Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing

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

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 queryability, 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 to determine budgeting and previously served ad performance to enhance present and future 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 transformation of the data schema or modifications to existing data values. These changes must not interfere with the normal query and update operations.

 Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 12

Mesa is Google's solution to these technical and operational challenges. Even though subsets of these requirements are solved by existing data warehousing systems, Mesa is unique in solving all of these problems simultaneously 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 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 next-generation distributed file system) [22, 23], BigTable [12], and MapReduce [19]. 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 [35].

Most commercial data warehousing products based on relational technology and data cubes [25] do not support continuous integration and aggregation of warehousing data every few minutes while providing near real-time answers to user queries. In general, these solutions are pertinent to the classical enterprise context where data aggregation into the warehouse occurs less frequently, e.g., daily or weekly. Similarly, none of Google's other in-house technologies for handling big data, specifically BigTable [12], Megastore [11], Spanner [18], and F1 [41], are applicable in our context. BigTable does not provide the necessary atomicity required by Mesa applications. While Megastore, Spanner, and F1 (all three are intended for online transaction processing) do provide strong consistency across geo-replicated data, they do not support the peak update throughput needed by clients of Mesa. However, Mesa does leverage BigTable and the Paxos technology underlying Spanner for metadata storage and maintenance.

Recent research initiatives also address data analytics and data warehousing at scale. Wong et al. [49] have developed a system that provides massively parallel analytics as a service in the cloud. However, the system is designed for multi-tenant environments with a large number of tenants with relatively small data footprints. Xin et al. [51] have developed Shark to leverage distributed shared memory to support data analytics at scale. Shark, however, focuses on in-memory processing of analysis queries. Athanassoulis et al. [10] have proposed the MaSM (materialized sort-merge) algorithm, which can be used in conjunction with flash storage to support online updates in data warehouses.

The key contributions of this paper are:

- We show how we have created a petascale data warehouse that has the ACID semantics required of a transaction processing system, and is still able to scale up to the high throughput rates required to process Google's ad metrics.
- We describe a novel version management system that batches updates to achieve acceptable latencies and high throughput for updates, as well as low latency and high throughput query performance.
- We describe a highly scalable distributed architecture that is resilient to machine and network failures within a single datacenter. We also present the geo-replicated architecture needed to deal with datacenter failures. The distinguishing aspect of our design is that application data is asynchronously replicated through independent and redundant processing at multiple datacenters, while only critical metadata is synchronously replicated by copying the state to all replicas. This technique minimizes the synchronization overhead in managing replicas across multiple datacenters, while providing very high update throughput.
- We show how schema changes for a large number of tables can be performed dynamically and efficiently without affecting correctness or performance of existing applications.
- We describe key techniques used to withstand the problems of data corruption that may result from software errors and hardware faults.
- We describe some of the operational challenges of maintaining a system at this scale with strong guarantees of correctness, consistency, and performance, and suggest areas where new research can contribute to improve the state of the art.

The rest of the paper is organized as follows. Section 2 describes Mesa's storage subsystem. Section 3 presents Mesa's system architecture and describes its multi-datacenter deployment. Section 4 presents some of the advanced functionality and features of Mesa. Section 5 reports our experiences from Mesa's development and Section 6 reports metrics for Mesa's production deployment. Section 7 reviews related work and Section 8 concludes the paper.

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 8:00am and 11:00am 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.

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

Figure 1: Three related Mesa tables

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* $F : V \times V \to V$ 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, i.e.:

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

where $(x_1, \ldots, x_k), (y_1, \ldots, y_k) \in V$ are any two tuples of column values, and $f_1, \ldots, f_k$ 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) 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.

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

Figure 2: Two Mesa updates

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*). 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).

