Mesa: A Geo-Replicated Online Data Warehouse for Google's Advertising System

By Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal

Abstract

Mesa is a highly scalable analytic data warehousing system that stores critical measurement data related to Google's Internet advertising business. Mesa is designed to satisfy a complex and challenging set of user and systems requirements, including near real-time data ingestion and retrieval, as well as high availability, reliability, fault tolerance, and scalability for large data and query volumes. Specifically, Mesa handles petabytes of data, processes millions of row updates per second, and serves billions of queries that fetch trillions of rows per day. Mesa is geo-replicated across multiple datacenters and provides consistent and repeatable query answers at low latency, even when an entire datacenter fails. This paper presents the Mesa system and reports the performance and scale that it achieves.

1. INTRODUCTION

Google runs an extensive advertising platform across multiple channels that serves billions of advertisements (or *ads*) every day to users all over the globe. Detailed information associated with each served ad, such as the targeting criteria, number of impressions and clicks, etc., are recorded and processed in real time. This data is used extensively at Google for different use cases, including reporting, internal auditing, analysis, billing, and forecasting. Advertisers gain fine-grained insights into their advertising campaign performance by interacting with a sophisticated front-end service that issues online and on-demand queries to the underlying data store. Google's internal ad serving platforms use this data in real time, determining budgeting and ad performance to enhance ad serving relevancy. As the Google ad platform continues to expand and as internal and external customers request greater visibility into their advertising campaigns, the demand for more detailed and fine-grained information leads to tremendous growth in the data size. The scale and business critical nature of this data result in unique technical and operational challenges for processing, storing, and querying. The requirements for such a data store are:

Atomic Updates. A single user action may lead to multiple updates at the relational data level, affecting thousands of consistent views, defined over a set of metrics (e.g., clicks and cost) across a set of dimensions (e.g., advertiser and country). It must not be possible to query the system in a state where only some of the updates have been applied.

Consistency and Correctness. For business and legal reasons, this system must return consistent and correct data. We require strong consistency and repeatable query results even if a query involves multiple datacenters.

Availability. The system must not have any single point of failure. There can be no downtime in the event of planned or unplanned maintenance or failures, including outages that affect an entire datacenter or a geographical region.

Near Real-Time Update Throughput. The system must support continuous updates, both new rows and incremental updates to existing rows, with the update volume on the order of millions of rows updated per second. These updates should be available for querying consistently across different views and datacenters within minutes.

Query Performance. The system must support latency-sensitive users serving live customer reports with very low latency requirements and batch extraction users requiring very high throughput. Overall, the system must support point queries with 99th percentile latency in the hundreds of milliseconds and overall query throughput of trillions of rows fetched per day.

Scalability. The system must be able to scale with the growth in data size and query volume. For example, it must support trillions of rows and petabytes of data. The update and query performance must hold even as these parameters grow significantly.

Online Data and Metadata Transformation. In order to support new feature launches or change the granularity of existing data, clients often require transformations of the data schema or modifications to existing data values. These changes must not interfere with the normal query and update operations.

Mesa is Google's solution to these technical and operational challenges for business critical data. Mesa is a distributed, replicated, and highly available data processing, storage, and query system for structured data. Mesa ingests data generated by upstream services, aggregates and persists the data internally, and serves the data via user queries. Even though this paper mostly discusses Mesa in the context

The original version of this paper, entitled "Mesa: Geo-Replicated, Near Real-Time, Scalable Warehousing," was published in the *Proceedings of the VLDB Endowment* 7, 12 (Aug. 2014), 1259–1270.

of ads metrics, Mesa is a generic data warehousing solution that satisfies all of the above requirements.

Mesa leverages common Google infrastructure and services, such as Colossus (Google's distributed file system),⁷ BigTable,³ and MapReduce.⁶ To achieve storage scalability and availability, data is horizontally partitioned and replicated. Updates may be applied at the granularity of a single table or across many tables. To achieve consistent and repeatable queries during updates, the underlying data is multi-versioned. To achieve update scalability, data updates are batched, assigned a new version number, and periodically (e.g., every few minutes) incorporated into Mesa. To achieve update consistency across multiple data centers, Mesa uses a distributed synchronization protocol based on Paxos.¹¹

In contrast, commercial DBMS vendors^{4, 14, 17} often address the scalability challenge through specialized hardware and sophisticated parallelization techniques. Internet services companies^{1, 12, 16} address this challenge using a combination of new technologies: key-value stores,^{3, 8, 13} columnar storage, and the MapReduce programming paradigm. However, many of these systems are designed to support bulk load interfaces to import data and can require hours to run. From that perspective, Mesa is very similar to an OLAP system. Mesa's update cycle is minutes and it continuously processes hundreds of millions of rows. Mesa uses multi-versioning to support transactional updates and queries across tables. A system that is close to Mesa in terms of supporting both dynamic updates and real-time querying of transactional data is Vertica.¹⁰ However, to the best of our knowledge, none of these commercial products or production systems has been designed to manage replicated data across multiple datacenters. Also, none of Google's other in-house data solutions^{2, 3, 5, 15} support the data size and update volume required to serve as a data warehousing platform supporting Google's advertising business.

