Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google

Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Zongchang (Jim) Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Zhi (Adam) Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Yalan (Maya) Meng, Prashant Mishra, Jay Patel, Rajesh S. R., Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Ye (Justin) Tang, Junichi Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Gensheng Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divyakant Agrawal, Jeff Naughton, Sujata Kosalge, Hakan Hacıgümüş
Google Inc

napa-paper@google.com

ABSTRACT

Google services continuously generate vast amounts of application data. This data provides valuable insights to business users. We need to store and serve these planet-scale data sets under the extremely demanding requirements of scalability, sub-second query response times, availability, and strong consistency; all this while ingesting a massive stream of updates from applications used around the globe. We have developed and deployed in production an analytical data management system, Napa, to meet these requirements. Napa is the backend for numerous clients in Google. These clients have a strong expectation of variance-free, robust query performance. At its core, Napa's principal technologies for robust query performance include the aggressive use of materialized views, which are maintained consistently as new data is ingested across multiple data centers. Our clients also demand flexibility in being able to adjust their query performance, data freshness, and costs to suit their unique needs. Robust query processing and flexible configuration of client databases are the hallmark of Napa design.

PVLDB Reference Format:
Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Zongchang (Jim) Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Zhi (Adam) Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Yalan (Maya) Meng, Prashant Mishra, Jay Patel, Rajesh S. R., Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Ye (Justin) Tang, Junichi Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Gensheng Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divyakant Agrawal, Jeff Naughton, Sujata Kosalge, Hakan Hacıgümüş. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. PVLDB, 14(12): 2986-2998, 2021.
doi:10.14778/3476311.3476377

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476377

Most of the related work in this area takes advantage of full flexibility to design the whole system without the need to support a diverse set of preexisting use cases. In comparison, a particular challenge we faced is that Napa needs to deal with hard constraints from existing applications and infrastructure, so we could not do a "green field" system, but rather had to satisfy existing constraints. These constraints led us to make particular design decisions and also devise new techniques to meet the challenges. In this paper, we share our experiences in designing, implementing, deploying, and running Napa in production with some of Google's most demanding applications.

1 INTRODUCTION

Google operates multiple services with more than a billion users around the globe. When providing these services, Google services rely on application data to provide better user experiences, to improve quality of service, and for billing. Google business users interact with this data through sophisticated analytical front-ends to gain insights into their businesses. These front-ends issue complex analytical queries over vast amounts of data and impose severe time constraints. In some cases, agreed-upon query response time goals are in the order of milliseconds. There are multiple petabytes of this data and it is continuously updated by a massive planetary-scale stream of updates. Users require the query results to be consistent and fresh, and demand continuous availability in the face of data center failures or network partitions. This paper describes *Napa*, an analytical data storage system that meets these challenging requirements.

There is a long history of innovations in OLAP (Online Analytical Processing) and Data Warehousing in the research literature and in industry. The majority of this work addresses a particular subset of requirements, such as scalability or high query performance and, often, the goal is to find an optimal or competitive solution while designing the whole solution from scratch. By contrast, when designing Napa, we had to address a comprehensive set of requirements, and we did not have the flexibility to start with a completely clean slate.

Napa was built to replace Mesa [19, 20], an earlier Google system. Napa has been operational for multiple years now. It inherited many

petabytes of historical data from Mesa and onboarded many new clients. While Mesa was built to serve a specific critical client with extreme latency requirements, Napa has a much broader mandate. We compare Mesa and Napa in the related work section; briefly, in contrast to Mesa, Napa was designed to be used Google-wide and to serve the diverse needs of many analytical applications. The following key aspects of Napa are the bedrock principles of its design and are aligned with our client requirements:

[Robust Query Performance] Consistent query performance is critical to data analytics users. Our clients expect low query latency, typically around a few hundreds of milliseconds, as well as low variance in latency regardless of the query and data ingestion load. Napa is able to guarantee robust query performance with consistent results despite daunting requirements for scale and system availability.

[Flexibility] While performance is important, our experience shows that it is not the only criterion for our clients. For instance, not all applications require millisecond response times, not all require the same freshness for ingested data, or for that matter, not all clients are willing to pay for "performance at any cost." Clients also require the flexibility to change system configurations to fit their dynamic requirements.

[High-throughput Data Ingestion] All Napa functions, including storage, materialized view maintenance, and indexing, must be performed under a massive update load. Napa implements a distributed table and view maintenance framework that is based on the LSM-tree (Log-Structured Merge-Tree) paradigm [25]. LSM is widely used in current generation of data warehouses and databases primarily to efficiently integrate and incorporate constantly emerging data into already existing data. Napa scales LSM to meet the challenges of Google's operating environment.

Napa's approach for robust query performance includes the aggressive use of materialized views, which are maintained consistently as new data is ingested across multiple data centers. This is in contrast to current trends in other systems that achieve performance by efficient scans of base data. Without indexed materialized views, delivering robust sub-second response times for the majority of our workloads is extremely difficult. The coverage of the materialized views for a query workload determines query performance, while the rate at which the views are refreshed affects freshness. Combined, varying how much of the workload is covered by views and how frequently they are refreshed provide levers by which clients can choose different cost/performance tradeoffs.

We hope that the goals for Napa, the constraints we faced, the decisions we made and techniques we developed, will be of general interest.

2 NAPA'S DESIGN CONSTRAINTS

Napa serves many applications in Google that differ in their requirements for three critical objectives (1) query performance, (2) data freshness, and (3) cost. The ideal, of course, is to achieve the highest query performance and highest data freshness at the lowest possible cost. We will use query performance and query latency interchangeably as for our purposes they are closely related (high performance implies low latency). Data freshness is measured by the time between when a row is added to a table to the time when it is available for querying. Freshness requirements range from a few minutes for freshness-sensitive clients to a couple of hours for cost-conscious clients. Costs are primarily machine resource costs that arise due to data processing – ingestion costs, background maintenance operations, and query execution. Typically, ingestion and maintenance costs dominate.

