BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse

Justin Levandoski
Garrett Casto
Mingge Deng*
Rushabh Desai
Pavan Edara
Google
Seattle, WA, USA

Thibaud Hottelier
Amir Hormati*
Anoop Johnson
Jeff Johnson
Dawid Kurzyniec
Google
Seattle, WA, USA
bigquery-sigmod2024@google.com

Sam McVeety
Prem Ramanathan
Gaurav Saxena
Vidya Shanmugam
Yuri Volobuev
Google
Seattle, WA, USA

ABSTRACT

BigQuery's cloud-native disaggregated architecture has allowed Google Cloud to evolve the system to meet several customer needs across the analytics and AI/ML workload spectrum. A key customer requirement for BigQuery centers around the unification of data lake and enterprise data warehousing workloads. This approach combines: (1) the need for core data management primitives, e.g., security, governance, common runtime metadata, performance acceleration, ACID transactions, provided by an enterprise data warehouses coupled with (2) harnessing the flexibility of the open source format and analytics ecosystem along with new workload types such as AI/ML over unstructured data on object storage. In addition, there is a strong requirement to support BigQuery as a multi-cloud offering given cloud customers are opting for a multi-cloud footprint by default.

This paper describes BigLake, an evolution of BigQuery toward a multi-cloud lakehouse to address these customer requirements in novel ways. We describe three main innovations in this space. We first present *BigLake tables*, making open-source table formats (e.g., Apache Parquet, Iceberg) first class citizens, providing fine-grained governance enforcement and performance acceleration over these formats to BigQuery and other open-source analytics engines. Next, we cover the design and implementation of *BigLake Object tables* that allow BigQuery to integrate AI/ML for inferencing and processing over unstructured data. Finally, we present *Omni*, a platform for deploying BigQuery on non-GCP clouds, focusing on the infrastructure and operational innovations we made to provide an enterprise lakehouse product regardless of the cloud provider hosting the data.

CCS CONCEPTS

• **Information systems → Database management system engines**.

*Work done while at Google.

KEYWORDS

Data Warehousing, Data Lake, Unstructured Data, Multi-Cloud Analytics

ACM Reference Format:
Justin Levandoski, Garrett Casto, Mingge Deng*, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati*, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, and Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugam, Yuri Volobuev. 2024. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3626246.3653388

1 INTRODUCTION

Google Cloud released BigQuery in 2010 as the first fully-managed serverless multi-tenant cloud data warehouse on the market. BigQuery's architecture separates compute, storage and shuffle, allowing each system component to evolve relatively independently [31]. Since its inception, this architecture has allowed BigQuery to evolve in multiple dimensions to address core customer problems at scale, including industry-first features like embedded ML (BQML) and high-throughput stream ingestion [21]. This has allowed BigQuery to continue to be one of the fastest growing services on the Google Cloud Platform (GCP).

The large-scale analytics ecosystem has converged recently toward a "lakehouse" architectural paradigm that merges traditional data-warehousing – typically OLAP/BI dashboarding and business reporting over structured relational data – with data lakes, bringing about new workload types such as the combination of AI/ML and large-scale analytics over unstructured data along with enterprise-ready workloads over open-format table formats such as Apache Iceberg [2]. In general, a modern cloud analytics architecture separates compute from storage [11, 20], with key capabilities pushed to the durable storage component to address enterprise customer requirements, such as governance, security, metadata management, and performance. BigQuery's disaggregated architecture is well-suited to provide these capabilities. BigQuery's query processing engine, Dremel, provides the ability to process data in-situ over various storage substrates [30, 31]. BigQuery's managed storage tier maintains a clean separation of compute and storage. This storage tier provides Read and Write APIs to allow third-party analytics engines (e.g., Spark [39], Presto/Trino [34]) to directly write to and consume its managed data [21, 22].

This paper provides an overview of how we evolved BigQuery to solve key customer problems in the modern lakehouse era. We describe how we extended the BigQuery architecture to support a general purpose lakehouse platform across both Google Cloud and non-GCP clouds through BigQuery Omni. We first describe *BigLake tables*, an evolution of BigQuery that makes tables in open-source storage formats (Parquet [3], Apache Iceberg) first class citizens within BigQuery to provide several common data management primitives that bridge the gap between data warehouses and open data lakes and analytics engines. We then describe the functionality behind BigQuery's Read and Write APIs, and how we extended this infrastructure to support open source data lakes on object storage to provide key features like uniform governance enforcement like column and row-level security across BigQuery and arbitrary open-source data lake analytics engines like Spark and Presto/Trino. We also describe how we achieved order-of-magnitude query processing performance improvements over open source storage formats by extending BigQuery's scalable physical metadata management [22] to BigLake tables. Last, we provide an overview of *BigLake managed tables* that provide fully managed features like ACID transactions and physical layout optimizations in Apache Iceberg format on customer-owned object storage.

Next, we describe *BigLake Object tables* that extend BigQuery to support unstructured data (e.g., documents, images, video, audio) as a first class citizen. Coupled with modern AI/ML inference techniques that BigQuery supports through our BQML inference engine, Object tables provide a powerful primitive to work with multi-modal data — spanning structured to unstructured data types — within a cloud data warehouse, including the ability to extract meaningful structure from objects and perform analysis alongside structured tables. In addition to Object tables, we describe how we extended Dremel to perform image inference completely within a relational query engine, which to our knowledge is an industry first. We also describe how we extend inference to remote models hosted in GCP VertexAI[1] through the BQML inference engine.

Last, we describe how we extended our lakehouse primitives in BigQuery to other clouds through *Omni*. BigQuery Omni is a pioneering Google Cloud technology that allows us to ship core architectural components of BigQuery on non-GCP clouds (currently AWS and Azure) as if they were running on Google core infrastructure such as Borg [37]. Specifically, we describe the core technology behind Omni and how we ship and run Dremel, BigQuery's query processing engine, on foreign clouds to query customer data lakes on Amazon S3 and Azure Blob Storage (or ADLS) in-situ. We also describe how BigLake primitives including fine-grained governance and performance acceleration work uniformly work on Omni as they do on GCP.

The rest of this paper is organized as follows. Section 2 provides an overview of BigQuery and the architecture and the core architectural components we evolved to build a multi-cloud lakehouse. We then provide an overview of BigLake tables in Section 3, focusing on the delegated access model, the BigLake Storage Read/Write APIs and common governance model, performance acceleration, and BigLake managed tables that provide fully managed BigQuery tables in Apache Iceberg format on customer object storage. Section 4 covers BigLake Object tables and unstructured data management in BigQuery, along with integration with BQML inference engine. The architecture and implementation of Omni is covered in Section 5. Section 6 provides a sample of interesting customer use case patterns that we observe in production. Finally, Section 7 concludes the paper.

[1]GCP VertexAI Platform: https://cloud.google.com/vertex-ai

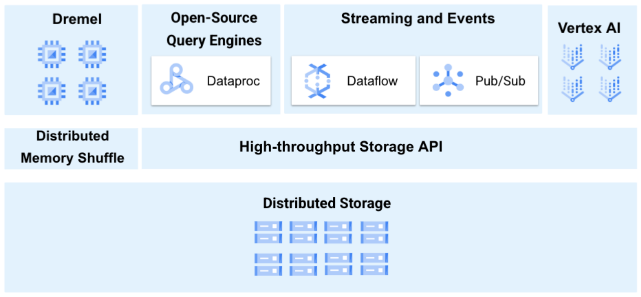

Dremel
Open-Source Query Engines
Dataproc
Streaming and Events
Dataflow
Pub/Sub
Vertex AI
Distributed Memory Shuffle
High-throughput Storage API
Distributed Storage

Figure 1: High-Level BigQuery Architecture

2 BIGQUERY ARCHITECTURE

BigQuery is a fully-managed, serverless data warehouse that enables scalable analytics over petabytes of data. The BigQuery architecture depicted in Figure 1 is based on the principle of separation of storage and compute. A replicated, reliable and distributed storage system holds the data, and elastic distributed compute nodes are responsible for data ingestion and processing. In addition, BigQuery also features a separate shuffle service built on top of disaggregated distributed memory that facilitates communication between compute nodes and provides query checkpointing for dynamic re-optimization. BigQuery also employs a set of "horizontal" services such as a control plane and job management, metadata/catalog, parser/frontend, and security/governance to name a few. These services tie the system together to provide an enterprise data warehouse offering.