Mesa achieves the required update scale by processing updates in batches. Mesa is, therefore, unique in that application data is redundantly (and independently) processed at all datacenters, while the metadata is maintained using synchronous replication. This approach minimizes the synchronization overhead across multiple datacenters in addition to providing additional robustness in face of data corruption.

2. MESA STORAGE SUBSYSTEM

Data in Mesa is continuously generated and is one of the largest and most valuable data sets at Google. Analysis queries on this data can range from simple queries such as, "How many ad clicks were there for a particular advertiser on a specific day?" to a more involved query scenario such as, "How many ad clicks were there for a particular advertiser matching the keyword 'decaf' during the first week of October between 8:00 am and 11:00 am that were displayed on google.com for users in a specific geographic location using a mobile device?"

Data in Mesa is inherently multi-dimensional, capturing all the microscopic facts about the overall performance of Google's advertising platform in terms of different dimensions. These facts typically consist of two types of attributes: *dimensional* attributes (which we call *keys*) and *measure*

attributes (which we call *values*). Since many dimension attributes are hierarchical (and may even have multiple hierarchies, e.g., the *date* dimension can organize data at the day/month/year level or fiscal week/quarter/year level), a single fact may be aggregated in multiple materialized views based on these dimensional hierarchies to enable data analysis using drill-downs and roll-ups. A careful warehouse design requires that the existence of a single fact is consistent across all possible ways the fact is materialized and aggregated.

2.1. The data model

In Mesa, data is maintained using *tables*. Each table has a *table schema* that specifies its structure. Specifically, a table schema specifies the *key space* K for the table and the corresponding *value space* V , where both K and V are sets. The table schema also specifies the *aggregation function* $F : V \times V \rightarrow V$ which is used to aggregate the values corresponding to the same key. The aggregation function must be associative (i.e., $F(F(v_0, v_1), v_2) = F(v_0, F(v_1, v_2))$) for any values $v_0, v_1, v_2 \in V$. In practice, it is usually also commutative (i.e., $F(v_0, v_1) = F(v_1, v_0)$), although Mesa does have tables with non-commutative aggregation functions (e.g., $F(v_0, v_1) = v_1$ to replace a value). The schema also specifies one or more *indexes* for a table, which are total orderings of K .

The key space K and value space V are represented as tuples of *columns*, each of which has a fixed type (e.g., int32, int64, string, etc.). The schema specifies an associative aggregation function for each individual value column, and F is implicitly defined as the coordinate-wise aggregation of the value columns, that is:

$$F((x_1, \dots, x_k), (y_1, \dots, y_k)) = (f_1(x_1, y_1), \dots, f_k(x_k, y_k)),$$

where $(x_1, \dots, x_k), (y_1, \dots, y_k) \in V$ are any two tuples of column values, and $f_1, \dots, f_k$ are explicitly defined by the schema for each value column.

As an example, Figure 1 illustrates three Mesa tables. All three tables contain ad click and cost metrics (value columns)

Figure 1. Three related Mesa tables.

Date	PublisherId	Country	Clicks	Cost
2013/12/31	100	US	10	32
2014/01/01	100	US	205	103
2014/01/01	200	UK	100	50

(a) Mesa table A

Date	AdvertiserId	Country	Clicks	Cost
2013/12/31	1	US	10	32
2014/01/01	1	US	5	3
2014/01/01	2	UK	100	50
2014/01/01	2	US	200	100

(b) Mesa table B

AdvertiserId	Country	Clicks	Cost
1	US	15	35
2	UK	100	50
2	US	200	100

(c) Mesa table C

broken down by various attributes, such as the date of the click, the advertiser, the publisher website that showed the ad, and the country (key columns). The aggregation function used for both value columns is *SUM*. All metrics are consistently represented across the three tables, assuming the same underlying events have updated data in all these tables. Figure 1 is a simplified view of Mesa's table schemas. In production, Mesa contains over a thousand tables, many of which have hundreds of columns, using various aggregation functions.

2.2. Updates and queries

To achieve high update throughput, Mesa applies updates in batches. The update batches themselves are produced by an upstream system outside of Mesa, typically at a frequency of every few minutes (smaller and more frequent batches would imply lower update latency, but higher resource consumption). Formally, an *update* to Mesa specifies a *version number* n (sequentially assigned from 0) and a set of *rows* of the form (table name, key, value). Each update contains at most one aggregated value for every (table name, key).