Of the three objectives, query performance has additional challenges. The clients not only care about low query execution latency; they also significantly care about predictable query performance, i.e., low variance in query latency. As an example, external Google reporting dashboards should continue to load with sub-second latency irrespective of the rate of newly arriving data. In other words, robust query performance is as important as raw query performance. Additionally, the queries may contain joins of one or more tables and are recurring in the sense that they are issued many times with different parameters.

2.1 Clients Need Flexibility

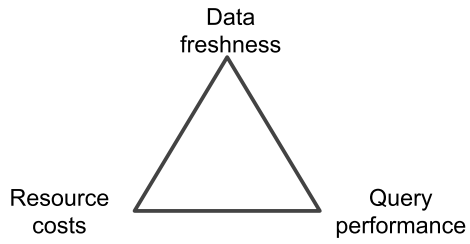

Figure 1: Three-way tradeoffs offered by Napa to maintain databases at an appropriate query performance, data freshness and cost.

Napa's clients can be categorized as making a three-way tradeoff between data freshness, resource costs, and query performance. Some clients need high freshness, while others may want to optimize for raw query performance or low costs.

An important consideration is the coupling of ingestion and storage. Here "ingestion" refers to data being presented to Napa and beginning its merge into the system, while "storage" refers to the new data having been applied to the base table and all materialized views it affects. One could couple ingestion to storage, meaning new data cannot be ingested until it has been fully processed. One could also couple new data with querying which would slow down query performance. Neither of these are acceptable to our clients as they lead to the following undesirable tradeoffs:

[Always sacrifice freshness] If ingestion is tightly coupled with storage, the ingestion can only go as fast as storage bandwidth. For example, an update in such a system will be committed only after it has been applied to the table and all its views. This design (used by Mesa) has the drawback that an additional view added to a table would make ingestion slower. Systems built around this design choice can offer high query performance but suffer from slower ingestion and may be forced to serve relatively stale data.

[Sacrifice query performance or consistency] View generation could be done opportunistically as part of querying such that on-the-fly view materialization is used to speed up subsequent queries (e.g., database cracking [21] and adaptive merging [17]). Asynchronous lazy models of maintaining views also exist (e.g., [1]) but those systems do not offer consistency between tables and their views. These schemes are not of much help for our client use-cases. A conceptual Napa system using this scheme could offer high freshness but would fail to meet the requirements around robust and high query performance.

As a result, Napa needs to provide clients the flexibility to tune the system and meet their goals around data freshness, resource costs, and query performance.

3 DESIGN CHOICES MADE BY NAPA

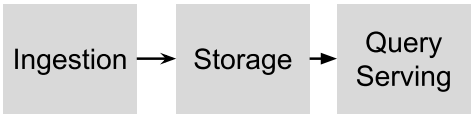

Figure 2: Conceptual Napa design consisting of three components.

Napa has to be highly scalable to process a stream of updates while simultaneously serving millions of queries with good performance. A key design choice in Napa is to rely on materialized views for predictable and high query performance.

Napa's high-level architecture consists of three main components as shown in the figure above.

(1) Napa's ingestion framework is responsible for committing updates into the tables. These updates are called deltas in Napa. The deltas written by the ingestion framework only serve to satisfy the durability requirements of the ingestion framework, and hence are write optimized. These deltas need to be further consolidated before they can be applied to tables and their associated views.
(2) The storage framework incrementally applies the updates to tables and their views. Napa tables and their views are maintained incrementally as log-structured merge-forests [25]. Thus, each table is a collection of updates. Deltas are constantly consolidated to form larger deltas; we call this process "compaction." The view maintenance layer transforms table deltas into view deltas by applying the corresponding SQL transformation. The storage layer is also responsible for periodically compacting tables and views.
(3) Query serving is responsible for answering client queries. The system performs merging of necessary deltas of the table (or view) at query time. Note that query latency is a function of the query time merge effort, so the faster the storage subsystem can process updates, the fewer deltas need to be merged at query time. F1 Query [27] is used as the query engine for data stored in Napa. We provide more details for query serving in Section 8.

Napa decouples ingestion from view maintenance, and view maintenance from query processing. This decoupling provides clients knobs to meet their requirements, allowing tradeoffs among freshness, performance, and cost. Note that Napa requires the consistency of the base tables and the views, so decoupling is a subtle yet important design choice that ensures Napa can keep making progress regardless of the performance of the individual components. Ingestion depends only on initial run generation, i.e., committing the updates, but not on merging or on view maintenance. Napa also provides clients with high level choices that translate to selectively indexing data and limits the amount of merging at query time.

As we discuss in the next section, with these design choices, Napa clients can choose "low effort" to optimize for cost, accepting reduced query performance. "Low effort" here means less aggressive compactions so that there is higher merging effort at query execution time. In a similar vein, low effort can also denote fewer materialized views, or reduced freshness, while still maintaining good query performance on queries that match the views. Similarly, clients can also choose to optimize for good query performance by paying for "higher effort," which results in low fan-in merges at query time, or can choose more targeted views.

3.1 Providing Flexibility to Clients

Users specify their requirements in terms of expected query performance, data freshness, and costs. These requirements are translated to internal database configurations such as the number of views, quota limits on processing tasks, the maximum number of deltas that can be opened during query processing, etc. These form a configuration of a client database at a point in time. However, the system is not static since data is constantly being ingested into the tables and one needs a dynamic yet easy to understand indicator of the database status in the context of the configuration generated from client requirements.

To that end, Napa introduces the concept called *Queryable Timestamp (QT)* to provide clients with a live marker (just like an advancing timestamp). QT is the direct indicator of freshness since *[Now() - QT]* indicates data delay. All data up to the QT timestamp can be queried by the client. Since QT can only be advanced when a required number of views have been generated with an upper-bound on the number of deltas, there is a guarantee that data used to serve query has met the conditions for delivering expected query performance. Furthermore, the continual advancement and staying within the freshness target of QT indicates the system is able to apply updates to the tables and views within the cost constraints specified in the database configuration. We discuss the QT concept in more detail in Section 6.