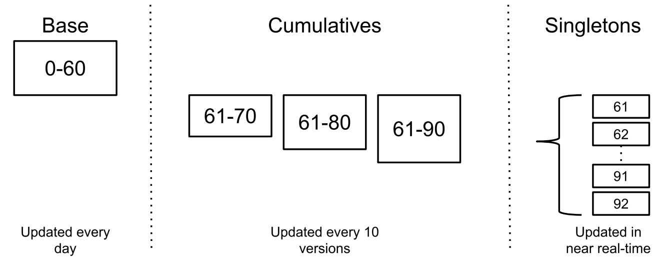

Figure 3: A two level delta compaction policy

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 $[B + 1, B + 1], [B + 2, B + 2], \ldots, [n, n]$, and then return the requested rows. Even though we run base expansion 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 $\{[0, B], [B + 1, V_1], [V_1 + 1, V_2], \ldots, [V_k + 1, n]\}$ 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 $[B + 1, B + 10], [B + 1, B + 20], [B + 1, B + 30], \ldots$, 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. For recent versions, the cumulative deltas compact a small number of singletons, and for older versions the cumulative deltas compact a larger number of versions. For example, a delta hierarchy may maintain the base, then a delta with the next 100 versions, then a delta with the next 10 versions after that, followed by singletons. Related approaches for storage management are also used in other append-only log-structured storage systems such as LevelDB [2] and BigTable. We note that Mesa's data maintenance based on differential updates is a simplified adaptation of differential storage schemes [40] that are also used for incremental view maintenance [7, 39, 53] and for updating columnar read-stores [28, 44].

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 *data files* 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 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 BigTable [12] and Colossus [22, 23]. 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.

3.1.1 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 metadata 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 metadata. 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.

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

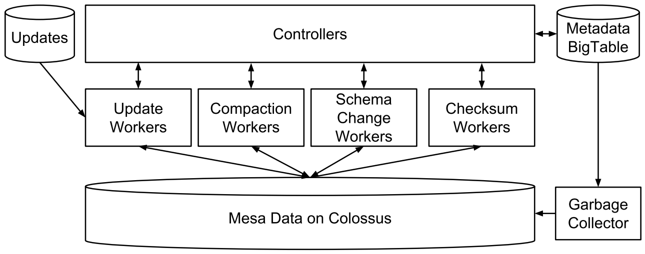

Figure 4: Mesa's controller/worker framework

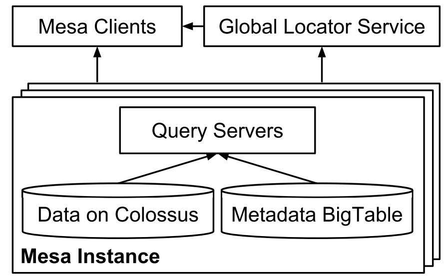

Figure 5: Mesa's query processing framework

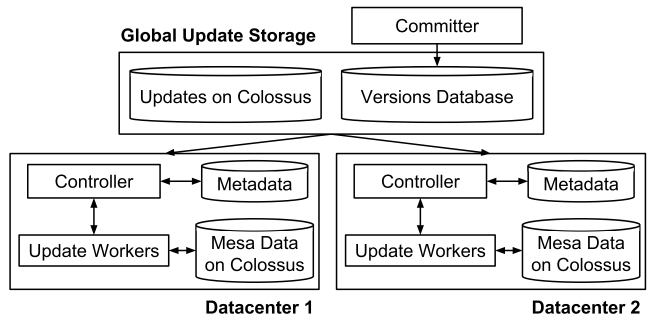

Figure 6: Update processing in a multi-datacenter Mesa deployment

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

3.1.2 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 [3], F1 [41], and Dremel [37] 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 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.

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.

3.2.1 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 Paxos [35] 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 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 metadata passing through the synchronously replicated Paxos-based versions database. Together, these two properties allow Mesa to simultaneously achieve very high query and update throughputs.

3.2.2 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. ENHANCEMENTS

In this section, we describe some of the advanced features of Mesa's design: performance optimizations during query processing, parallelized worker operations, online schema changes, and ensuring data integrity.