A disaggregated architecture such as this provides flexibility to evolve components relatively independently and is key to several of the features we describe in this paper. As depicted in Figure 1, this flexibility in architectural evolution allows for closer integration with several other key Google Cloud infrastructure services in storage, analytics, and AI/Ml to form a modern data and AI platform. For instance, on the analytics side, the BigQuery lakehouse allows close integration with open-source engines like Spark and Presto hosted in Dataproc [5] that can write or read directly to/from BigQuery storage. Similarly, close integration with Vertex AI — GCP's AI/ML platform — allows us to harness powerful primitives for AI tasks like inference over multi-modal data managed by BigQuery, whether it be structured tables or unstructured data such as images, documents, or video. Such an architecture also allowed us to ship core pieces of BigQuery infrastructure such as Dremel on non-GCP clouds. In the rest of this section, we provide an overview of the main BigQuery components that we evolved to support the customer need for a multi-cloud lakehouse.

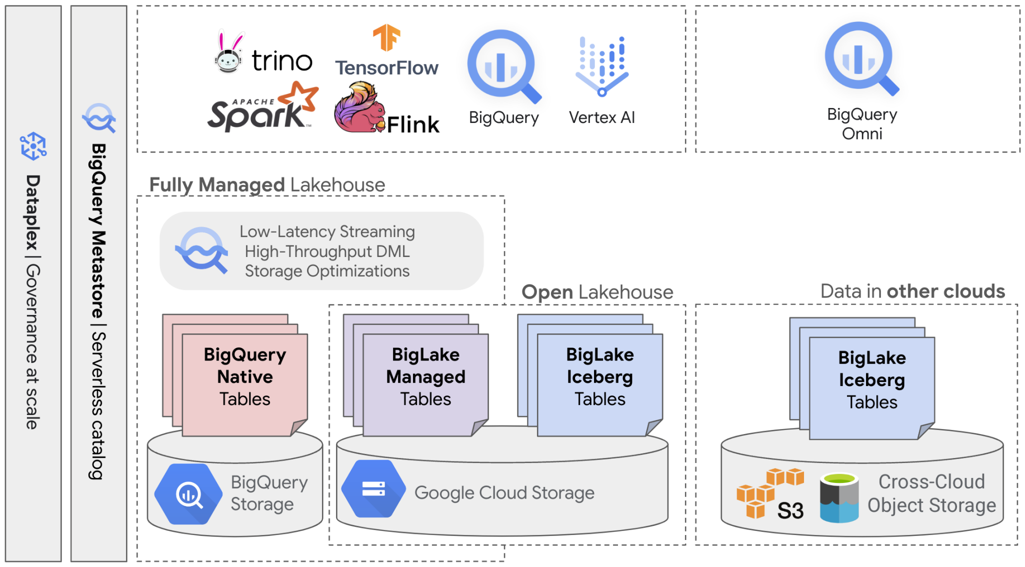


Figure 2: BigLake Architecture

2.1 Dremel and In-Situ Analytics

Dremel is BigQuery's massively scalable parallel query processing engine. The original release of Dremel in 2006 tightly coupled storage with compute [30], but eventually evolved to separate compute and storage to support in-situ data analysis. As described in previous work [31], this led to the proliferation of a "data lake" ecosystem within Google, with a self-describing columnar shared storage format on common GFS [24] and later Colossus [27] storage that interoperated with a number of workflow tools and analytics engines [17, 18, 25, 33]. To support a full suite of important enterprise data warehousing features, BigQuery built support for Managed Storage[2]

With the release of BigQuery, Dremel's in-situ analytics capabilities were extended to query open-source storage formats such as Parquet, Avro, and ORC on Google Cloud Storage (GCS) through *external tables*. In this model, BigQuery provided basic read-only access to self-describing files on cloud object storage, without support for basic query optimization, data modifications, nor core enterprise features like security and governance. In Section 3, we describe how we evolved BigQuery to eliminate this enterprise feature gap with BigLake tables, and also allowed customers to use this functionality through our Storage APIs to provide these important capabilities to managed data lakes running third-party engines such as Apache Spark.

2.2 Storage APIs

On release, the storage and compute components of BigQuery were hermetic from a user perspective, even though they were technologically distinct and architecturally separate. Data had to be loaded into BigQuery managed storage before it could be queried and data had to be exported to GCS in order to access it via other analytical engines. In order to meet the needs of evolving data analytics pipelines, we added the Storage API to BigQuery to remove these constraints and allow external engines to access the BigQuery managed storage layer in a manner similar to Dremel. This API contains two services for ingress and egress: the Read API and Write API.

[2]https://cloud.google.com/bigquery/docs/reference/storage

2.2.1 Read API. The BigQuery Read API offers a high-performance, scalable way of accessing BigQuery managed storage and BigLake tables. The Read API is implemented as a gRPC-based protocol that uses an efficient binary serialization format, with scalability features such as support for multiple streams for reading disjoint sets of rows in parallel (suitable for parallel clients such as external analytics engines like Apache Spark or Presto/Trino). The Read API also provides a feature-rich governance layer that enforces the same coarse- and fine-grained access control and data visibility mechanisms as core BigQuery (column- and row-level access control, data masking), and allows for efficient reads of a subset of data through filter pushdown and column projection. We expand on the details of how we leverage governance and filter pushdown in Section 3. The Read API has two main methods:

- `CreateReadSession`: this allows the user to specify the parameters of the table read that they will be performing. A given read session provides consistent point-in-time reads. It requires specifying the table that will be read from, and optionally allows for a variety of modifications to what data is required and how the user will consume it. For instance, what timestamp is used to read the data, what columns are read, and query predicates. As output, this method returns a list of stream objects.
- `ReadRows`: that is called using the stream objects provided by *CreateReadSession* where each stream contains some subset of the data. Multiple clients can be used to read data from individual streams, and while the initial number of streams is selected to provide enough parallelism for the amount of data to be read, dynamic work rebalancing is possible through further splitting individual streams.

The Read API embeds Superluminal [17], a C++ library that does high-performance vectorized execution of GoogleSQL expressions and operators. Superluminal enables efficient columnar scans of the data and enforcement of user predicates and security filters. Superluminal applies the projections, user/security filters, data masking and transcodes the result into Apache Arrow, allowing for high-performance query execution from a variety of engines.

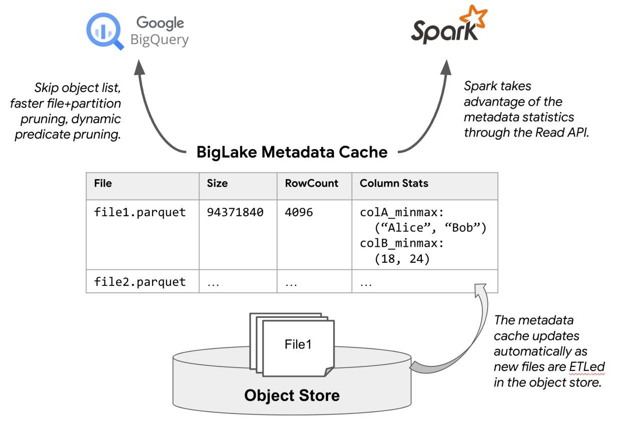
2.2.2 Write API. The Write API provides a mechanism for scalable, high-speed and high-volume streaming data ingestion into BigQuery with support for multiple streams, "exactly once" delivery semantics, stream-level and cross-stream transactions, and an efficient gRPC-based wire protocol. Similar to the Read API, a user creates a session, and then uses a single or multiple write streams to append rows to the destination table. Different writing modes are available to provide the desired processing semantics (real-time streaming or batch commit) and commit guarantees.