We illustrate three categories of Napa clients and how the system uses QT advancement criteria to tune Napa:

[Tradeoff freshness] Napa has a cost-conscious client that runs a Google-wide internal experimental analysis framework. For this client, good query performance and moderate costs are important, even if the system needs lower data freshness. For this client, Napa's QT advancement criteria is contingent on maintaining a moderate number of views and fewer deltas to merge at query execution time. To keep the cost low, Napa's execution framework uses fewer worker tasks and cheaper opportunistic machine resources for view maintenance. As a result, even though view maintenance occurs at

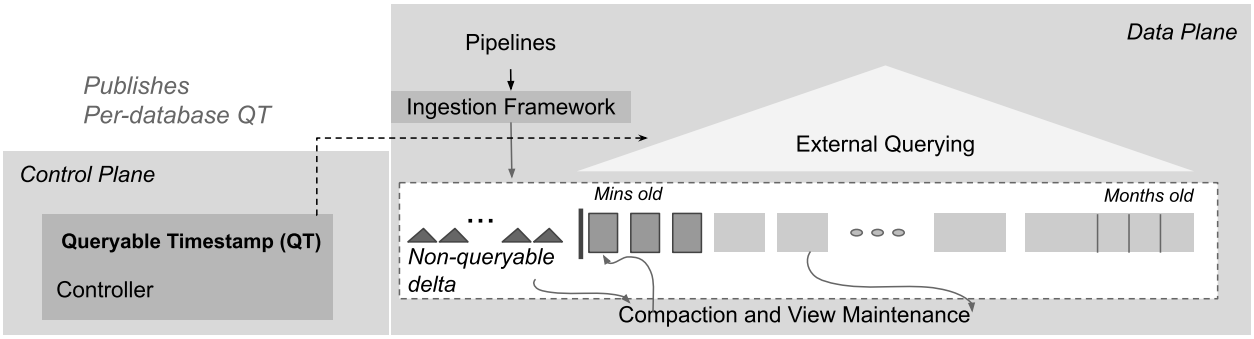

Figure 3: Napa architecture showing the major system components.

a slower pace, and hence data freshness is impacted, Napa provides this client good query performance at a moderate resource cost.

[Tradeoff query performance] Some Napa clients require fresh answers but have low or moderate query performance demands. For these clients, the QT advancement criteria is contingent on fewer views, but there can be relatively more deltas that need to be merged at query execution time. Since there are more deltas for each table and view, the query performance is lower. The query serving framework spends more time in I/O and collapses more rows, which would have otherwise happened offline during view maintenance and compaction. Napa's execution framework directs more workers for ingestion than view maintenance, since view maintenance effort is low. Therefore, these clients are able to trade query performance for better freshness and lower resource costs.

[Tradeoff costs] Napa has a client that powers Google external dashboards. For this client good query performance and data freshness are of utmost importance, even at higher costs. For such clients, Napa's QT advancement criteria is contingent on numerous views (sometimes 100s for a single table), and that the number of deltas at merge time is very low to ensure shorter query execution time. Napa uses a large number of worker tasks to ensure this QT criteria can be met quickly by both faster ingestion and high throughput view maintenance. This QT advancement criteria provides the client the desired query performance and data freshness, however, at a relatively high resource cost.

Such different categories of client requirements are part of the system configuration and Napa uses these configurations as a guidance to deliver the stipulated query performance, data freshness, and resource costs.

3.2 Data Availability

Over the past decade, most services within Google are architected to withstand data center scale outages that may result from catastrophic failures or scheduled maintenance. Google services, including Napa, provide the guarantee that the system remains operational in spite of such outages. The underlying paradigm to provide this level of fault-tolerance is to replicate client databases at multiple data centers and ensure that the database replicas are mutually consistent. A straightforward approach would be to execute Napa ingestion activity as synchronous transactions using a globally consistent transactional system, such as Google's Spanner [7]. Instead, Napa uses an approach in which it decouples the execution of data and metadata operations such that the data operations are executed asynchronously at each of the replicas at a data center and metadata operations are used periodically to ensure that the replicas remain synchronized with each other. In particular, relatively infrequent metadata operations use Spanner to ensure the mutual consistency of all the replicas. The orchestration of the synchronous and asynchronous mode of this highly distributed machinery is a key innovation in Napa. The queryable timestamp indicates a state at which all tables and views in a database are globally consistent across all data centers. Even though compaction and view maintenance are carried out asynchronously at each replica, the system moves from one consistent state to another.

4 SYSTEM ARCHITECTURE

Napa's high-level architecture consists of data and control planes as shown in Figure 3. The architecture is deployed at multiple data centers to manage the replicas at each data center. The data plane consists of ingestion, storage, and query serving. The control plane is made up of a controller that coordinates work among the various subsystems. The controller is also responsible for synchronizing and coordinating metadata transactions across multiple data centers. Napa clients create databases and tables along with their associated schemas. The clients can optionally create materialized views for each table.

Napa is built by leveraging the existing Google infrastructure components, which reduced the overall development cost but posed a challenge as some design choices had already been made for us. For instance, Napa is built on Google's Colossus File System [12, 14] with its disaggregated storage infrastructure. Hence, a table in Napa is a collection of files in Colossus. Napa uses Spanner for those functions that require strict transaction semantics, e.g., metadata management and storing system state. Napa uses F1 Query [27] for query serving and large scale data processing, such as view creation and maintenance. F1 Query is a prominent SQL-compliant query processing system within Google and serves as the query engine for data stored in Napa. F1 Query supports streaming and batch processing, which means that the same system can be used for interactive lookup queries as well as those that process large amounts of data. Note that the alternative to not using existing Google infrastructure was to develop these on our own, which would have been prohibitive in terms of engineering costs, not to mention the duplication of the effort.