A *query* to Mesa consists of a version number n and a predicate P on the key space. The response contains one row for each key matching P that appears in some update with version between 0 and n . The value for a key in the response is the aggregate of all values for that key in those updates. Mesa actually supports more complex query functionality than this, but all of that can be viewed as pre-processing and post-processing with respect to this primitive.

As an example, Figure 2 shows two updates corresponding to tables defined in Figure 1 that, when aggregated, yield tables A, B, and C. To maintain table consistency (as discussed in Section 2.1), each update contains consistent rows for the two tables, A and B. Mesa computes

the updates to table C automatically, because they can be derived directly from the updates to table B. Conceptually, a single update including both the *AdvertiserId* and *PublisherId* attributes could also be used to update all three tables, but that could be expensive, especially in more general cases where tables have many attributes (e.g., due to a cross product).

Note that table C corresponds to a materialized view of the following query over table B: `SELECT SUM(Clicks), SUM(Cost) GROUP BY AdvertiserId, Country`. This query can be represented directly as a Mesa table because the use of *SUM* in the query matches the use of *SUM* as the aggregation function for the value columns in table B. Mesa restricts materialized views to use the same aggregation functions for metric columns as the parent table.

To enforce update atomicity, Mesa uses a multi-versioned approach. Mesa applies updates in order by version number, ensuring atomicity by always incorporating an update entirely before moving on to the next update. Users can never see any effects from a partially incorporated update.

The strict ordering of updates has additional applications beyond atomicity. Indeed, the aggregation functions in the Mesa schema may be non-commutative, such as in the standard key-value store use case where a (key, value) update completely overwrites any previous value for the key. More subtly, the ordering constraint allows Mesa to support use cases where an incorrect fact is represented by an inverse action. In particular, Google uses online fraud detection to determine whether ad clicks are legitimate. Fraudulent clicks are offset by *negative facts*. For example, there could be an update version 2 following the updates in Figure 2 that contains negative clicks and costs, corresponding to marking previously processed ad clicks as illegitimate. By enforcing strict ordering of updates, Mesa ensures that a negative fact can never be incorporated before its positive counterpart.

2.3. Versioned data management

Versioned data plays a crucial role in both update and query processing in Mesa. However, it presents multiple challenges. First, given the aggregatable nature of ads statistics, storing each version independently is very expensive from the storage perspective. The aggregated data can typically be much smaller. Second, going over all the versions and aggregating them at query time is also very expensive and increases the query latency. Third, naïve pre-aggregation of all versions on every update can be prohibitively expensive.

To handle these challenges, Mesa pre-aggregates certain versioned data and stores it using *deltas*, each of which consists of a set of rows (with no repeated keys) and a *delta version* (or, more simply, a *version*), represented by $[V_1, V_2]$, where V_1 and V_2 are update version numbers and $V_1 \leq V_2$. We refer to deltas by their versions when the meaning is clear. The rows in a delta $[V_1, V_2]$ correspond to the set of keys that appeared in updates with version numbers between V_1 and V_2 (inclusively). The value for each such key is the aggregation of its values in those updates. Updates are incorporated into Mesa as *singleton deltas* (or, more simply, *singletons*).

Figure 2. Two Mesa updates.

Date	PublisherId	Country	Clicks	Cost
2013/12/31	100	US	+10	+32
2014/01/01	100	US	+150	+80
2014/01/01	200	UK	+40	+20

(a) Update version 0 for Mesa table A

Date	AdvertiserId	Country	Clicks	Cost
2013/12/31	1	US	+10	+32
2014/01/01	2	UK	+40	+20
2014/01/01	2	US	+150	+80

(b) Update version 0 for Mesa table B

Date	PublisherId	Country	Clicks	Cost
2014/01/01	100	US	+55	+23
2014/01/01	200	UK	+60	+30

(c) Update version 1 for Mesa table A

Date	AdvertiserId	Country	Clicks	Cost
2013/01/01	1	US	+5	+3
2014/01/01	2	UK	+60	+30
2014/01/01	2	US	+50	+20

(d) Update version 1 for Mesa table B

The delta version $[V_1, V_2]$ for a singleton corresponding to an update with version number n is denoted by setting $V_1 = V_2 = n$.

A delta $[V_1, V_2]$ and another delta $[V_2 + 1, V_3]$ can be aggregated to produce the delta $[V_1, V_3]$, simply by merging row keys and aggregating values accordingly. (As discussed in Section 2.4, the rows in a delta are sorted by key, and, therefore, two deltas can be merged in linear time.) The correctness of this computation follows from associativity of the aggregation function F . Notably, correctness does not depend on commutativity of F , as whenever Mesa aggregates two values for a given key, the delta versions are always of the form $[V_1, V_2]$ and $[V_2 + 1, V_3]$, and the aggregation is performed in the increasing order of versions.