3 BIGLAKE TABLES

For many years, BigQuery supported enterprise-level data management capabilities (e.g., fine-grained security, ACID transactions, physical metadata management) through its native storage layer. Support for querying open-source data lakes on object storage was implemented via basic read-only external tables. However, with customers deploying more and more open-source data lakes, we started hearing several key customer requirements that these data lake deployments should have (1) the same level of enterprise data management capabilities as BigQuery, no matter where the data is stored, and (2) have many of these advanced features available to other data lake analytics engines, such as Spark and Presto/Trino. In other words, customers wanted a single core platform that solved the difficult data management problems once, but have it work across storage substrates (e.g., BigQuery storage or data lakes on object storage) and analytics stacks.

Our solution to these customer requirements is *BigLake tables* that evolve and extend pieces of BigQuery components to provide a managed lakehouse. BigLake tables provide uniform enterprise data management features across various analytics engines and storage platforms. Figure 2 provides a high-level overview of BigLake tables and how they interact with the extended analytics and storage ecosystem. The key ideas behind BigLake tables are two-fold. First, they extends external open-source data lake tables to be first-class citizens in BigQuery, making table definitions in the BigQuery catalog the source of truth (instead of through self-describing files) to allow features like fine-grained security. Second, these tables provide enterprise functionality to the broader analytics engine ecosystem through the Read/Write APIs.

In this section, we provide a technical overview of BigLake tables. First, we discuss the initial release that uses a delegated access model (Section 3.1) to data lakes on object store and with the ability to provide fine-grained governance and security such as column/row-level security and data masking through the Read API that works uniformly across external analytics engines like Spark or Trino (Section 3.2). We then discuss subsequent releases that provide performance acceleration through physical metadata caching that has allowed us to improve performance for certain data lakes workloads by up to an order of magnitude through both Dremel and Spark


Google BigQuery
Spark
Skip object list, faster file+partition pruning, dynamic predicate pruning.
Spark takes advantage of the metadata statistics through the Read API.
BigLake Metadata Cache
File1
Object Store
The metadata cache updates automatically as new files are ETLed in the object store.

File	Size	RowCount	Column Stats
file1.parquet	94371840	4096	colA_minmax: ("Alice", "Bob") colB_minmax: (18, 24)
file2.parquet	...	...	...

Figure 3: Performance acceleration for open-source data lake tables

accessing data through the BigLake Read API (Section 3.3 and 3.4). We then provide an overview of recent work on BigLake managed tables (Section 3.5), that provide a fully managed read/write table format in the Apache Iceberg format on cloud object storage.

3.1 Delegated Access Model

Typically, query engines accessing external storage forward the querying user credentials to the object store, which in turn performs its own data access authorization checks. This model does not work for BigLake tables for two reasons: (1) credential forwarding implies that the user has direct access to raw data files, which would allow users to bypass fine-grained access controls such as data masking or row-level security; and (2) BigLake tables need to access storage outside of the context of a query to perform maintenance operations, for example refreshing the metadata cache (Section 3.3), or background data reclustering (Section 3.5).

BigLake tables rely on a delegated access model where users associate a connection object with each table. The connection object contains service account credentials that are granted read-only access to the object store. The table uses the connection credentials to process queries and perform background maintenance operations. Users can reuse the same connection object for multiple tables: typically, our customers use one connection per data lake.

3.2 Fine-Grained Security

BigLake tables provide consistent and unified fine-grained (row and column-level) access controls independent of storage (data lake or data warehouse) or analytics engine (BigQuery or open-source query engines like Spark).

- For BigQuery users, the delegated access model enables BigQuery to enforce column-security, data masking, and row-level filtering using the same implementation for data in object stores or in its native storage.
- The current status quo in open-source analytics places the responsibility of enforcing the fine-grained access controls with the query engines. This leads to two downsides: (1) security policies such as data masking and row-level filtering

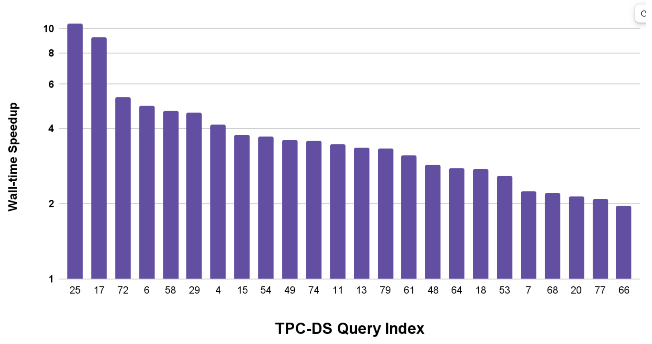


Figure 4: TPC-DS Speedup with Performance Acceleration

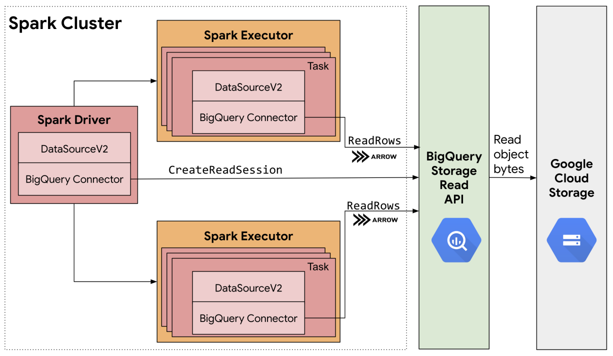


Figure 5: External analytics engine interaction with the Read API

are tied to a SQL dialect, requiring duplication of governance policies across multiple query engines (2) the model of entrusting the query engine to enforce filtering do not work well on engines like Apache Spark that are designed to run arbitrary procedural code directly within the query engine worker processes.

BigLake tables offer a stronger security model where the Read API establishes a security trust boundary and applies the same set of fine-grained access controls before data is returned to the query engine, with zero trust granted to the query engine itself. As a result, BigLake tables are able to provide a uniform security model that extends to external query engines with no expectation of trust from the external query engines.

3.3 Performance Acceleration

Open source tables that are not backed by modern managed table formats (e.g. Apache Iceberg [2], Apache Hudi [1], Delta Lake [10]) employ limited physical metadata: typically, only the file system prefix of a table or a partition is stored in the metadata. As a result, query engines need to perform listing operations on object storage buckets to obtain the list of data files to operate on. Listing of large cloud object store buckets with millions of files is inherently slow. On partitions that query engines cannot prune, the engine needs to peek at data file-level metadata, such as headers or footers, to determine if it can skip data blocks, requiring several additional object reads. These aspects can impose a significant overhead in query planning and execution.

To accelerate query performance, BigLake tables support a feature called *metadata caching*. Figure 3 shows how BigLake automatically collects and maintains physical metadata about files in object storage. BigLake tables use the same scalable physical metadata management system employed for BigQuery native tables, known as Big Metadata [22]. The use of Big Metadata allows using the same distributed query processing and data management techniques that we employ for managing data to handle metadata.

Using Big Metadata, BigLake tables cache file names, partitioning information, and physical metadata from data files, such as physical size, row counts and per-file column-level statistics in a columnar cache. The cache tracks metadata at a finer granularity than systems like the Hive Metastore [35, 36], allowing BigQuery and storage APIs to avoid listing files from object stores and achieve high-performance partition and file pruning.

The statistics collected in the physical metadata management layer enable both BigQuery and Apache Spark query engines to build optimized high-performance query plans. To measure the performance gains, we performed a power run of the TPC-DS 10T benchmark where each query is executed sequentially, on a BigQuery reservation with 2000 slots. Figure 4 shows the TPC-DS query speedup for a subset of the queries and how the BigQuery query execution time improved for queries through the statistics collected by the BigLake metadata layer. Overall, the wall clock execution time decreased by a factor of four with metadata caching.

3.4 Accelerating Spark Performance over Storage APIs

A large subset of BigQuery customers use Apache Spark in addition to BigQuery SQL. The BigQuery storage APIs provide Spark high throughput read/write access to the data in BigQuery native and BigLake tables. The open-source Spark BigQuery Connector [7] provides an out-of-the-box integration of the storage APIs with Spark DataFrames using Spark's *DataSourceV2* interface. During query planning, the Spark driver creates the read API session, which returns a list of read streams. During the execution, Spark executors perform a parallel read of the streams. The read API returns the rows in the Apache Arrow columnar data and Spark's native support for Apache Arrow minimizes the memory copies.

