical data so you can use ML outputs to make meaningful business decisions without worrying about annoying ETL processes.

Grabbing the results of a BigQuery ML model is as simple as another derived table in Looker:

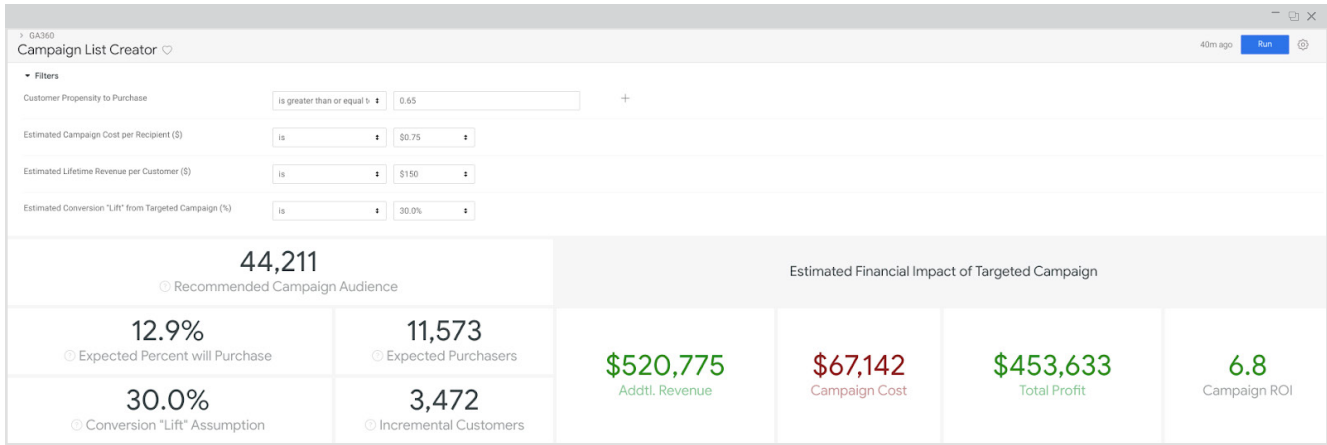
```
view: future_purchase_prediction {
  derived_table: {
    sql: SELECT * FROM ml.PREDICT(
      MODEL ${future_purchase_model.SQL_TABLE_NAME},
      (SELECT * FROM ${future_input.SQL_TABLE_NAME}));;
  }
  dimension: predicted_will_purchase_in_future {
    type: number
    description: "Binary classification based on max predicted value"
  }
  dimension: predicted_will_purchase_in_future_probability {
    value_format_name: percent_2
    type: number
    sql: ${TABLE}.predicted_will_purchase_in_future_probs[ORDINAL(1)].prob;
  }
}
```

Once this is in place, we can set it and forget it. Regardless of how our model changes, this is just a set of pointers to that model, and the unlabeled data we'll use to make predictions, `future_input.` We also have reference to the prediction and the probability of that prediction, so we don't need to worry about writing that bit of SQL every time we want to look at it.

Combining the above, here's an example of how we could actually check on the performance of our model (using the various BigQuery ML performance metrics provided in the [ML.CONFUSION_MATRIX](#), [ML.EVALUATE](#), and [ML.ROC_CURVE](#) functions) in one clean Looker Dashboard. Again, set it and forget it. Make the dashboard once, retrain the model by altering the input, and simply hitting "run" on your dashboard.



Once you have a model that you like, it can be incorporated into a variety of live use cases. Since the flow of data is already automated, using Looker dashboard, schedules, alerts, and a [wide array of data delivery options](#), we can operationalize this information very easily. Here is a simple example of a dashboard using the outputs of our model to create an ROI calculator for potential marketing campaigns. This could be used to actually create lists for campaigns.



Common use cases may include:

- Use clustering algorithms to create organic categories for your customers to better understand your user base. Provide this information to your marketing team for better informed decision making.
- Find anomalies in your transactions using regression analysis, to alert you of potentially fraudulent activity.
- Create a recommendation engine to show customers likely products that they'll want based on historical data.

Learn more:

[BigQueryML Looker block](#)

Advanced BigQuery Features

BI Engine

BI Engine is BigQuery's fast, in-memory analysis service. With BI Engine, data is automatically moved between in-memory storage, the BigQuery cache, and BigQuery storage to optimize query performance. BI Engine is enabled via the Cloud Console where an administrator can add and remove BI Engine memory capacity using the BigQuery UI.

BI Engine is most useful for Looker administrators who require extremely fast query response time and improved concurrency. Common use cases in Looker include the analysis of large data sets or streaming data where speed and freshness is important. BI Engine can be used to help Looker dashboards load faster, particularly where they rely on larger queries.

When BI Engine is unable to accelerate a query, performance will revert to that of a typical BigQuery query – automatically. This means there's no need for a Looker administrator to tune or modify their Looker queries to accommodate BI Engine. Query

types that typically cannot take advantage of the high-performance capabilities of BI Engine include pivots, multi-level JOINS, and distinct aggregates.

For the fastest queries and rapid dashboard loading, using BI Engine in conjunction with Looker aggregate awareness is recommended. Looker's aggregate awareness capabilities complement BI Engine's in-memory approach and materialized views (PDTs) located in-memory provide exceptionally fast query response times.

Learn more:

[BI Engine documentation](#)

BigQuery Omni

BigQuery Omni is a multicloud analytics solution that lets you access and analyze data across clouds, including Google Cloud, AWS, and Azure (coming soon). Because moving data between clouds can be onerous, BigQuery Omni provides the ability to query data directly from Amazon S3 buckets, without moving data between clouds.

There is no specific configuration required to connect Looker to BigQuery Omni. Looker connects to BigQuery Omni through the standard BigQuery JDBC connection, and BigQuery-specific SQL commands are transmitted by the BigQuery system to the query engine located in AWS. Queries are executed in AWS by BigQuery, and only query results are communicated to Looker. Because only query results are returned, costs associated with data movement between clouds are minimized.

At the time of this writing, BigQuery Omni access to S3 data is read-only. This precludes the use of PDTs and aggregate awareness functionality in Looker, as it is not possible to materialize query results in S3 using BigQuery Omni. Best practices for improving performance and efficiency of queries include leveraging the Looker cache to provide faster access to common queries.

Conclusion/Summary

The integrations and best practices for Google BigQuery and Looker are constantly evolving and improving. While this document is intended to be a useful reference, Looker documentation is the most accurate and frequently updated source for this information and it is suggested that readers consult it regularly to learn the latest. Other resources of interest include Looker's help documents and the Looker community.

[Looker documentation](#)

[Looker help center](#)

[The Looker community](#)

[Looker blogs](#)

If you need help or information beyond that available here or online, please contact your Looker representative for assistance.