Mesa allows users to query at a particular version for only a limited time period (e.g., 24 hours). This implies that versions that are older than this time period can be aggregated into a *base delta* (or, more simply, a *base*) with version $[0, B]$ for some *base version* $B \geq 0$, and after that any other deltas $[V_1, V_2]$ with $0 \leq V_1 \leq V_2 \leq B$ can be deleted. This process is called *base compaction*, and Mesa performs it concurrently and asynchronously with respect to other operations (e.g., incorporating updates and answering queries).

Note that for compaction purposes, the time associated with an update version is the time that version was generated, which is independent of any time series information that may be present in the data. For example, for the Mesa tables in Figure 1, the data associated with 2014/01/01 is never removed. However, Mesa may reject a query to the particular depicted version after some time. The date in the data is just another attribute and is opaque to Mesa.

With base compaction, to answer a query for version number n , we could aggregate the base delta $[0, B]$ with all singleton deltas $[B + 1, B + 1]$, $[B + 2, B + 2]$, $\dots$, $[n, n]$, and then return the requested rows. Even though we run base compaction frequently (e.g., every day), the number of singletons can still easily approach hundreds (or even a thousand), especially for update intensive tables. In order to support more efficient query processing, Mesa maintains a set of *cumulative deltas* D of the form $[U, V]$ with $B < U < V$ through a process called *cumulative compaction*. These deltas can be used to find a *spanning set* of deltas $\{[0, B], [B + 1, V_1], [V_1 + 1, V_2], \dots, [V_k + 1, n]\}$ for a version n that requires significantly less aggregation than simply using the singletons. Of course, there is a storage and processing cost associated with the cumulative deltas, but that cost is amortized over all operations (particularly queries) that are able to use those deltas instead of singletons.

The *delta compaction policy* determines the set of deltas maintained by Mesa at any point in time. Its primary purpose is to balance the processing that must be done for a query, the latency with which an update can be incorporated into a Mesa delta, and the processing and storage costs associated with generating and maintaining deltas. More specifically, the delta policy determines: (i) what deltas (excluding the singleton) must be generated prior to allowing an update

version to be queried (synchronously inside the update path, slowing down updates at the expense of faster queries), (ii) what deltas should be generated asynchronously outside of the update path, and (iii) when a delta can be deleted.

An example of delta compaction policy is the *two-level* policy illustrated in Figure 3. Under this example policy, at any point in time there is a base delta $[0, B]$, cumulative deltas with versions $[B + 1, B + 10]$, $[B + 1, B + 20]$, $[B + 1, B + 30]$, $\dots$, and singleton deltas for every version greater than B . Generation of the cumulative $[B + 1, B + 10x]$ begins asynchronously as soon as a singleton with version $B + 10x$ is incorporated. A new base delta $[0, B']$ is computed approximately every day, but the new base cannot be used until the corresponding cumulative deltas relative to B' are generated as well. When the base version B changes to B' , the policy deletes the old base, old cumulative deltas, and any singletons with versions less than or equal to B' . A query then involves the base, one cumulative, and a few singletons, reducing the amount of work done at query time. Mesa currently uses a variation of the two-level delta policy in production that uses multiple levels of cumulative deltas.⁹

2.4. Physical data and index formats

Mesa deltas are created and deleted based on the delta compaction policy. Once a delta is created, it is immutable, and therefore there is no need for its physical format to efficiently support incremental modification.

The immutability of Mesa deltas allows them to use a fairly simple physical format. The primary requirements are only that the format must be space efficient, as storage is a major cost for Mesa, and that it must support fast seeking to a specific key, because a query often involves seeking into several deltas and aggregating the results across keys. To enable efficient seeking using keys, each Mesa table has one or more *table indexes*. Each table index has its own copy of the data that is sorted according to the index's order.

The details of the format itself are somewhat technical, so we focus only on the most important aspects. The rows in a delta are stored in sorted order in *datafiles* of bounded size (to optimize for filesystem file size constraints). The rows themselves are organized into *row blocks*, each of which is individually transposed and compressed. The transposition lays out the data by column instead of by row to allow for better compression. Since storage is a

Figure 3. A two-level delta compaction policy.

major cost for Mesa and decompression performance on read/query significantly outweighs the compression performance on write, we emphasize the compression ratio and read/decompression times over the cost of write/compression times when choosing the compression algorithm.