Napa clients use ETL pipelines to insert data into their tables. The ingestion framework can sustain very high load, such as tens of GB/s of compressed data. Client data are delivered to any of the Napa replicas and Napa ensures that the data ingestion is incorporated at all the data centers. This significantly simplifies the design of ingestion pipelines.

Napa excels at serving workloads where clients issue aggregation queries with complex filters, e.g., those powering dashboards. As a result, the storage and view maintenance framework is a key component of Napa to maintain these aggregations. The storage framework is responsible for *compacting* tables and incrementally updating views. Compaction requires merging deltas, typically with a high fan-in, to create larger deltas, which reduces merge operations during online querying. This is similar to the post-processing in LSM-trees, where I/O spent by an offline process shifts work away from online querying.

Query serving deals with the necessary caching, prefetching and merging of deltas at run-time. The goal of query serving is to serve queries with low latency and low variance. Low latency is achieved by directing the queries to precomputed materialized views as opposed to the base table, and parallel execution of queries. Low variance is achieved by controlling the fan-in of the merges as well as a range of other I/O reduction and tail tolerance techniques.

Napa relies on views as the main mechanism for good query performance. Napa tables including the materialized views are sorted, indexed, and range-partitioned by their (multi-part) primary keys. This may be a surprising design choice given the recent trends in the database community favoring scan-based query processing. Napa's choice is largely motivated by the strict latency and resource requirements of its workloads, making it necessary to leverage indexed key lookups. Most Napa queries can effectively be answered by range-partitioned indexed tables. Note that range partitioning comes with its set of issues, such as hotspotting and load balancing due to temporal keys. For such cases, other partitioning schemes are also being investigated, but these are beyond the scope of this paper. The consequence of LSM and large-scale indexing means that Napa heavily relies on merging and sorting performance for efficiency. Hence, considerable effort was spent on speeding up sorting, merging, and group-by operators.

The Napa controller schedules compaction and view update tasks to keep the count of deltas for a table to a configurable value. These storage tasks are needed to keep the Queryable Timestamp (QT) as fresh as possible given the cost tradeoffs. The database QT forms the basis of freshness of a database and is used by the query system to provide robust query performance as described earlier. Napa supports database freshness of near-real time to a few hours; most clients require their databases to achieve approximately tens of minutes of freshness. If the freshness falls out of the desired range, the system continues to serve the client queries. However, the served data in that case would be stale as compared to the freshness requirements and administrative actions such as adjusting the tradeoff by temporarily allowing higher cost may be needed to bring the freshness back within the range. Napa has hundreds of databases with hundreds to thousands of tables and views each with a steady ingestion rate. Yet, the system is able to maintain all these databases at the desired freshness, which is a testament to the robustness of our design.

5 INGESTING TRILLIONS OF ROWS

The goal of the ingestion framework is to allow ingestion pipelines to insert large volumes of the data into Napa without significant overhead. Recall that one of Napa's key techniques is to decouple ingestion from view maintenance and indexing to provide clients trade offs across freshness, query performance, and costs. The ingestion framework contributes to this design via two mechanisms as is shown in Figure 4. First, the goal of the ingestion framework is to accept data, perform minimal processing, and make it durable without considering the pace of subsequent view maintenance. All ingested rows are assigned a metadata timestamp for ordering, and then marked as committed after other durability conditions, such as replication, have been satisfied. Second, the ingestion framework provides knobs to limit the peak machine costs by allowing configurations to increase or decrease the numbers of tasks that accept data and perform the ingestion work of batching, aggregating, and replicating.

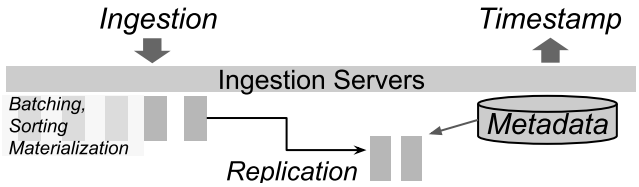

Figure 4: Napa ingestion is responsible for inserting updates to a table

Clients deliver the data to be ingested to any one of the Napa replicas and it is Napa's responsibility to ensure that the data is ingested at all the replicas to ensure availability. The ingestion framework produces write-optimized deltas, in that they are small and their physical sizes are limited by the memory buffer of servers. These deltas are not immediately available for querying since there are many of these deltas, which will slow down query serving because it has to merge them. We refer to these deltas as *unqueryable* and require that they be compacted before they can be queried.

6 QUERYABLE TIMESTAMP

The queryable timestamp (QT) of a table is a timestamp which indicates the freshness of data that can be queried. If QT(table) = X, all data that was ingested into the table before time X can be queried by the client and the data after time X is not part of the query results. In other words, the freshness of a table is [Now() - QT]. QT acts as a barrier such that any data ingested after X is hidden from client queries. The value of QT will advance from X to Y once the data ingested in (Y-X) range has been optimized to meet the query performance requirements. In turn, clients can use Napa's configuration options, and this single client visible metric to tune freshness, query performance, and costs. For example, if clients want high query performance and low costs, but can trade off freshness, the system prioritizes using fewer machine resources for view maintenance to reduce costs, and QT may progress slowly, thus indicating reduced data freshness.