4.1 Query Server Performance Optimizations

Mesa's query servers perform *delta pruning*, where the query server examines metadata that describes the key range that each delta contains. If the filter in the query falls outside that range, the delta can be pruned entirely. This optimization is especially effective for queries on time series data that specify recent times because these queries can frequently prune the base delta completely (in the common case where the date columns in the row keys at least roughly correspond to the time those row keys were last updated). Similarly, queries specifying older times on time series data can usually prune cumulative deltas and singletons, and be answered entirely from the base.

A query that does not specify a filter on the first key column would typically require a scan of the entire table. However, for certain queries where there is a filter on other key columns, we can still take advantage of the index using the *scan-to-seek* optimization. For example, for a table with index key columns A and B, a filter $B = 2$ does not form a prefix and requires scanning every row in the table. Scan-to-seek translation is based on the observation that the values for key columns before B (in this case only A) form a prefix and thus allow a seek to the next possibly matching row. For example, suppose the first value for A in the table is 1. During scan-to-seek translation, the query server uses the index to look up all rows with the key prefix $(A = 1, B = 2)$. This skips all rows for which $A = 1$ and $B < 2$. If the next value for A is 4, then the query server can skip to $(A = 4, B = 2)$, and so on. This optimization can significantly speed up queries, depending on the cardinality of the key columns to the left of B.

Another interesting aspect of Mesa's query servers is the notion of a *resume key*. Mesa typically returns data to the clients in a streaming fashion, one block at a time. With each block, Mesa attaches a resume key. If a query server becomes unresponsive, an affected Mesa client can transparently switch to another query server, resuming the query from the resume key instead of re-executing the entire query. Note that the query can resume at any Mesa instance. This is greatly beneficial for reliability and availability, especially in the cloud environment where individual machines can go offline at any time.

4.2 Parallelizing Worker Operation

Mesa's controller/worker framework consists of a controller that coordinates a number of different types of Mesa workers, each of which is specialized to handle a specific operation that involves reading and/or writing Mesa data for a single Mesa table.

Sequential processing of terabytes of highly compressed Mesa table data can routinely take over a day to complete for any particular operation. This creates significant scalability bottleneck in Mesa as table sizes in Mesa continue to grow. To achieve better scalability, Mesa typically uses the MapReduce framework [19] for parallelizing the execution of different types of workers. One of the challenges here is to partition the work across multiple mappers and reducers in the MapReduce operation.

To enable this parallelization, when writing any delta, a Mesa worker samples every s-th row key, where s is a parameter that we describe later. These row key samples are stored alongside the delta. To parallelize reading of a delta version across multiple mappers, the MapReduce launcher first determines a spanning set of deltas that can be aggregated to give the desired version, then reads and merges the row key samples for the deltas in the spanning set to determine a balanced partitioning of those input rows over multiple mappers. The number of partitions is chosen to bound the total amount of input for any mapper.

The main challenge is to define s so that the number of samples that the MapReduce launcher must read is reasonable (to reduce load imbalance among the mappers), while simultaneously guaranteeing that no mapper partition is larger than some fixed threshold (to ensure parallelism). Suppose we have m deltas in the spanning set for a particular version, with a total of n rows, and we want p partitions. Ideally, each partition should be of size n/p. We define each row key sample as having *weight* s. Then we merge all the samples from the deltas in the spanning set, choosing a row key sample to be a partition boundary whenever the sum of the weights of the samples for the current partition exceeds n/p. The crucial observation here is that the number of row keys in a particular delta that are not properly accounted for in the current cumulative weight is at most s (or 0 if the current row key sample was taken from this particular delta). The total error is bounded by $(m-1)s$. Hence, the maximum number of input rows per partition is at most $n/p + (m-1)s$. Since most delta versions can be spanned with a small value of m (to support fast queries), we can typically afford to set a large value for s and compensate for the partition imbalance by increasing the total number of partitions. Since s is large and determines the sampling ratio (i.e., one out of every s rows), the total number of samples read by the MapReduce launcher is small.