Mesa also stores an *index file* corresponding to each data file. (Recall that each data file is specific to a higher-level table index.) An index entry contains the *short key* for the row block, which is a fixed size prefix of the first key in the row block, and the offset of the compressed row block in the data file. A naïve algorithm for querying a specific key is to perform a binary search on the index file to find the range of row blocks that may contain a short key matching the query key, followed by a binary search on the compressed row blocks in the data files to find the desired key.

3. MESA SYSTEM ARCHITECTURE

Mesa is built using common Google infrastructure and services, including BigTable³ and Colossus.⁷ Mesa runs in multiple datacenters, each of which runs a single *instance*. We start by describing the design of an instance. Then we discuss how those instances are integrated to form a full multi-datacenter Mesa deployment.

3.1. Single datacenter instance

Each Mesa instance is composed of two subsystems: update/maintenance and querying. These subsystems are decoupled, allowing them to scale independently. All persistent metadata is stored in BigTable and all data files are stored in Colossus. No direct communication is required between the two subsystems for operational correctness.

Update/maintenance subsystem. The update and maintenance subsystem performs all necessary operations to ensure the data in the local Mesa instance is correct, up to date, and optimized for querying. It runs various background operations such as loading updates, performing table compaction, applying schema changes, and running table checksums. These operations are managed and performed by a collection of components known as the *controller/worker framework*, illustrated in Figure 4.

The *controller* determines the work that needs to be done and manages all *table metadata*, which it persists in the *metadata BigTable*. The table metadata consists of detailed state and operational metadata for each table,

including entries for all delta files and update versions associated with the table, the delta compaction policy assigned to the table, and accounting entries for current and previously applied operations broken down by the operation type.

The controller can be viewed as a large-scale table meta-data cache, work scheduler, and work queue manager. The controller does not perform any actual table data manipulation work; it only schedules work and updates the meta-data. At startup, the controller loads table metadata from a BigTable, which includes entries for all tables assigned to the local Mesa instance. For every known table, it subscribes to a metadata feed to listen for table updates. This subscription is dynamically updated as tables are added and dropped from the instance. New update metadata received on this feed is validated and recorded. The controller is the exclusive writer of the table metadata in the BigTable.

The controller maintains separate internal queues for different types of data manipulation work (e.g., incorporating updates, delta compaction, schema changes, and table checksums). For operations specific to a single Mesa instance, such as incorporating updates and delta compaction, the controller determines what work to queue. Work that requires globally coordinated application or global synchronization, such as schema changes and table checksums, are initiated by other components that run outside the context of a single Mesa instance. For these tasks, the controller accepts work requests by RPC and inserts these tasks into the corresponding internal work queues.

Worker components are responsible for performing the data manipulation work within each Mesa instance. Mesa has a separate set of worker pools for each task type, allowing each worker pool to scale independently. Mesa uses an in-house worker pool scheduler that scales the number of workers based on the percentage of idle workers available. A worker can process a large task using MapReduce.⁹

Each idle worker periodically polls the controller to request work for the type of task associated with its worker type until valid work is found. Upon receiving valid work, the worker validates the request, processes the retrieved work, and notifies the controller when the task is completed. Each task has an associated maximum ownership time and a periodic lease renewal interval to ensure that a slow or dead worker does not hold on to the task forever. The controller is free to reassign the task if either of the above conditions could not be met; this is safe because the controller will only accept the task result from the worker to which it is assigned. This ensures that Mesa is resilient to worker failures. A *garbage collector* runs continuously to delete files left behind due to worker crashes.

Since the controller/worker framework is only used for update and maintenance work, these components can restart without impacting external users. Also, the controller itself is sharded by table, allowing the framework to scale. In addition, the controller is stateless – all state information is maintained consistently in the BigTable. This ensures that

Figure 4. Mesa's controller/worker framework.

Mesa is resilient to controller failures, since a new controller can reconstruct the state prior to the failure from the meta-data in the BigTable.

Query subsystem. Mesa's query subsystem consists of *query servers*, illustrated in Figure 5. These servers receive user queries, look up table metadata, determine the set of files storing the required data, perform on-the-fly aggregation of this data, and convert the data from the Mesa internal format to the client protocol format before sending the data back to the client. Mesa's query servers provide a limited query engine with basic support for server-side conditional filtering and "group by" aggregation. Higher-level database engines such as MySQL and F1 use these primitives to provide richer SQL functionality such as join queries.

Mesa clients have vastly different requirements and performance characteristics. In some use cases, Mesa receives queries directly from interactive reporting front-ends, which have very strict low-latency requirements. These queries are usually small but must be fulfilled almost immediately. Mesa also receives queries from large extraction-type workloads, such as offline daily reports, that send millions of requests and fetch billions of rows per day. These queries require high throughput and are typically not latency sensitive (a few seconds/minutes of latency is acceptable). Mesa ensures that these latency and throughput requirements are met by requiring workloads to be labeled appropriately and then using those labels in isolation and prioritization mechanisms in the query servers.