An important criterion to ensure good query performance is to optimize the underlying data for reads and ensure views are available to speed up the queries. A table in Napa is a collection of

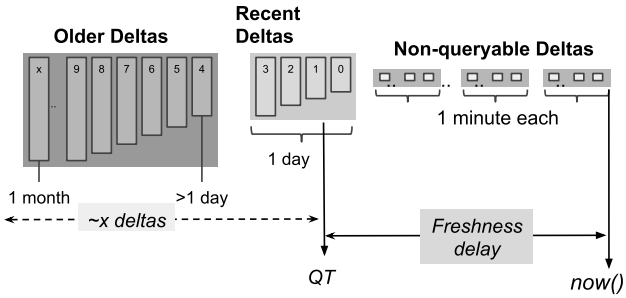

Figure 5: Queryable timestamp decouples query performance from storage performance.

all of its delta files, each delta corresponding to updates received for the table over a window of time, as indicated in the Figure 5. The non-queryable deltas correspond to newly received updates written by the ingestion framework in the most recent time window (typically seconds). The largest deltas, on the other hand, span a time window of weeks or even months. Each delta is sorted by its keys, range partitioned, and has a local B-tree like index. These deltas are merged as needed at query time. While Napa is a column store, it has to manage the dual concerns of maintaining views on the tables while achieving fast lookups. We borrow ideas from row-stores such as B-trees and PAX [2] layouts in our physical design to achieve these seemingly disparate goals.

Most client queries have tight latency constraints and this places hard limits on the maximum number (say, x) of deltas that should be opened and merged during query execution. In particular, the queryable timestamp (QT) is the delta which forms x's boundary, counting from the oldest delta towards the newest. Typically, this limit is a few tens of deltas, and is automatically configured depending on the query performance requirements on the database. An automated module dynamically adjusts this limit based on the query workload; tables with a high query workload and stringent query performance requirements have a lower limit but those with lesser demanding query requirements have a higher limit. There are some practical limitations on how large a number x can be supported. As that number gets larger, queries start getting affected by tail effects. Given that query time merging is quite expensive, by keeping the number of deltas for a given database near constant, Napa is able to provide robust query performance, i.e., a strong guarantee that the variance in query latency is low.

QT is essentially dependent on the progress of background operations such as compactions and incremental view maintenance. The QT of the database is the minimum of the QT of all the tables in the database. QT is also used to give clients a consistent view of data across all Napa replicas. Each replica has a local value of QT which is based on how fresh the data is in the local replica. The global value of QT is computed from the local QT values based on query serving availability requirements. For example, if we have 5 Napa replicas with the local QT values as 100, 90, 83, 75, 64 and query serving requires a majority of replicas to be available, then the new QT across all sites is set to 83 since the majority of the replicas are up to date at least up to 83. Napa will use the replicas whose QT is at least 83 to answer queries, as it is guaranteed that queries to these replicas only need to read locally available deltas.

7 MAINTAINING VIEWS AT SCALE

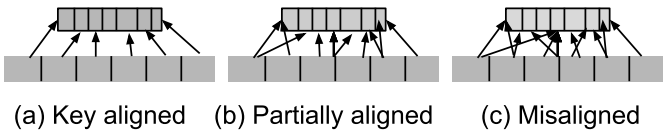

Figure 6: Classes of views based on the commonality of the view and base table key columns.

Napa's storage subsystem is responsible for maintaining views and compacting deltas. It is also responsible for ensuring data integrity, durability via replication across data centers, and handling outages from individual machines to entire data centers.

When building Napa, our aim was to ensure that the storage subsystem efficiently manages thousands of tables and views, routinely petabyte scale, even in the presence of data skew. While Napa supports materialized views that are joins of multiple tables, in the following, we discuss the challenges with views on single tables. The skew in view maintenance happens in the process of transforming the updates on the base tables into updates on the views. The mapping of the base table key space to the view key space may lead to discontinuities where most of the base table updates may map to a narrow view key range resulting in skews. As the QT of the database is determined by the slowest view or table, the system has to adjust automatically to the variations in size and aforementioned data skews to ensure that the QT is not susceptible to the straggler views or tables. The storage subsystem also adjusts to the cost budget by varying the number of views, tasks, and the type of machine resources used. In particular, the key aspects of the view maintenance framework includes the following:

[Use of F1 Query as a "data pump"] Napa's design choice is to use Google's F1 Query [27] as a relational data pump to compact tables and maintain views. The view maintenance uses the query optimizer that can make good choices among alternative plans as we show below in Section 7.1.

[Replanning to avoid data skews] The system can re-plan on the fly if it detects data skews. For example, the first key of many tables in Napa is a date column which has a few distinct values. Even though the base table may have hundreds of key columns, most of the key columns are mostly zero or have strong correlation with another key. At our scale, the failure to detect skews would mean that the view maintenance query may never finish resulting in unbounded freshness delays. This is a direct benefit of Napa using F1 Query as a data pump.

[Intelligence in the loop] A database can advance QT only if all the tables and views have caught up. This means that the QT is blocked by the slowest views and requires a fairly sophisticated straggler mitigation. The Napa controller implements the intelligence for tail mitigation. The principle techniques here are selecting data centers for task execution based on the historical load, active straggler task termination based on progress, and concurrent task execution to bound the size of the tails.

7.1 Query optimizations challenges in View Maintenance

Napa's view maintenance process effectively exploits data properties in the input. View update queries have to solve unique optimization challenges due to the amount of data processed and due to specific data properties (e.g., cardinality, sparseness, correlations) that complicate query processing at scale. Efficiently processing large amounts of data means that one has to be careful to not destroy beneficial data properties such as sortedness and partitioning, which are hard to recreate.

A concrete example of a data property is the sort order of the view to be updated vis-a-vis the base table. One approach is to re-sort the view keys based on the view sort order regardless of the base table sort order. Given our scale, this would be an expensive processing proposition. Instead, it is beneficial to preserve input sortedness as much as possible; exploit sortedness even if the view's sort order and the base table sort order only partially overlaps. Similarly, changing the data partitioning property requires moving data across the network, which typically also clobbers sort, and should be avoided unless absolutely necessary. These ideas are not new but rather extensions of "interesting orderings" in the database literature [28], which is our motivation behind using a SQL-compliant data processing system (i.e., F1 Query) as a relational pump to maintain views. Broadly speaking, there are three classes of views based on the cost of maintaining them as shown in Figure 6.