4.3 Schema Changes in Mesa

Mesa users frequently need to modify schemas associated with Mesa tables (e.g., to support new features or to improve query performance). Some common forms of schema change include adding or dropping columns (both key and value), adding or removing indexes, and adding or removing entire tables (particularly creating *roll-up* tables, such as creating a materialized view of monthly time series data from a previously existing table with daily granularity). Hundreds of Mesa tables go through schema changes every month.

Since Mesa data freshness and availability are critical to Google's business, all schema changes must be *online*: neither queries nor updates may block while a schema change is in progress. Mesa uses two main techniques to perform online schema changes: a simple but expensive method that covers all cases, and an optimized method that covers many important common cases.

The naïve method Mesa uses to perform online schema changes is to (i) make a separate copy of the table with data stored in the new *schema version* at a fixed update version, (ii) replay any updates to the table generated in the meantime until the new schema version is current, and (iii) switch the schema version used for new queries to the new schema version as an atomic controller BigTable metadata operation. Older queries may continue to run against the old schema version for some amount of time before the old schema version is dropped to reclaim space.

This method is reliable but expensive, particularly for schema changes involving many tables. For example, suppose that a user wants to add a new value column to a family of related tables. The naïve schema change method requires doubling the disk space and update/compaction processing resources for the duration of the schema change.

Instead, Mesa performs a *linked schema change* to handle this case by treating the old and new schema versions as one for update/compaction. Specifically, Mesa makes the schema change visible to new queries immediately, handles conversion to the new schema version at query time on the fly (using a default value for the new column), and similarly writes all new deltas for the table in the new schema version. Thus, a linked schema change saves 50% of the disk space and update/compaction resources when compared to the naïve method, at the cost of some small additional computation in the query path until the next base compaction. Linked schema change is not applicable in certain cases, for example when a schema change reorders the key columns in an existing table, necessitating a re-sorting of the existing data. Despite such limitations, linked schema change is effective at conserving resources (and speeding up the schema change process) for many common types of schema changes.

4.4 Mitigating Data Corruption Problems

Mesa uses tens of thousands of machines in the cloud that are administered independently and are shared among many services at Google to host and process data. For any computation, there is a non-negligible probability that faulty hardware or software will cause incorrect data to be generated and/or stored. Simple file level checksums are not sufficient to defend against such events because the corruption can occur transiently in CPU or RAM. At Mesa's scale, these seemingly rare events are common. Guarding against such corruptions is an important goal in Mesa's overall design.

Although Mesa deploys multiple instances globally, each instance manages delta versions independently. At the logical level all instances store the same data, but the specific delta versions (and therefore files) are different. Mesa leverages this diversity to guard against faulty machines and human errors through a combination of online and offline data verification processes, each of which exhibits a different trade-off between accuracy and cost. Online checks are done at every update and query operation. When writing deltas, Mesa performs row ordering, key range, and aggregate value checks. Since Mesa deltas store rows in sorted order, the libraries for writing Mesa deltas explicitly enforce this property; violations result in a retry of the corresponding controller/worker operation. When generating cumulative deltas, Mesa combines the key ranges and the aggregate values of the spanning deltas and checks whether they match the output delta. These checks discover rare corruptions in Mesa data that occur during computations and not in storage. They can also uncover bugs in computation implementation. Mesa's sparse index and data files also store checksums for each row block, which Mesa verifies whenever a row block is read. The index files themselves also contain checksums for header and index data.

In addition to these per-instance verifications, Mesa periodically performs global offline checks, the most comprehensive of which is a global checksum for each index of a table across all instances. During this process, each Mesa instance computes a strong row-order-dependent checksum and a weak row-order-independent checksum for each index at a particular version, and a global component verifies that the table data is consistent across all indexes and instances (even though the underlying file level data may be represented differently). Mesa generates alerts whenever there is a checksum mismatch.