The query servers for a single Mesa instance are organized into multiple *sets*, each of which is collectively capable of serving all tables known to the controller. By using multiple sets of query servers, it is easier to perform query server updates (e.g., binary releases) without unduly impacting clients, who can automatically failover to another set in the same (or even a different) Mesa instance. Within a set, each query server is in principle capable of handling a query for any table. However, for performance reasons, Mesa prefers to direct queries over similar data (e.g., all queries over the same table) to a subset of the query servers. This technique allows Mesa to provide strong latency guarantees by allowing for effective query server in-memory pre-fetching and caching of data stored in Colossus, while also allowing for excellent overall throughput by balancing load

Figure 5. Mesa's query processing framework.

across the query servers. On startup, each query server registers the list of tables it actively caches with a *global locator service*, which is then used by clients to discover query servers.

There are many optimizations involved in query processing.⁹ One important class is skipping unnecessary rows or deltas. Another is allowing a failed query to resume mid-stream from another query server, possibly in another data center.

3.2. Multi-datacenter deployment

Mesa is deployed in multiple geographical regions in order to provide high availability. Each instance is independent and stores a separate copy of the data. In this section, we discuss the global aspects of Mesa's architecture.

Consistent update mechanism. All tables in Mesa are multi-versioned, allowing Mesa to continue to serve consistent data from previous states while new updates are being processed. An upstream system generates the update data in batches for incorporation by Mesa, typically once every few minutes. As illustrated in Figure 6, Mesa's *committer* is responsible for coordinating updates across all Mesa instances worldwide, one version at a time. The committer assigns each update batch a new version number and publishes all metadata associated with the update (e.g., the locations of the files containing the update data) to the *versions database*, a globally replicated and consistent data store build on top of the Paxos¹¹ consensus algorithm. The committer itself is stateless, with instances running in multiple datacenters to ensure high availability.

Mesa's controllers listen to the changes to the versions database to detect the availability of new updates, assign the corresponding work to update workers, and report successful incorporation of the update back to the versions database. The committer continuously evaluates if *commit criteria* are met (specifically, whether the update has been incorporated by a sufficient number of Mesa instances across multiple geographical regions). The committer enforces the commit criteria across all tables in the update. This property is essential for maintaining consistency of related tables (e.g., a Mesa table that is a materialized view over another Mesa table). When the commit criteria are met, the committer declares the update's version number

Figure 6. Update processing in a multi-datacenter Mesa deployment.

to be the new *committed version*, storing that value in the versions database. New queries are always issued against the committed version.

Mesa's update mechanism design has interesting performance implications. First, since all new queries are issued against the committed version and updates are applied in batches, Mesa does not require any locking between queries and updates. Second, all update data is incorporated asynchronously by the various Mesa instances, with only meta-data passing through the synchronously replicated Paxos-based versions database. Together, these two properties allow Mesa to simultaneously achieve very high query and update throughputs.

New Mesa instances. As Google builds new datacenters and retires older ones, we need to bring up new Mesa instances. To bootstrap a new Mesa instance, we use a peer-to-peer load mechanism. Mesa has a special *load worker* (similar to other workers in the controller/worker framework) that copies a table from another Mesa instance to the current one. Mesa then uses the update workers to catch up to the latest committed version for the table before making it available to queries. During bootstrapping, we do this to load all tables into a new Mesa instance. Mesa also uses the same peer-to-peer load mechanism to recover from table corruptions.

4. EXPERIENCES AND LESSONS LEARNED

In this section, we briefly highlight the key lessons we have learned from building a large-scale data warehousing system over the past few years. We provide a list that is by no means exhaustive.

Cloud Computing and Layered Architecture. Mesa is a distributed cloud system that is built on top of other distributed cloud systems, such as Colossus and BigTable. Mesa's distributed architecture is clearly critical to its scalability. The layered approach is also crucial, as it has allowed us to focus on the key aspects of Mesa, delegating complexity to other systems where appropriate. Often this delegation has a performance cost, but we have found that we can leverage other aspects of the architecture to compensate. For example, when we built Mesa we migrated data from high-performance local disks to Colossus, compensating for the increased seek times by having query servers aggressively pre-fetch data with a lot of parallelism.

Application Level Assumptions. One has to be very careful about making strong assumptions about applications while designing large-scale infrastructure. For example, when designing Mesa's predecessor system, we made an assumption that schema changes would be very rare. This assumption turned out to be wrong, and we found that Mesa needed to support online schema changes that do not block either queries or updates, often without making extra copies of the data.⁹ Due to the constantly evolving nature of a live enterprise, products, services, and applications are in constant flux. Furthermore, new applications come on board either organically or due to acquisitions of other companies that need to be supported. In summary, the design should be as

general as possible with minimal assumptions about current and future applications.