- The cheapest views to maintain in our framework are those that share a prefix with the base table. For example, the base table has keys (A, B, C), while the view is on (A, B). In this case, the framework avoids sorting completely by clustering the input based on common key prefix (i.e., A, B) and aggregating in a streaming manner.
- The second class of views are those that have a partial prefix with the base table but not a complete prefix. For example, the base table has (A, B, C, D) while the view is on (A, B, D). Even in this case, we are able to exploit the input sort order by clustering the input base table on (A,B) and then sorting on D for each of the groups of unique (A, B). Note that clustering on a partial prefix here and in the above example can result in skews, which needs to be detected and remedied.
- The third class of views are those where the base table and views do not share any prefix. For example, the base table is (A, B, C, D) while the view is (D, C, A). There are few opportunities for optimization and these views are the most expensive in practice since they require both re-partitioning and re-sorting.

Some views have a high aggregation reduction (even 100-1000x) when compared with the base table and hence the view updates are tiny compared to the original table update. There are also views that are nearly the same size as the base table. For views with high cardinality reduction, preserving the sort order is not paramount since the output is small enough that it might be feasible to focus exclusively on reducing the cardinality and re-sort the output if needed. On the other hand, for cases where views have low aggregation, i.e., the view is of similar size as the base table, sort and merge efficiency becomes important. Thus, we spent considerable

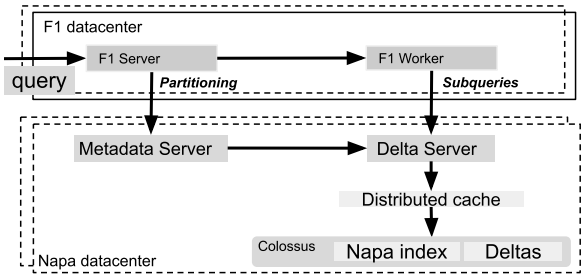

Figure 7: Mechanics of client query serving in Napa.

engineering effort in developing a state-of- the-art sort library for Napa. The same library is employed across all Napa components that sort data– from the ingestion servers to the sort operators in *F1 Query*. The principal techniques in our sorting and merging library are based on prior literature (e.g., [15, 22, 23]): normalized keys for efficient comparisons, poor man's normalized keys [15] for cache efficiency, tree-of-losers priority queues [23] for a minimal comparison count, and offset-value coding [22] for caching partial comparisons. The key accomplishment has been to implement these known techniques, tune the algorithms, and deploy the library for Google-scale processing.

7.2 Mechanics of Compaction

Compaction combines multiple input deltas into a single output delta. Compaction improves query performance and reduces storage consumption by 1) sorting inputs together and 2) aggregating multiple updates to the same rows. Compacting asynchronously with respect to querying both reduces merging work at query time and leverages the compacted result across multiple queries. However, compactions are expensive for high ingestion rate tables and they reduce data freshness by delaying when data becomes queryable. As mentioned earlier, the client's configuration controls this tradeoff. For example, a configuration that optimizes for query performance will compact frequently such that the maximum number of deltas merged at query time is less than 10, but such a configuration has significant ingestion delay and high compaction costs.

Since the delta files are individually sorted, compaction is essentially merge sorting. Unlike client queries where the fan-in of the merge is kept small and bounded to avoid tail effects, it is intentionally kept large during compaction so that the height of the merge tree is small, thus minimizing key comparisons. A compaction query has a fan-in of up to a thousand inputs beyond which the merge performance deteriorates. The merge process divides a fixed memory budget among the various inputs. At a thousand or so inputs, the memory per input stream is small. To add to that, the merge process stops when one of the inputs is consumed. At a thousand merge-way this happens 100x more frequently than it would happen at 10-way merge. The combination of these two effects make large merge-ways non-performant, which is remedied by I/O prefetching.

8 ROBUST QUERY SERVING PERFORMANCE

For many Napa clients, obtaining query results within the order of milliseconds is a critical requirement for their business use cases.

The strict latency requirement applies to tail cases (e.g. 99th percentile), for range lookups to petabyte sized tables, and even when the underlying shared infrastructure fluctuates in performance and availability. The section describes how the query serving subsystem achieves robust performance, using Queryable Timestamp (QT), materialized views, and a range of other techniques.

8.1 Reducing Data in the Critical Path

Napa uses multiple techniques to reduce the amount of data read to answer queries on the critical path. Whenever possible, Napa uses views to answer a query instead of the base table, since views with aggregation functions may have significantly less data. When F1 workers read data from Delta Servers (as shown in Figure 7), filters and partial aggregations are pushed down to minimize the amount of bytes transferred to F1 workers via the network. This is critical as F1 Query workers and Napa storage are not always collocated in the same data center, and cross data center network transfers tend to have larger variance in delay than intra-data center transfers. Napa also relies on parallelism to reduce the amount of data each subquery has to read. Napa maintains sparse B-tree indexes on its stored data, and uses them to quickly partition an input query into thousands of subqueries that satisfy the filter predicates. This partitioning mechanism additionally looks at the latency budget and availability of query serving resources to achieve good performance.

8.2 Minimizing Number of Sequential I/Os

Given the Google-scale datasets, and our reliance on shared and disaggregated storage, it is common to hit high latency if metadata (e.g., data statistics, view definitions, delta metadata) or data has to be read from disk or even SSDs. When a query is issued, Napa uses the value of QT to decide the version of metadata to be processed. The metadata in turns determines what data has to be processed. Therefore, metadata reads are on the critical path of query serving. Napa ensures all metadata can always be served from memory without contacting the persistent storage. This is achieved by affinity-based distributed metadata caching with periodic background refreshes. A particular QT is delayed to wait for the completion of periodic background refresh of metadata.

All data reads go through a transparent distributed data caching layer through which file I/O operations pass. The distributed cache is read-through and shares work on concurrent read misses of the same data. Sharing work is critical for the efficiency of the distributed cache: multiple Delta Servers often need to read an overlapping range of index files when processing different subqueries of the same query, and the distributed data caching makes sure such reads are processed only once.