As a lighter weight offline process, Mesa also runs a global aggregate value checker that computes the spanning set of the most recently committed version of every index of a table in every Mesa instance, reads the aggregate values of those deltas from metadata, and aggregates them appropriately to verify consistency across all indexes and instances. Since Mesa performs this operation entirely on metadata, it is much more efficient than the full global checksum.

When a table is corrupted, a Mesa instance can automatically reload an uncorrupted copy of the table from another instance, usually from a nearby datacenter. If all instances are corrupted, Mesa can restore an older version of the table from a backup and replay subsequent updates.

5. EXPERIENCES & 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. A key lesson is to prepare for the unexpected when engineering large scale infrastructures. Furthermore, at our scale many low probability events occur and can lead to major disruptions in the production environment. Below is a representative list of lessons, grouped by area, that is by no means exhaustive.

Distribution, Parallelism, and Cloud Computing. Mesa is able to manage large rates of data growth through its absolute reliance on the principles of distribution and parallelism. The cloud computing paradigm in conjunction with a decentralized architecture has proven to be very useful to scale with growth in data and query load. Moving from specialized high performance dedicated machines to this new environment with generic server machines poses interesting challenges in terms of overall system performance. New approaches are needed to offset the limited capabilities of the generic machines in this environment, where techniques which often perform well for dedicated high performance machines may not always work. For example, with data now distributed over possibly thousands of machines, Mesa's query servers aggressively pre-fetch data from Colossus and use a lot of parallelism to offset the performance degradation from migrating the data from local disks to Colossus.

Modularity, Abstraction and Layered Architecture. We recognize that layered design and architecture is crucial to confront system complexity even if it comes at the expense of loss of performance. At Google, we have benefited from modularity and abstraction of lower-level architectural components such as Colossus and BigTable, which have allowed us to focus on the architectural components of Mesa. Our task would have been much harder if we had to build Mesa from scratch using bare machines.

Capacity Planning. From early on we had to plan and design for continuous growth. While we were running Mesa's predecessor system, which was built directly on enterprise class machines, we found that we could forecast our capacity needs fairly easily based on projected data growth. However, it was challenging to actually acquire and deploy specialty hardware in a cost effective way. With Mesa we have transitioned over to Google's standard cloud-based infrastructure and dramatically simplified our capacity planning.

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. 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.

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.

Human Factors. Finally, one of the major challenges we face is that behind every system like Mesa, there is a large engineering team with a continuous inflow of new employees. The main challenge is how to communicate and keep the knowledge up-to-date across the entire team. We currently rely on code clarity, unit tests, documentation of common procedures, operational drills, and extensive cross-training of engineers across all parts of the system. Still, managing all of this complexity and these diverse responsibilities is consistently very challenging from both the human and engineering perspectives.

6. 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 seven 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

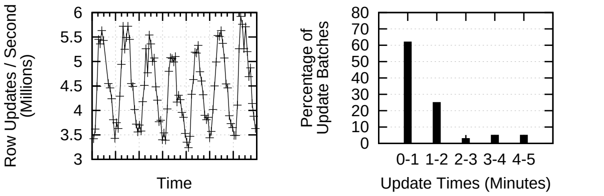

Figure 7: Update performance for a single data source over a seven day period

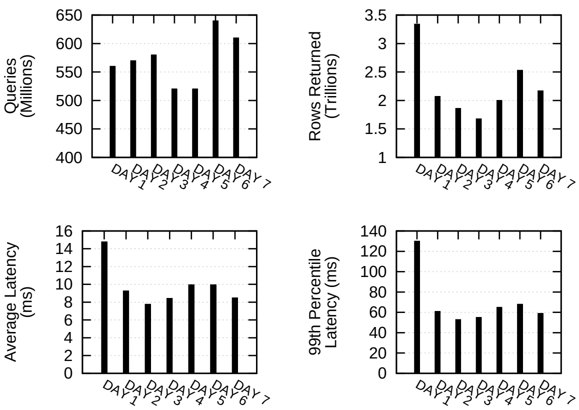

Figure 8: Query performance metrics for a single data source over a seven day period

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

6.1 Update Processing

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

6.2 Query Processing

Figure 8 illustrates Mesa's query performance over a seven 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 to 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 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.