Geo-Replication. Although we support geo-replication in Mesa for high data and system availability, we have also seen added benefit in terms of our day-to-day operations. In Mesa's predecessor system, when there was a planned maintenance outage of a datacenter, we had to perform a laborious operations drill to migrate a 24×7 operational system to another datacenter. Today, such planned outages, which are fairly routine, have minimal impact on Mesa.

Data Corruption and Component Failures. Data corruption and component failures are a major concern for systems at the scale of Mesa. Data corruptions can arise for a variety of reasons and it is extremely important to have the necessary tools in place to prevent and detect them. Similarly, a faulty component such as a floating-point unit on one machine can be extremely hard to diagnose. Due to the dynamic nature of the allocation of cloud machines to Mesa, it is highly uncertain whether such a machine is consistently active. Furthermore, even if the machine with the faulty unit is actively allocated to Mesa, its usage may cause only intermittent issues. Overcoming such operational challenges remains an open problem, but we discuss some techniques used by Mesa in Ref.⁹

Testing and Incremental Deployment. Mesa is a large, complex, critical, and continuously evolving system. Simultaneously maintaining new feature velocity and the health of the production system is a crucial challenge. Fortunately, we have found that by combining some standard engineering practices with Mesa's overall fault-tolerant architecture and resilience to data corruptions, we can consistently deliver major improvements to Mesa with minimal risk. Some of the techniques we use are: unit testing, private developer Mesa instances that can run with a small fraction of production data, and a shared testing environment that runs with a large fraction of production data from upstream systems. We are careful to incrementally deploy new features across Mesa instances. For example, when deploying a high-risk feature, we might deploy it to one instance at a time. Since Mesa has measures to detect data inconsistencies across multiple datacenters (along with thorough monitoring and alerting on all components), we find that we can detect and debug problems quickly.

5. MESA PRODUCTION METRICS

In this section, we report update and query processing performance metrics for Mesa's production deployment. We show the metrics over a 7-day period to demonstrate both their variability and stability. We also show system growth metrics over a multi-year period to illustrate how the system scales to support increasing data sizes with linearly increasing resource requirements, while ensuring the required query performance. Overall, Mesa is highly decentralized and replicated over multiple datacenters, using hundreds to thousands of machines at each datacenter for both update and query processing. Although we do not report the proprietary details of our deployment,

the architectural details that we do provide are comprehensive and convey the highly distributed, large-scale nature of the system.

5.1. Update processing

Figure 7 illustrates Mesa update performance for one data source over a 7-day period. Mesa supports hundreds of concurrent update data sources. For this particular data source, on average, Mesa reads 30–60 megabytes of compressed data per second, updating 3–6 million distinct rows and adding about 300 thousand new rows. The data source generates updates in batches about every 5 min, with median and 95th percentile Mesa commit times of 54 and 211 s. Mesa maintains this update latency, avoiding update backlog by dynamically scaling resources.

5.2. Query processing

Figure 8 illustrates Mesa's query performance over a 7-day period for tables from the same data source as above. Mesa executed more than 500 million queries per day for those tables, returning 1.7–3.2 trillion rows. The nature of these production queries varies greatly, from simple point lookups to large range scans. We report their average and 99th percentile latencies, which show that Mesa answers most queries within tens to hundreds of milliseconds. The large

Figure 7. Update performance for a single data source over a 7-day period.

Figure 8. Query performance metrics for a single data source over a 7-day period.

difference between the average and tail latencies is driven by multiple factors, including the type of query, the contents of the query server caches, transient failures and retries at various layers of the cloud architecture, and even the occasional slow machine.

In Figure 9, we report the scalability characteristics of Mesa's query servers. Mesa's design allows components to independently scale with augmented resources. In this evaluation, we measure the query throughput as the number of servers increases from 4 to 128. This result establishes linear scaling of Mesa's query processing.

5.3. Growth

Figure 10 illustrates the data and CPU usage growth in Mesa over a 24-month period for one of our largest production data sets. Total data size increased almost 500%, driven by update rate (which increased by over 80%) and the addition of new tables, indexes, and materialized views. CPU usage increased similarly, driven primarily by the cost of periodically rewriting data during base compaction, but also affected by one-off computations such as schema changes, as well as optimizations that were deployed over time. Figure 10 also includes fairly stable latency measurements by a monitoring tool that continuously issues synthetic point queries to Mesa that bypass the query server caches. In fact, throughout this period, Mesa answered user point queries with latencies consistent with those shown in Figure 8, while maintaining a similarly high rate of rows returned.