One of our goals was that customers using Spark against BigLake tables should get a similar price-performance compared to the baseline of Spark directly reading the Parquet data from GCS. This would enable uniform data governance spanning Spark and BigQuery, across data lake and warehouse data assets. Customers would not need to choose between price-performance and security. Accomplishing this goal required performance improvements that spanned the stack.

Our initial prototype of Parquet scans in the read API reused Dremel's row-oriented Parquet reader. The rows are then translated into the Superluminal columnar in-memory format. While this was relatively simple to implement, it is inefficient due to the translation from Parquet columns into rows and back into Arrow columnar batches.

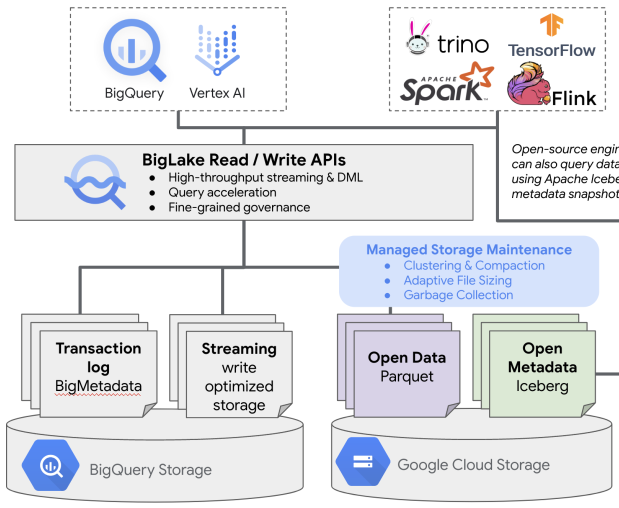

BigQuery
Vertex AI
trino
TensorFlow
Spark
Flink
BigLake Read / Write APIs
High-throughput streaming & DML
Query acceleration
Fine-grained governance
Open-source engines can also query data using Apache Iceberg metadata snapshots
Managed Storage Maintenance
Clustering & Compaction
Adaptive File Sizing
Garbage Collection
Transaction log BigMetadata
Streaming write optimized storage
Open Data Parquet
Open Metadata Iceberg
BigQuery Storage
Google Cloud Storage

Figure 6: Overview of BigLake Managed Tables

We implemented a vectorized Parquet reader that can directly emit Superluminal columnar batches from Parquet. Superluminal can operate directly on dictionary and run-length encoded data. This allowed vectorized execution of the entire ReadRows pipeline, doubling the read throughput and improved the server-side CPU efficiency by an order of magnitude.

We extended the *CreateReadSession* API to return data statistics collected in Big Metadata. The Spark connector uses these statistics to improve query plans. This unlocked optimizations such as dynamic partition pruning on queries with snowflake joins, improved join reordering and exchange operator reuse. The combined effect of these optimizations led to a 5x improvement in the Spark query performance observed on the TPC-DS benchmark. On the TPC-H benchmark, Spark performance against BigLake tables now match or exceed the baseline of Spark's direct GCS reads. Future work in this area will include:

- *Efficiency of the ReadRows payload.* Clients typically spend a non-trivial amount of CPU cycles on the TLS decryption of *ReadRows* payload. Dictionary and run-length encodings on the Arrow columnar batches can significantly reduce the amount of bytes that need to be sent over the wire.
- *Reuse of read sessions.* Dynamic partition pruning can generate new predicates during runtime and this results in the recreation of Read API sessions. Creating a Read API session is expensive on the server side since it requires enumerating/pruning files and persisting stream metadata to Spanner. Dynamic partition pruning can be redesigned to reuse read API sessions.
- *Aggregate pushdown.* DataSourceV2 supports pushing down partial aggregates such as *MIN/MAX/SUM*. The Read API can be extended to compute the partial aggregates using Superluminal, returning a much smaller payload to Spark.

```
SELECT uri, predictions FROM
  ML.PREDICT(
    MODEL dataset1.resnet50,
    (
      SELECT ML.DECODE_IMAGE(data) AS image
      FROM dataset1.files
      WHERE content_type = 'image/jpeg'
      AND create_time > TIMESTAMP('23-11-1')
    )
  );
```

Listing 1: In-engine inference on images

3.5 BigLake Managed Tables

BigLake managed tables (BLMTs) offer the fully managed experience of BigQuery managed tables while storing data in customer-owned cloud storage buckets using open file formats. BLMTs support DML, high-throughput streaming through the Write API, and background storage optimizations (adaptive file sizing, file reclustering and coalescing, and garbage collection). Data is stored in Parquet, while metadata is stored and managed using Big Metadata. Users can export an Iceberg snapshot of the metadata into cloud storage, allowing any engine capable of understanding Iceberg to query the data directly. Iceberg snapshots are currently triggered using a SQL statement. In the future, the Iceberg snapshots will be automatically generated asynchronously as part of table commits.

BLMT is different from open table formats such as Iceberg and Delta Lake in a couple of aspects:

- BLMTs are not constrained by the need to atomically commit metadata to an object store. Object stores can update/replace an object only a handful of times per second, thus placing a limit on the number of mutations per second that can be performed with pure object store tables.
- Open table formats store the transaction log along with the data. A malicious writer can potentially tamper with the transaction log and rewrite table history.

Using Big Metadata as the metadata source-of-truth provide the following benefits:

- *Write throughput.* Big Metadata is backed by a stateful service that caches the tail of the transaction log in memory. Big Metadata periodically converts the transaction log to columnar baselines for read efficiency. During queries, Dremel reads the columnar baselines and reconciles it with the tail. The combination of in-memory state and columnar baselines allows table mutations at a rate much higher than what is possible with open table formats without sacrificing read performance.
- *Multi-table transactions.* Reusing Big Metadata enables BLMT to support features such as multi-table transactions that are currently unsupported in open table formats.
- *Strong security model.* Since writers cannot directly mutate the transaction log, the table metadata is tamper-proof with reliable audit history. Writers do not need to be trusted for security nor for correctness and integrity.

```
CREATE OR REPLACE MODEL
  mydataset.invoice_parser
REMOTE WITH CONNECTION
  us.myconnection
OPTIONS (
  remote_service_type = 'cloud_ai_document',
  document_processor = 'proj/my_processor');

SELECT *
FROM ML.PROCESS_DOCUMENT(
  MODEL mydataset.invoice_parser,
  TABLE mydataset.documents
);
```

Listing 2: External inference using Document AI

4 SUPPORTING UNSTRUCTURED DATA

The vast majority of the world's data is unstructured such as documents, audio, and images. Unstructured data is harder to analyze than structured or semi-structured data: it is siloed from traditional data warehousing, requiring engineers to build custom data pipelines to integrate structured and unstructured data insights. BigQuery Object tables provide a SQL interface to object store metadata. With Object tables, BigLake enables users to analyze unstructured data using local and remote AI services using familiar SQL commands.

4.1 Object Tables

Object tables are system-maintained tables where each row represents an object, and columns contain object attributes such as URI, object size, MIME type, creation time. The output of `SELECT *` on an object table is equivalent to `ls` or `dir` on a filesystem. BigLake features extend naturally to unstructured data:

- *Fine-grained Security.* Object tables take advantage of the delegate access model (Section 3.1) to maintain the following invariant: access to a row implies access to the content of the corresponding object. For example, a row-access policy can limit access to objects uploaded in the last 30 days. Object tables can generate signed URLs for each object they have access to. Signed URLs provide a mechanism to extend the BigLake governance umbrella outside BigQuery. For example, users can call out to a Cloud Function (registered in BigQuery as a remote user-defined function) which uses signed URLs to access and process objects directly from the object store.
- *Scalability.* Working with a large number of unstructured files is challenging because it runs into the same object store limitations outlined in Section 3.3. When listing billions of objects can take hours, a simple task such as maintaining an up-to-date list of assets for training or validation is difficult. Object tables store object store metadata as table data and thus inherit the scalability of BigQuery.