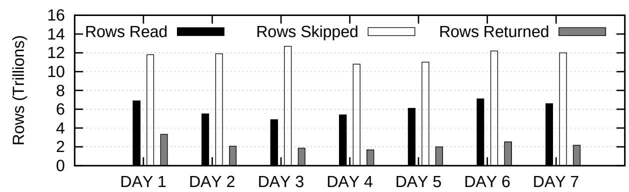

Figure 9: Rows read, skipped, and returned

Figure 10: Scalability of query throughput

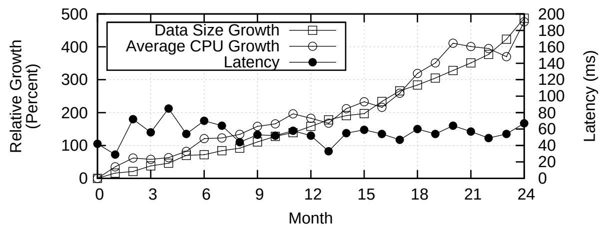

Figure 11: Growth and latency metrics over a 24 month period

Figure 9 illustrates the overhead of query processing and the effectiveness of the scan-to-seek optimization discussed in Section 4.1 over the same 7 day period. The rows returned are only about 30%-50% of rows read due to delta merging and filtering specified by the queries. The scan-to-seek optimization avoids decompressing/reading 60% to 70% of the delta rows that we would otherwise need to process.

In Figure 10, 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.

6.3 Growth

Figure 11 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 11 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.

7. RELATED WORK

Traditionally, RDBMS are widely used to manage structured data with strong consistency guarantees. However, they have difficulty with the scalability and performance required by modern data-driven applications. Key-value stores (also referred to as NoSQL systems) emerged to make non-relational storage systems highly scalable [1, 4, 12, 17, 20, 24]. Key-value stores achieve the required scalability by sacrificing transactional and strong consistency guarantees. Mesa explores a new point in the design space with high scalability, strong consistency, and transactional guarantees by restricting the system to be only available for batched and controlled updates that are processed in near real-time.

Data warehouses [9, 14] provide OLAP support for mining and analyzing large scale data. There exists an extensive body of research in this area: efficient heuristics for view selection [26, 27, 52], view maintenance [7, 15, 30, 39, 53], data cubes [25, 32, 42], schema evolution [31] and indexing [33, 38] and caching [21, 29, 43] in data warehouses. Much of this work is in the context of centralized architectures and mutable storage. We envision adapting some of these techniques in Mesa by extending them for the massively distributed architectures in the cloud, which in general provisions immutable storage using log-structured file-systems. Other industrial research groups have undertaken similar efforts for view maintenance over key-value stores [8].

In the commercial context, the demand for real-time and scalable data warehousing is constantly growing due to the increasing reliance on online and real-time analytics of business critical data. In the past few years, there has been an explosion of data volumes for both traditional enterprises as well as companies that provide internet-scale services. Industry leaders such as Teradata, SAP [5], Oracle [48] and EMC/Greenplum [16] have addressed this challenge by leveraging more powerful and parallel hardware in combination with sophisticated parallelization techniques in the underlying data management software. Internet services companies such as Twitter [36], LinkedIn [50], Facebook [45, 46, 47], Google [13, 37], and others [6] address the scalability challenge by leveraging a combination of new technologies: key-value stores, 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 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 [34]. However, to the best of our knowledge, none of these commercial products or production systems have been designed to manage replicated data across multiple datacenters. Furthermore, it is not clear if these systems are truly cloud enabled or elastic. They may have a limited ability to dynamically provision or decommission resources to handle load fluctuations.