The distributed caching layer significantly reduces the number of I/Os but cannot eliminate them, as the total working set size for Napa's query serving is significantly larger than the aggregated cache memory available. Therefore, Napa performs offline and online prefetching to further reduce the number of sequential I/Os in the critical path. Offline prefetching occurs as soon as data is ingested for frequently queried tables, before QT advances to make the new data available to query. Online prefetching starts when a query arrives and is performed by a shadow query executor which shares the data access pattern with the main query executor but skips all query processing steps. Since the shadow query executor skips processing, it runs ahead of the main query executor, achieving the effect of more accurate prefetching than disk readahead based on past accesses.

8.3 Combining Small I/Os

During query serving, Napa aggressively parallelizes the work by partitioning the query into fine grained units and then parallelizing I/O calls across deltas and across queried columns. However, parallelization comes with its own cost, especially with respect to tail latency. Suppose each Napa query issues 1000 parallel I/Os to disk. Napa's 90th percentile latency would be affected by the underlying disk storage's 99.99th percentile latency, which is often much higher than its 90th, 99th, and 99.9th percentile latency. To combat such amplification on tail latency, Napa uses QT to limit the number of queryable deltas. In addition, Napa also tries to combine small I/Os as much as possible, by using the following two techniques: lazy merging across deltas and size-based disk layout.

[Lazy merging across deltas] In a straightforward query plan, Napa exposes itself as a data source with primary key to the query optimizer. Each Delta Server, when processing a subquery, must first merge rows across all deltas based on the full primary key. When there are thousands (N) of subqueries and several tens (M) of deltas, the number of parallel I/Os are in the order of tens of thousands ($N \times M$). However, due to the parallelism each subquery reads very little data from most deltas. Meanwhile, a large fraction of Napa queries require merging based on a subset of primary keys in the subsequent phase of the query plan. In these cases, Napa adapts the query plan to avoid cross-delta merging in Delta Server and lets each Delta Server only process one delta, combining $N \times M$ parallel I/Os into close to N parallel I/Os.

[Size-based disk layout] Napa uses a custom-built columnar storage format supporting multiple disk layout options, which are applied based on delta sizes. The PAX layout [2], which can combine all column accesses into one I/O for lookup queries, is applied to small deltas. For large deltas, column-by-column layout is used that is I/O efficient for scan queries but requires one I/O per column for lookup queries. This size-based choice ensures that Napa receives columnar storage benefits as well as reduces I/O operations.

8.4 Tolerating Tails and Failures

Napa is built on Google infrastructure which is prone to performance and availability fluctuations due to its shared nature, especially at the tail. Napa adopts the principle of tolerating tail latency, rather than eliminating it, because eliminating all sources of variability for such a complex and interdependent system is infeasible [9]. For a non-streaming RPC, such as the RPC between Metadata Server and Delta Server, Napa uses the mechanism of *hedging*, which sends a secondary RPC identical to the original one to a different server after a certain delay, and waits for the faster reply. For a streaming RPC, such as the RPC between F1 worker and Delta Server, Napa estimates its expected progress rate and requires the server executing it periodically to report progress, together with a *continuation* token. If the reported progress is below expectation or the report is missing, the last continuation token would be used

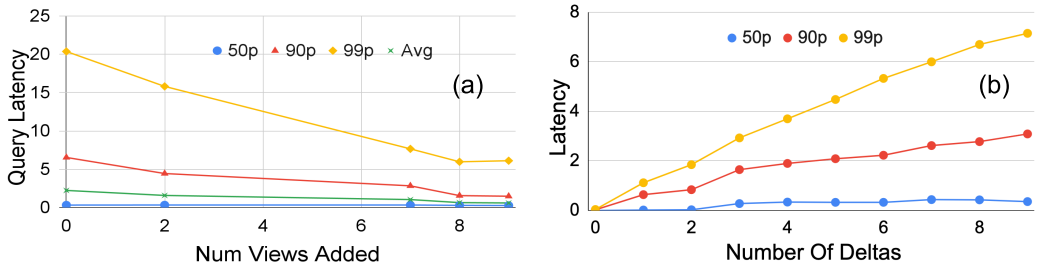

Figure 8: Figure shows (a) query latency reduction with the increasing number of views, (b) number of deltas that a query is allowed to span and the corresponding latency impact (lower is better).

to restart a new streaming RPC on a different server without losing progress. Pushdown operators like filtering and partial aggregation need to be carefully handled in progress reporting as they can significantly reduce the data size, causing progress reports to be superficially low or even missing. Napa uses bytes processed before filtering and partial aggregation as the progress rate metric and periodically forces these operators to flushes its internal state to generate a progress report with a continuation token.

For datacenter-wide issues which impact query serving but not ingestion, the above tail tolerance mechanisms would kick in and automatically reroute the queries to servers in a neighboring data center. When ingestion is impacted, datacenter-local QT is delayed in affected data centers and the query would be directly routed to other data centers based on the local QT values.

9 PRODUCTION METRICS INSIGHTS

Napa manages thousands of tables and views in production, where many tables are petabyte scale. It serves over a billion queries per day and ingests trillions of rows. In this section we discuss how Napa is able to provide robust query performance through three techniques: (1) by more actively using views, Napa reduces raw query performance and variance even at 99th percentile, (2) by changing storage policies, Napa can reduce the number of deltas and hence the tail latency, and (3) by decoupling ingesting, view maintenance, and query execution; Napa can mitigate the impact of infrastructure and workload changes on query performance. We also describe the workload characteristics of three production Napa clients who have differing freshness, query performance, and cost requirements. All figures in this section are from actual production data. We have normalized the units in the Y-axis of the graphs below to preserve the confidentiality of our business data. Note that we have chosen to only show those results where the relative improvements of the various curves in the graphs are more important than their absolute values.