Object tables reuse much of the metadata caching mechanism described in Section 3.3. BigLake infrastructure automatically collects object store metadata into a cache. For plain structured tables the metadata cache is used to list and prune data files. For Object tables, the metadata cache is used directly as a data source by the SQL runtime. Each file in the metadata cache turns into an Object table row.

Object tables enable users to wrangle billions of objects in seconds. For example, creating a 1% random sample of a large dataset of images can take hours with Python script calling object store APIs. With Object tables, it takes two lines of SQL and executes in seconds.

4.2 Inference and Integration with AI/ML

To process unstructured data, BigQuery ML[3] supports both inference within the query engine and outside the query engine by relying on external compute. In-engine inference utilizes the customer's Dremel compute footprint. As a result, in-engine inference jobs benefit from Dremel's ability to autoscale quickly and transparently to react to bursty workloads. The downside is that the maximum model size is constrained by Dremel worker memory; models greater than 2GB cannot be loaded. In contrast, external inference is not limited by Dremel memory and can leverage specialized hardware accelerators. However, external AI services tend to be more limited in terms of auto scaling agility, and there is an extra communication cost to ship data back and forth.

Listing 1 shows an example of in-engine inference and Listing 2 leverages an external service to extract entities from documents. The query in Listing 1 applies the ResNet [26] image classification model to JPEG files. The query (1) reads an object table named `dataset1.files` and filters it down to JPEG images and that have been uploaded since Nov 1, 2023; (2) generates inference directly in BigQuery using a imported model named `dataset1.resnet50`; and (3) returns the object URI and the inferred class.

Listing 2 shows a query that uses a proprietary Document AI model to parse receipts. The Document AI model can be fine-tuned by users, thus creating a "processor". The query (1) registers a Cloud Document AI processor endpoint as a remote model; (2) executes inference remotely on all the files referenced by the object table named `mydataset.documents`; and (3) returns all the fields extracted from each document.

4.2.1 In-Engine Inference. BigQuery ML has the ability to load and run TensorFlow [9], TensorFlow Lite, and ONNX [13] models directly in the Dremel workers. Memory is a challenge Dremel workers have a relatively small amount of working memory A "unit" of unstructured data tends to be larger than a typical relational row and thus consumes more memory. Compounding the problem, both the model execution and parsing of complex unstructured data formats (e.g., JPEG) must be sandboxed for security purposes. The sandboxes add additional memory overhead.

Scheduling fairly and efficiently in a multi-tenant query engine is a challenge. Introducing high-memory workers to process unstructured data would have significantly increased the scheduling complexity. Instead, we chose to lean on Dremel's ability to scale horizontally very quickly using the following observation: model inputs are preprocessed into tensors (dense multi-dimensional arrays) before inference. The tensor sizes are much smaller than the initial input image and can be efficiently exchanged by workers.

[3]BigQuery ML: https://cloud.google.com/bigquery/docs/bqml-introduction

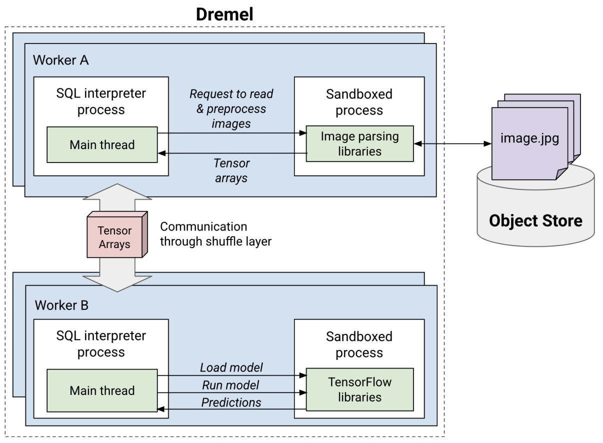


Figure 7: Distributing prepossessing and inference across workers

For example, preprocessing an image involves decoding the image format (e.g., JPEG), resizing the image to match to expected input size (e.g., 224*224), and converting color format appropriately.

Figure 7 provides an overview of distributed processing and inference in Dremel. We insert extra distributed operations in the query plan to schedule the preprocessing and the inference execution in different workers. This ensures that the raw images and the model are never loaded together in the same worker, minimizing the amount of worker memory required at the cost of extra communication between workers.

4.2.2 External Inference. BigQuery supports two forms of external inference: (1) using customer-owned models hosted on Vertex AI (Google Cloud model serving platform) and (2) inference using Google's first-party models. First-party models are accessed through a dedicated service and API such as Document AI, Vision AI, or Speech-to-Text. Inference using customer-owned models hosted on Vertex AI follows the same SQL script shown in Listing 1. The key difference is that the model, `dataset1.resnet50` in the example has been declared as a remote model instead of local one. In Dremel, external inference using customer-owned models relies on the infrastructure to execute UDFs remotely. Dremel reads the unstructured data files from the object store, preprocesses them into tensors matching the model input signature, and triggers inference against the model REST endpoint. The model outputs (predictions) are returned as raw JSON blobs that can then be parsed with SQL.

For first-party models (Listing 2), we built dedicated table value functions (TVFs) like `ML.PROCESS_DOCUMENT` to integrate with each AI vertical. Using TVFs allows us to offer a simpler user experience: we automatically post-process the model outputs to make them easier to work with in SQL, for example by flattening multiple levels of nested fields. This is only possible because the model output is part of a fixed API. To perform inferences, Dremel passes URIs and access tokens allowing the first-party model to read data from the object store directly. In contrast with customer-owned models, unstructured files are not read by Dremel at all when generating inferences with first-party models, which reduces communication across services.

5 OMNI

The rise of cloud computing has been one of the biggest trends in the industry over the course of the last two decades. Within that space, a major sub-trend has been the advent of multi-cloud. While many cloud customers intend to use a single cloud platform when starting out or migrating on-premise workloads, many end up using multiple platforms as a result of mergers and acquisitions (e.g., the acquired company uses another cloud). At the same time, some customers strategically choose to use multiple cloud providers from the beginning, for business, regulatory compliance, and risk management reasons. The pace of innovation in the cloud computing space is very high, and the choice of the best technology offered by different cloud providers is a rapidly moving target. Customers looking to use the best-of-breed technology may have a hard time doing so within the confines of a single cloud platform. All of these trends combine to drive up customer adoption of multi-cloud. Most cloud providers focus on creating first-party solutions that showcase their platforms, or help orchestrate migrations to their platforms, and there are not many first-party solutions available that embrace multi-cloud as a desirable permanent state.

BigQuery has long been the leading data analytics platform for Google Cloud. Some of the core characteristics of the BigQuery architecture are enabled by the Google internal technology stack that provides essential building blocks (e.g., Borg [37], Colossus [27], Spanner [12, 19]) for creating highly scalable services. These dependencies — and the lack of suitable replacements outside of Google — created a formidable barrier for making BigQuery available in another cloud. However, the rise of multi-cloud as a prevailing customer usage pattern creates a strong incentive to design a first-party solution that allows customers to analyze data residing in multiple clouds, without a need to replicate large datasets. The advancements with BigLake described earlier provide the technological foundation that enabled BigQuery Omni: a novel approach to cross-cloud data analytics. With Omni, customers can analyze data residing in BigQuery managed storage, Google Cloud Storage (GCS), Amazon S3, and Azure Blob Storage in-place, using a "single pane of glass" paradigm.

Omni launched generally available on AWS in 2020 and on Azure in early 2021. On the most fundamental level, Omni enables BigQuery's compute engine to run on all major cloud platforms (AWS, Azure, GCP) by bringing Dremel to the data, instead of asking customers to move or copy data across clouds to GCP to bring data to Dremel. While conceptually simple, achieving this goal required solving several technical challenges. As depicted in Figure 8, the implementation approach of Omni is running key pieces of the BigQuery data plane on foreign clouds using Kubernetes clusters, close to the data residing in cloud-specific object stores (e.g., S3, Azure Blob storage), while keeping the control plane on the GCP. It supports several use cases including ad-hoc analytics on data in-situ in other clouds, all the way to cross-cloud analytics that allow customers to query tables across different clouds in a seamless manner. The rest of this section describes the architecture of Omni and several of its novel features.