None of Google's other in-house data solutions [11, 12, 18, 41] can support the data size and update volume required to serve as a data warehousing platform supporting Google's advertising business. Mesa achieves this scale by processing updates in batches. Each update takes a few minutes to commit and the metadata for each batch is committed using Paxos to achieve the same strong consistency that Megastore, Spanner and F1 provide. 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.

8. CONCLUSIONS

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 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.

Real-time analysis over vast volumes of continuously generated data (informally, "Big Data") has emerged as an important challenge in the context of database and distributed systems research and practice. One approach has been to use specialized hardware technologies (e.g., massively parallel machines with high-speed interconnects and large amounts of main memory). Another approach is to leverage cloud resources with batched parallel processing based on a MapReduce-like programming paradigm. The former facilitates real-time data analytics at a very high cost whereas the latter sacrifices analysis on fresh data in favor of inexpensive throughput.

In contrast, Mesa is a data warehouse that is truly cloud enabled (running on dynamically provisioned generic machines with no dependency on local disks), is geo-replicated across multiple datacenters, and provides strong consistent and ordered versioning of data. Mesa also supports petabyte-scale data sizes and large update and query workloads. In particular, Mesa supports high update throughput with only minutes of latency, low query latencies for point queries, and high query throughput for batch extraction query workloads.

9. 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.

10. REFERENCES

[1] HBase. http://hbase.apache.org/.

[2] LevelDB. http://en.wikipedia.org/wiki/LevelDB.

[3] MySQL. http:www.mysql.com.

[4] Project Voldemart: A Distributed Database. http://www.project-voldemort.com/voldemort/.

[5] SAP HANA. http://www.saphana.com/welcome.

[6] A. Abouzeid, K. Bajda-Pawlikowski, et al. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[7] D. Agrawal, A. El Abbadi, et al. Efficient View Maintenance at Data Warehouses. In *SIGMOD*, pages 417–427, 1997.

[8] P. Agrawal, A. Silberstein, et al. Asynchronous View Maintenance for VLSD Databases. In *SIGMOD*, pages 179–192, 2009.

[9] M. O. Akinde, M. H. Bohlen, et al. Efficient OLAP Query Processing in Distributed Data Warehouses. *Information Systems*, 28(1-2):111–135, 2003.

[10] M. Athanassoulis, S. Chen, et al. MaSM: Efficient Online Updates in Data Warehouses. In *SIGMOD*, pages 865–876, 2011.

[11] J. Baker, C. Bond, et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, pages 223–234, 2011.

[12] F. Chang, J. Dean, et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[13] B. Chattopadhyay, L. Lin, et al. Tenzing A SQL Implementation on the MapReduce Framework. *PVLDB*, 4(12):1318–1327, 2011.

[14] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[15] S. Chen, B. Liu, et al. Multiversion-Based View Maintenance Over Distributed Data Sources. *ACM TODS*, 29(4):675–709, 2004.

[16] J. Cohen, J. Eshleman, et al. Online Expansion of Largescale Data Warehouses. *PVLDB*, 4(12):1249–1259, 2011.

[17] B. F. Cooper, R. Ramakrishnan, et al. PNUTS: Yahoo!'s Hosted Data Serving Platform. *PVLDB*, 1(2):1277–1288, 2008.

[18] J. C. Corbett, J. Dean, et al. Spanner: Google's Globally-Distributed Database. In *OSDI*, pages 251–264, 2012.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[20] G. DeCandia, D. Hastorun, et al. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, pages 205–220, 2007.

[21] P. Deshpande, K. Ramasamy, et al. Caching Multidimensional Queries Using Chunks. In *SIGMOD*, pages 259–270, 1998.

[22] A. Fikes. Storage Architecture and Challenges. http://goo.gl/pF6kmz, 2010.

[23] S. Ghemawat, H. Gobioff, et al. The Google File System. In *SOSP*, pages 29–43, 2003.

[24] L. Glendenning, I. Beschastnikh, et al. Scalable Consistency in Scatter. In *SOSP*, pages 15–28, 2011.