6. CONCLUSION

In this paper, we present an end-to-end design and implementation of a geo-replicated, near real-time, scalable data warehousing system called Mesa. The engineering design of Mesa leverages foundational research ideas in the areas of databases and distributed systems. In particular, Mesa supports online queries and updates while providing

Figure 9. Scalability of query throughput.

Figure 10. Growth and latency metrics over a 24-month period.

strong consistency and transactional correctness guarantees. It achieves these properties using a batch-oriented interface, guaranteeing atomicity of updates by introducing transient versioning of data that eliminates the need for lock-based synchronization of query and update transactions. Mesa is geo-replicated across multiple datacenters for increased fault-tolerance. Finally, within each datacenter, Mesa's controller/worker framework allows it to distribute work and dynamically scale the required computation over a large number of machines to provide high scalability.

Acknowledgments

We would like to thank everyone who has served on the Mesa team, including former team members Karthik Lakshminarayanan, Sanjay Agarwal, Sivasankaran Chandrasekar, Justin Tolmer, Chip Turner, and Michael Ballbach, for their substantial contributions to the design and development of Mesa. We are also grateful to Sridhar Ramaswamy for providing strategic vision and guidance to the Mesa team. Finally, we thank the anonymous reviewers, whose feedback significantly improved the paper.

References

1. Abouzeid, A., Bajda-Pawlikowski, K., et al. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *PVLDB* 2, 1 (2009), 922–933.
2. Baker, J., Bond, C., et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR* (2011). 223–234.
3. Chang, F., Dean, J., et al. Bigtable.
4. Cohen, J., Eshleman, J., et al. Online expansion of largescale data warehouses. *PVLDB* 4, 12 (2011), 1249–1259.
5. Corbett, J.C., Dean, J., et al. Spanner: Google's globally-distributed database. In *OSDI* (2012). 251–264.
6. Dean, J., Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
7. Fikes, A. Storage architecture and challenges. <http://goo.gl/pF6kmz>, 2010.
8. Glendenning, L., Beschastnikh, I., et al. Scalable consistency in scatter. In *SOSP* (2011). 15–28.
9. Gupta, A., Yang, F., et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *Vldb* (2014).
10. Lamb, A., Fuller, M., et al. The Vertica analytic database: C-Store 7 years later. *PVLDB* 5, 12 (2012), 1790–1801.
11. Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.
12. Lee, G., Lin, J., et al. The unified logging infrastructure for data analytics at Twitter. *PVLDB* 5, 12 (2012), 1771–1780.
13. Project Voldemort: A Distributed Database. <http://www.project-voldemort.com/voldemort/>.
14. SAP HANA. <http://www.saphana.com/welcome>.
15. Shute, J., Vingralek, R., et al. FI: A distributed SQL database that scales. *PVLDB* 6, 11 (2013), 1068–1079.
16. Thusoo, A., Shao, Z., et al. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD* (2010). 1013–1020.
17. Weiss, R. A technical overview of the oracle exadata database machine and exadata storage server. *Oracle White Paper: Oracle Corporation, Redwood Shores*, 2012.

Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong,

Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal, Google, Inc., Mountain View, CA.

Copyright held by authors/owner.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs International 4.0 License.

World-Renowned Journals from ACM

ACM publishes over 50 magazines and journals that cover an array of established as well as emerging areas of the computing field. IT professionals worldwide depend on ACM's publications to keep them abreast of the latest technological developments and industry news in a timely, comprehensive manner of the highest quality and integrity. For a complete listing of ACM's leading magazines & journals, including our renowned Transaction Series, please visit the ACM publications homepage: www.acm.org/pubs.

ACM Transactions on Interactive Intelligent Systems

ACM Transactions on Interactive Intelligent Systems (TIIS). This quarterly journal publishes papers on research encompassing the design, realization, or evaluation of interactive systems incorporating some form of machine intelligence.

ACM Transactions on Computation Theory

ACM Transactions on Computation Theory (ToCT). This quarterly peer-reviewed journal has an emphasis on computational complexity, foundations of cryptography and other computation-based topics in theoretical computer science.

PLEASE CONTACT ACM MEMBER SERVICES TO PLACE AN ORDER
 Phone: 1.800.342.6626 (U.S. and Canada)
 +1.212.626.0500 (Global)
 Fax: +1.212.944.1318
 (Hours: 8:30am–4:30pm, Eastern Time)
 Email: acmhelp@acm.org
 Mail: ACM Member Services
 General Post Office
 PO Box 30777
 New York, NY 10087-0777 USA

Association for
Computing Machinery

Advancing Computing as a Science & Profession

www.acm.org/pubs