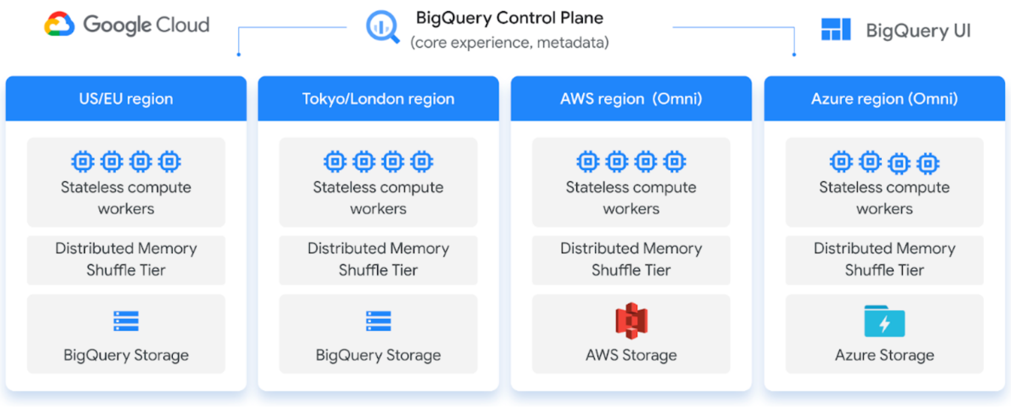


Figure 8: Omni High-Level Architecture

5.1 Architecture and Deployment

Omni uses a hybrid cross-cloud architecture consisting of many (20+) micro services connected via Stubby, Google's internal RPC framework that supports policy-based authorization [14]. Each micro service has its own set of authorization rules that define which other services can talk to it. These rules are defined statically, and remain constant throughout a deployment's lifetime.

Omni is a regional service with full regional isolation guarantees, with all Omni data and metadata stored within a region. Omni's deployment consists of two components: (1) binary deployments of pre-built binaries such as Dremel and (2) configuration deployments that consist of serialized configuration files. All binaries and config files are built from Google's Piper [32] source system similar to the rest of Google's software, which has processes in place to ensure every code submitted has appropriate reviewers and approval.

Omni follows a multi-phase rollout deployment model similar to BigQuery on GCP, where the deployment of binaries/configs progresses through one or more regions at a time. A set of validations are run and then the deployment proceeds to the next set of regions in a predetermined order. Config deployments are separate from binary deployment and usually follow a shorter time window for the entire deployment phase.

Omni's control plane consists of the same micro services that are used in BigQuery on GCP. This includes services for handling job management (the Job Server), query pre-processing, and other metadata management APIs. Users interact with standard BigQuery public APIs hosted by the control-plane. All query requests from users are handled by the Job Server. The Job Server does various preprocessing tasks, such as query validation, IAM-based authorization, metadata lookup, and then forwards the query to the data-plane on the foreign cloud.

Omni's data plane components run in AWS or Azure inside a Kubernetes cluster [15]. Data plane components include the Dremel query engine [30] and caching layer [31], and a few required infrastructure dependencies. RPC communication between services in the control and data-planes relies on Stubby, transported over a VPN connection. The data-plane also runs a few Borg [37] dependencies, such as Chubby [16], to provide a consistent runtime execution environment across all clouds, and make the components cloud-agnostic. Google also relies on various internal services, tools and pipelines to satisfy privacy/security and compliance requirements.

5.2 VPN/Networking

Given Omni's control and data plane split between GCP and foreign clouds, secure communication between the two planes is critical. Figure 9 provides an overview of how we ensure this secure channel. Omni uses a QUIC-based [28] zero-trust VPN. This VPN enables network endpoints hosted outside of Google production data centers to transparently communicate with services within Google. This VPN enables a Borg-like execution environment for services running in Kubernetes pods with a Low Overhead Authentication Service (or LOAS [23]), an internal Google framework that provides security for client-server communications.

This VPN is built on existing Google infrastructure components, such as Google Front Ends (GFEs) that provide services like public IP hosting, denial-of-service protection, and TLS termination that enable scale, ease of deployment and operations. A VPN client runs in Kubernetes pods and connects to a VPN terminator inside the Google network. The clients are load-balanced between a pool of terminators for each hosted region, and are allocated production IP addresses. Once the connection is established, all packets to/from Google production are exchanged over the encrypted connection that support both HTTP and Stubby protocols.

The VPN provides multiple layers of security. It enforces IP access control that drops any packets that are not in an allow-listed range. It is both a forward and a reverse network proxy, and verifies that all session-traffic is conformant to the underlying protocol. A policy engine is used to accept or deny inbound/outbound requests to services based on authenticated user information.

5.3 Security

Omni is a multi-tenant system that is built to provide the same level of security benefits as a single tenant architecture. Typically, single tenant systems provide a greater level of isolation in terms of security and privacy, at the cost of scaling flexibility for the customers. With our stringent security practices we aim to achieve

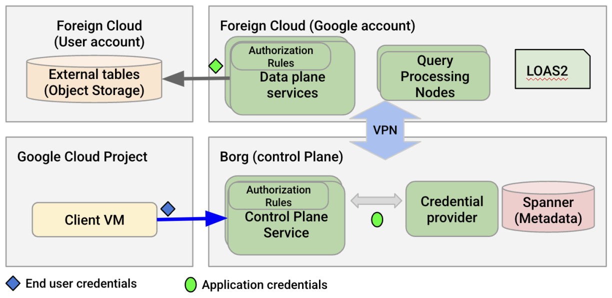


Figure 9: Omni VPN/Networking Infrastructure

the best of both worlds. To achieve this, we have built various defense-in-depth security features into the Omni architecture.

5.3.1 *Object access credentials scope (Per-Query Isolation).* When a query is run, it is sent to the BigQuery job server, which computes a superset of the object paths that the query will need access to by reading the table metadata. The Job Server obtains the necessary credentials for bucket access and scopes down the credentials to the exact paths that the query needs access to. These credentials are sent to Dremel worker nodes as part of the query execution pipeline. With these restrictions, the Dremel workers will only have access to the buckets/paths that the query is accessing. This provides isolation between the queries and limits the blast radius of a potential data breach to the set of tables accessed by that query.

5.3.2 *LOAS based addition authentication through untrusted proxy (Per Query/Regional isolation).* Dremel worker nodes running in Kubernetes in foreign clouds communicate with the control plane services running in Borg using LOAS protocol [23]. We built an additional security layer that acts as an untrusted proxy that sits in between Dremel and the Borg control plane. Each query uses a per-query session token which limits the scope of resources the query can access. The untrusted proxy terminates the LOAS protocol, validates the per-query session token, validates additional RPC parameters and allows the traffic only after all these checks are passed. This allows us to provide per-session isolation between the data plane nodes and the control plane components. If a Dremel node is compromised for any reason, any attempts to reach borg services outside the bounds of the query execution context will be denied by the untrusted proxy.

5.3.3 *Security Realms (Regional isolation).* All Google production services communicate with each other through Borg identity (aka a LOAS user). Authorization in Borg services is done through RPC security policy that allows explicit access to LOAS users/groups. For example, to allow Borg user A to talk to Borg user B, the latter must explicitly allow the former through RPC security policy (RPC-SP). This RPC-SP configuration is loaded at the start of service initialization and provides isolation between Borg services. We leverage the RPC-SP infrastructure in Omni by having separate user space for each Omni region. Each Omni region has a completely unique set of LOAS users that are not shared with any other BigQuery region. This provides geographic isolation for services running in the Omni region by preventing any unauthorized communication with other BigQuery regions (including other Omni regions).