[25] J. Gray, A. Bosworth, et al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tabs and Sub-Totals. In *IEEE ICDE*, pages 152–159, 1996.

[26] H. Gupta and I. S. Mumick. Selection of Views to Materialize Under a Maintenance Cost Constraint. In *ICDT*, 1999.

[27] V. Harinarayan, A. Rajaraman, et al. Implementing Data Cubes Efficiently. In *SIGMOD*, pages 205–216, 1996.

[28] S. Héman, M. Zukowski, et al. Positional Update Handling in Column Stores. In *SIGMOD*, pages 543–554, 2010.

[29] H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. Snakes and Sandwiches: Optimal Clustering Strategies for a Data Warehouse. In *SIGMOD*, pages 37–48, 1999.

[30] H. V. Jagadish, I. S. Mumick, et al. View Maintenance Issues for the Chronicle Data Model. In *PODS*, pages 113–124, 1995.

[31] A. Koeller and E. A. Rundensteiner. Incremental Maintenance of Schema-Restructuring Views in SchemaSQL. *IEEE TKDE*, 16(9):1096–1111, 2004.

[32] L. V. S. Lakshmanan, J. Pei, et al. Quotient cube: How to Summarize the Semantics of a Data Cube. In *VLDB*, pages 778–789, 2002.

[33] L. V. S. Lakshmanan, J. Pei, et al. QC-Trees: An Efficient Summary Structure for Semantic OLAP. In *SIGMOD*, pages 64–75, 2003.

[34] A. Lamb, M. Fuller, et al. The Vertica Analytic Database: C-Store 7 Years Later. *PVLDB*, 5(12):1790–1801, 2012.

[35] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[36] G. Lee, J. Lin, et al. The Unified Logging Infrastructure for Data Analytics at Twitter. *PVLDB*, 5(12):1771–1780, 2012.

[37] S. Melnik, A. Gubarev, et al. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB*, 3(1-2):330–339, 2010.

[38] N. Roussopoulos, Y. Kotidis, et al. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *SIGMOD*, pages 89–99, 1997.

[39] K. Salem, K. Beyer, et al. How To Roll a Join: Asynchronous Incremental View Maintenance. In *SIGMOD*, pages 129–140, 2000.

[40] D. Severance and G. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.*, 1(3):256–267, 1976.

[41] J. Shute, R. Vingralek, et al. F1: A Distributed SQL Database That Scales. *PVLDB*, 6(11):1068–1079, 2013.

[42] Y. Sismanis, A. Deligiannakis, et al. Dwarf: Shrinking the PetaCube. In *SIGMOD*, pages 464–475, 2002.

[43] D. Srivastava, S. Dar, et al. Answering Queries with Aggregation Using Views. In *VLDB*, pages 318–329, 1996.

[44] M. Stonebraker, D. J. Abadi, et al. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[45] A. Thusoo, J. Sarma, et al. Hive: A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[46] A. Thusoo, J. Sarma, et al. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *IEEE ICDE*, pages 996–1005, 2010.

[47] A. Thusoo, Z. Shao, et al. Data Warehousing and Analytics Infrastructure at Facebook. In *SIGMOD*, pages 1013–1020, 2010.

[48] R. Weiss. A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server. *Oracle White Paper. Oracle Corporation, Redwood Shores*, 2012.

[49] P. Wong, Z. He, et al. Parallel Analytics as a Service. In *SIGMOD*, pages 25–36, 2013.

[50] L. Wu, R. Sumbaly, et al. Avatara: OLAP for Web-Scale Analytics Products. *PVLDB*, 5(12):1874–1877, 2012.

[51] R. S. Xin, J. Rosen, et al. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, pages 13–24, 2013.

[52] J. Yang, K. Karlapalem, et al. Algorithms for Materialized View Design in Data Warehousing Environment. In *VLDB*, pages 136–145, 1997.

[53] Y. Zhuge, H. Garcia-Molina, et al. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *PDIS*, pages 146–157, 1996.