5.3.4 *Human Authentication.* Google follows very stringent security practices [6] when it comes to human access to the service and user data stored in Google/Borg infrastructure. In Omni, we have extended the best practices in Borg infrastructure to apply to the foreign clouds. Omni has automated most tasks that access the user or service data. However, all production systems require a certain level of human access in order to diagnose and resolve issues in production. Any human access to production systems is fully audited and regulated through use of multi-factor authentication. Each Googler refreshes their access daily with a credential for their production identity and their corporate identity that is signed with their physical security key [38]. Our VM infrastructure trusts the Google-wide corporate SSH certificate authority for signing SSH credentials and provisions a set of trusted users on the VMs based on internally managed groups. This allows us to use an offline system for user authentication which is important when responding to an incident where services may be down. When a Googler escalates privilege on a machine, we re-authenticate with the SSH certificate through a Pluggable Authentication Module (PAM) to guard against escalation in the event of a container escape. All human access to production resources are logged to our internal Googler Internal Logging system which can be independently audited and verified.

5.3.5 *Binary Authorization.* The industry is becoming increasingly aware of supply chain [8] attacks, and the adoption of API driven containerized management systems only increases the vulnerability of launching arbitrary code. We take a layered approach to ensure the auditability of all code that runs in Omni. The operating system image is built within Google on the open source Container-Optimized OS project [4]. This image incorporates the minimal number of dependent open source software packages needed to operate the cluster, along with proprietary extensions and their checksum. We have pre-loaded a number of binaries to improve startup speed and also reduce any potential attack vectors. The primary containerized workloads are built inside Google and stamped with checksum and manifests which are verified at runtime through a Google API. This API verifies that the binaries are built with the verified, committed and reviewed source code. This allows us to enforce that no unexpected binaries can be run by Kubernetes, while avoiding the need to bake container images into the OS that would be problematic for upgrades and emergency patches.

5.4 Dremel on Non-GCP Clouds

Dremel runs natively on Borg infrastructure within Google. To move Dremel to a foreign cloud, a major challenge was bringing enough dependencies to the foreign cloud to ensure full functionality and performance guarantees. To achieve this, we built a minimal borg-like environment in AWS that consists of key services used by Dremel. This environment consists of (1) Chubby [16], Google's distributed lock service (2) Stubby [14], Google's internal RPC framework that supports policy-based authorization, (3) Envelope, a sidecar container equivalent to jobs running in Borg that provides various services like authentication and RPC translation, (4) Dremel's in-memory shuffle tier [31] to provide a distributed high performance file system for data exchange and checkpointing for dynamic query optimization, and (5) Pony [29], a high performance user-space host networking stack. We migrated all these

```
SELECT o.order_id, o.order_total, ads.id
  FROM local_dataset.ads_impressions AS ads
  JOIN aws_dataset.customer_orders AS o
  ON o.customer_id = ad.customer_id
```

Listing 3: Cross Cloud Query Example

services to AWS to make them run natively in AWS VPC. Most of the listed dependencies here are very low level services. One exception is Shuffle, which uses Spanner for metadata and state management. Since Spanner is not available in foreign clouds, we replaced it with databases that are natively available on the foreign clouds, which worked for the relatively smaller state-tracking needs of Shuffle (the larger scale transactions needed by the BigQuery catalog remain on the control plane on GCP) . We also locked down access to external networks through NAT gateways.

To achieve the maximum performance in foreign clouds, we run both TPC-H and TPC-DS tests and compare with other systems including BigQuery on GCP. We also have incorporated performance monitoring as part of our engineering culture so any new product release has to pass the performance runs to be release ready. With this, we were able to bring in all capabilities of Dremel to the foreign cloud without sacrificing its performance and flexibility.

5.5 Operating in a Multi-Cloud Environment

The services required to operate the query engine are managed by a large number of teams within Google. They are constantly being improved and require specialized knowledge to operate. We indexed heavily on providing the same release and support experience for these teams in order to reduce the cost of supporting Omni. This required building integrations or adapters into Google's release system, monitoring, logging, crash analysis, distributed tracing, internal DNS, and service discovery systems.

Creating bridges to these existing systems has proven to be very beneficial to the operation of Omni. We initially provided very little tooling to make the onboarding of new services smooth but found the large knowledge gap challenging for external teams. Creating teams tasked with cluster infrastructure and application management encapsulated the knowledge of the differences between Borg and Kubernetes which resulted in solutions that eased onboarding.

In building multi-cloud systems, there is no right answer on how cloud agnostic to go. The design should share as much as feasible, but have clear cutouts for cloud-specific configuration and functionality. We started with individual VM management as a lowest common denominator but quickly ran into reliability issues. Different cloud platforms have their own strengths that you do not want to lock yourself out of in the name of consistency. For example, the AWS AutoScalingGroup is a powerful primitive for reducing compute stock outs and optimizing cost for burstable workloads.

5.6 Cross Cloud Analytics

Data analytics users commonly have data in multiple clouds and regions. There are tangible business needs to bring this data together, but the fragmentation of this data across locales and storage systems makes it challenging. Traditional approaches to dealing with multiple data silos typically involve running multiple ETL pipelines to maintain multiple data copies, an approach that has several downsides, including data staleness, egress and data storage cost, complexity, and compliance (e.g. data wipeout) concerns. Many systems have tried to take a federated approach to data where a query is dispatched from one system to another, with results streaming back to a central point. Omni takes a different approach, with the same query engine being colocated with the data and operating on the primary data copy *in situ*, enabling optimizations that reduce egress costs and increase performance and efficiency. We describe two Omni innovations in this space through both cross-cloud queries and cross-cloud materialized views.

5.6.1 Cross Cloud Queries. Most databases and data analytics engines do not allow a direct join between tables residing in different regions (or clouds), for several reasons. Some of the more serious complications are the need for a bandwidth-intensive and expensive cross-region data transfer in the general case, and the difficulty of accessing table metadata and data in other locations. The Omni architecture provides a way to overcome such complications by leveraging the availability of a fully managed query engine in multiple clouds. BigQuery users can execute a query that joins data from a GCP region with one or more Omni regions in a single SQL statement as shown in Listing 3. This functionality is one of the first commercially available offerings that break the regional and cloud barriers and enable easy cross-cloud analytics in public clouds. We leverage subquery pushdown, BigQuery cross-region metadata availability, and the high throughput streaming service along with secure VPN for the data movement between regions.

When a query reaches the server, we parse the query and identify if there are any tables that are located in remote regions. If all the referenced tables are in the same region, the query proceeds to execute like a normal query. However, if we find table references to one or more regions that allow cross cloud queries, then we retrieve the remote table metadata, and split the query into regional subqueries with appropriate filter push down. We then submit new queries as cross-region `Create Table As Select` queries that run in the remote regions and push the data back into the local tables. The BigQuery query engine running locally in each region executes the corresponding query and pushes the results to the primary region using BigQuery's high throughput streaming APIs. This provides fast transfer of subquery results (typically a fraction of the total local table size thanks to filters pushed down to each region) back to the primary region. Once the remote queries are complete, all the remote data is available in temporary tables in the same region. The query is then rewritten to perform a regular join between local and temp tables.

5.6.2 Cross Cloud Materialized Views. A core use cases for Omni is supporting incremental cross cloud data transfer. Cross Cloud Materialized Views (CCMV), depicted in Figure 10, provide incremental replication of data from Omni region to GCP region by maintaining the replication state. CCMVs enable various ETL and dashboard use cases, along with seamlessly integrating a customer's foreign-cloud data with GCP-based services such as Vertex AI.

Omni CCMVs replicate incrementally to reduce egress costs. We first create a local materialized view in the foreign-cloud region with object storage as the storage medium. This local materialized view is periodically refreshed. If there are any new changes to the source, the new data is read and the materialized view is updated/appended

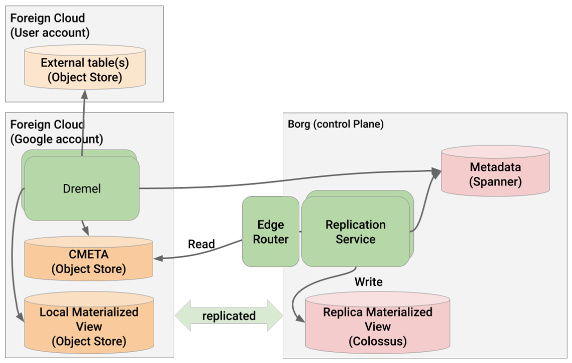


Figure 10: Cross-Cloud Materialized View Architecture

incrementally. For any upsert/delete in the source, we recreate the local materialized view partition that the source belonged to, avoiding re-replication of the entire materialized view. The materialized view replication process relies on stateful file based replication that copies the files from AWS S3 to Google's Colossus file system [27]. Network throughput is controlled through Google Cloud project quotas. Customers are charged based on the physical bytes copied.

Through the architecture of local and replica MV, we provide a single pane of glass with seamless access through the familiar BigQuery user interface for defining, querying and managing local and replica MVs across clouds. This reduces complexity of managing complex ETL pipelines especially when data is changing periodically. It significantly reduces egress costs of transferring data across clouds by only shipping incremental updates. It works out of the box, automatically refreshing and updating the cross-cloud MV based on the refresh interval specified by the user. Finally, it provides secured governed access to materialized view across clouds both for local analytics and cross-cloud analytics needs.

6 PRODUCTION USE CASES

BigQuery's evolution as a multi-cloud lakehouse has unlocked new possibilities for data management and has empowered our customers to handle previously challenging or impossible tasks. The following outlines several notable and repeatable customer usage patterns that we see in production:

Seamless Analytics on a Single Data Copy. Prior to BigQuery's transformation into a multi-cloud lakehouse, customer data remained isolated across data lakes and warehouses. While BigQuery could query data stored in cloud storage through external tables, performance limitations imposed by object store APIs and access restrictions to underlying buckets hindered its capabilities. BigLake introduced unified storage management across Cloud Storage and BigQuery storage, bringing enhanced performance and granular access control to open-format data stored in customer buckets. This breakthrough enables customers to store a single copy of data in either Cloud Storage or BigQuery storage while still running performant and secure analytics using BigQuery and open-source engines like Spark. Open-source engines can access BigLake tables using the BigQuery Storage API, leveraging fine-grained access control and query acceleration through metadata caching.

Cross-Cloud Query and Analysis. Before BigQuery's evolution as a multi-cloud lakehouse, performing cross-cloud analytics required physically moving data across clouds, a cumbersome and time-consuming process. BigQuery Omni now empowers customers to query data across clouds seamlessly using cross-cloud joins and maintains fine-grained access control through a unified BigQuery experience. Additionally, cross-cloud materialized views provide enhanced control and flexibility, facilitating efficient data querying.

Multi-modal Data Analysis with SQL Simplicity for Machine Learning. BigQuery's initial data format support was limited to structured and semi-structured data. With the introduction of BigLake Object tables, customers can now analyze unstructured data within BigQuery using the same governance framework employed for structured data. Customers can import vision models or call remotely hosted models in Vertex AI to perform document, speech, and vision analytics using the power and simplicity of SQL. Customers utilize these capabilities in a diverse array of applications, with prominent examples including:

- *Metadata Extraction.* Leveraging machine learning models to extract valuable metadata from unstructured data sources, enabling its seamless integration with structured data for comprehensive analytics.
- *Training Corpus Definition.* Effectively defining training corpus for large models by filtering for sensitive data or implementing sampling techniques. This ensures the protection of sensitive information while maintaining the integrity of the training dataset.
- *Granular Security Enforcement.* Implementing fine-grained security controls to govern access and sharing of unstructured data stored in the cloud. This safeguards sensitive information while facilitating authorized collaboration.

7 CONCLUSION

In summary, we presented the three key innovations that have evolved BigQuery toward a multi-cloud lakehouse. *BigLake tables* evolve pieces of core BigQuery storage and metadata infrastructure to unify managed warehouse data with open-source data lakes, providing uniform fine-grained governance, significant performance improvements to data lake tables, and ultimately fully managed BigLake tables that provide ACID transactions and other valuable features over data in customer-owned storage. BigQuery's support for unstructured data through *BigLake Object tables* is a novel approach to integrating data types like documents, images, and audio into an enterprise data warehouse. Our approach of implementing both in-engine and external inference techniques enable several novel data management use-cases spanning analytics and AI/ML. *Omni* allows our lakehouse features to enable multi-cloud use cases by seamlessly shipping Dremel and other key dependencies on non-GCP clouds with enterprise grade security. To provide customers with a seamless cross-cloud analytics experience, Omni implements cross-cloud queries and cross-cloud materialized views that minimizing cloud egress costs while still processing data in-situ. These innovations ultimately enable new workloads for customers that we hope will lead to even more innovation in the large-scale data analytics space.

REFERENCES

[1] 2023. *Apache Hudi.* https://hudi.apache.org/

[2] 2023. *Apache Iceberg: The open table format for analytic datasets.* https://iceberg.apache.org/

[3] 2023. *Apache Parquet.* https://parquet.apache.org/

[4] 2023. *Container Optimized OS.* https://cloud.google.com/container-optimized-os/docs

[5] 2023. *Google Cloud Dataproc.* https://cloud.google.com/dataproc

[6] 2023. *Google Infrastructure Security.* https://cloud.google.com/docs/security/infrastructure/design#operational-security

[7] 2023. *Spark BigQuery Connector.* https://github.com/GoogleCloudDataproc/spark-bigquery-connector

[8] 2023. *Supply Chain Levels for Software Artifacts.* https://slsa.dev

[9] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://download.tensorflow.org/paper/whitepaper2015.pdf

[10] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *PVLDB* 13, 12 (2020), 3411–3424.

[11] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD*. ACM, 2205–2217.

[12] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *SIGMOD*. 331–343.

[13] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[14] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly.

[15] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.

[16] Michael Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI*. 335–350.

[17] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-Ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil Mckay, Selcuk Aya, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *PVLDB* 12, 12 (2019), 2022–2034.

[18] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. 2011. Tenzing A SQL Implementation On The MapReduce Framework. *PVLDB* 4, 12 (2011), 1318–1327.

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*. 251–264.

[20] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.

[21] Pavan Edara, Jonathan Forbes, and Bigang Li. 2024. Vortex: A Stream-oriented Storage Engine For Big Data Analytics. In *SIGMOD*.

[22] Pavan Edara and Mosha Pasumansky. 2021. Big Metadata: When Metadata is Big Data. *PVLDB* 14, 12 (2021), 3083–3095.

[23] Cesar Ghali, Adam Stubblefield, Ed Knapp, Jiangtao Li, Benedikt Schmidt, and Julien Boeuf. 2017. *Application Layer Transport Security.* Technical Report. Google.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *SOSP*. 29–43.

[25] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Ganceanu, and Marc Nunkesser. 2012. Processing a Trillion Cells per Mouse Click. *PVLDB* 5, 11 (2012), 1436–1446.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[27] Dean Hildebrand and Denis Serenyi. 2021. *Colossus under the hood: a peek into Google's scalable storage system.* https://tinyurl.com/google-colossus

[28] Adam Langley et al. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM*. ACM, 183–196.

[29] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve D. Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena E. Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: a microkernel approach to host networking. In *SOSP*. 399–413.

[30] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB* 3, 1 (2010), 330–339.

[31] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *PVLDB* 13, 12 (2020), 3461–3472.

[32] Rachel Potvin and Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. *Commun. ACM* 59, 7 (2016), 78–87.

[33] Bart Samwel et al. 2018. F1 Query: Declarative Querying at Scale. *PVLDB* 11, 12 (2018), 1835–1848.

[34] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *ICDE*. 1802–1813.

[35] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Antony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB* 2, 2 (2009), 1626–1629.

[36] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*. 996–1005.

[37] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *EuroSys*. 1–17.

[38] Rory Ward and Betsy Beyer. 2014. BeyondCorp: A New Approach to Enterprise Security. *login Usenix Mag.* 39, 6 (2014).

[39] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct 2016), 56–65.