9.1 Views and QT Help Achieve Robust Query Performance

First, most client queries are aggregation queries, and materialized views are typically at least an order of magnitude smaller than the base tables from which they are derived. Reading from views not only improves raw performance, but also improves tail latency as their smaller size is more cache friendly, and requires less compute resources, which reduces the chance of contention for query resources. Figure 8(a) is an example of the client's query latency at different percentiles with respect to the number of views. For this particular workload, by adding only two views, the client can improve their average and 90th percentile query performance by 1.5x. The most profound impact is at the 99th percentile query latency which keeps improving as we add up to 8 views. Beyond 8 views the query performance reaches a plateau as most queries are now able to use views instead of the base table.

Second, latency can be improved by reducing the number of deltas that have to be opened, read, and merged at query time. Figure 8(b) shows that as we change storage policies to reduce the number of deltas, the query latency improves significantly. For this workload, optimizations to lazily merge across deltas described in Section 8.3 cannot be applied because there are no subsequent aggregations in the query, because of which it is critical to limit the number of deltas. The biggest impact is at the 99th percentile latency which reduces by more than 3.6x as the number of deltas is changed from 8 to 2. The main reasons are: (1) fewer deltas means there are less number of small, parallel IOs which are prone to cause latency tails, (2) fewer deltas also means that data is premerged and aggregated, and less processing is required at query time.

9.2 Handling Infrastructure Issues

Figure 9 shows that Napa is able to guarantee its clients stable query performance even when the ingestion load changes or there are infrastructure outages. Figure 9 shows the workload of a client over a period of a few hours. Napa decouples ingestion from view maintenance and querying which allows us to optimize for low variance in query latency, in some cases by trading off data freshness. Figure 9(a) shows that the client continuously sends data to Napa, with some variance in the input rate over the course of the week. Figure 9(b) shows that the view maintenance performance dropped for the duration between X and Y indicating an infrastructure issue which affected the tasks updating views. However, the query serving latency remains near constant (Figure 9(d)) throughout the whole duration. In this particular example, client queries continued to be fast, however, for certain parts during the outage data freshness was impacted, as seen in Figure 9(c) where the value of delay is high.

9.3 Client Workloads

Figures 10(a)-(d) show how Napa provides clients the flexibility to optimize for different performance and cost metrics. These production clients have differing requirements as discussed below.

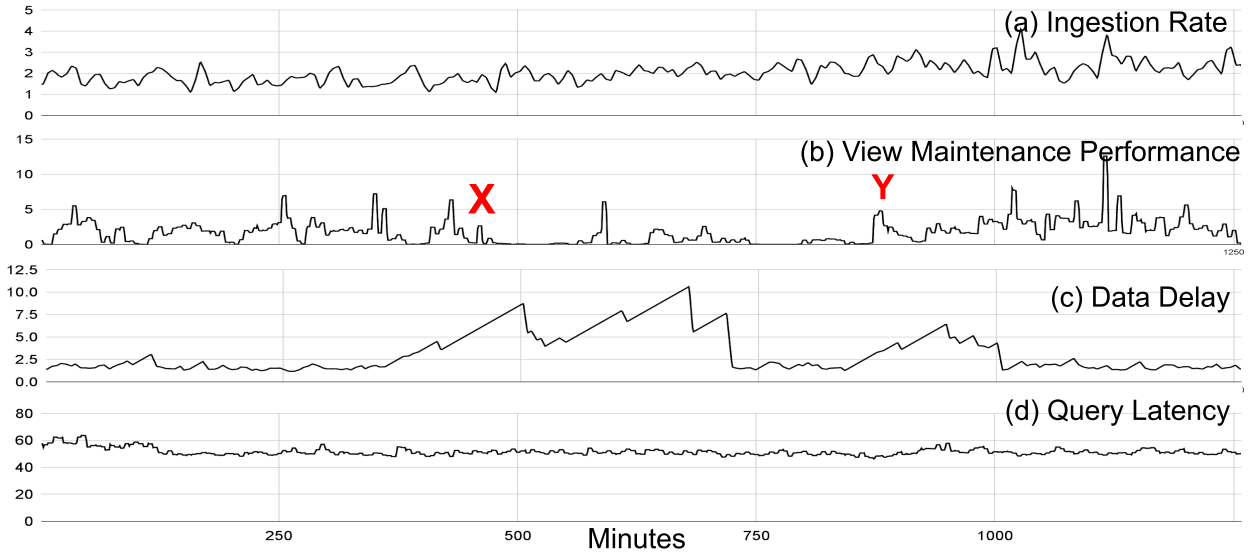

Figure 9: Napa's decoupling means view maintenance backlog does not impact query performance.

[Client A: Tradeoff freshness] is an internal experimentation and analysis application that wants moderate query performance and low costs, but can tolerate lower freshness. It has tuned Napa to have a moderate number of views and achieves reasonably good query performance. Given that it has the highest ingestion rate, it trades off data freshness to keep the costs moderate.

[Client B: Tradeoff query performance] is an application that cares the most about low costs but can tolerate lower query performance. This client uses Napa in a way that less effort is spent on optimizing data for client queries. As a result this client has the lowest resource costs (Figure 10(d)) even though it has a higher ingestion rate than Client C.

[Client C: Tradeoff costs] is a critical external user-facing application that has high freshness and high query performance requirements, and is willing to pay high costs to achieve them. This client has tuned Napa to have a large number of views and uses frequent compactions to keep the number of deltas low. As a result, this client has better data freshness (Figure 10(b)) and query performance (compared to Client A and B), but has to pay the increased resource costs even though its ingestion rate is lower than the other two clients (Figure 10(c)).

10 RELATED WORK

There are numerous commercial offerings for analytical data management, e.g., from Teradata, Oracle, IBM, Microsoft, Snowflake, Amazon, and many other companies. Technology advances on these commercial systems include columnar storage, query optimization, as well as multiple designs for write- and update-optimized indexing.

Cloud Offerings Traditional data analytics systems have evolved from tight coupling of storage and compute to the disaggregated model of decoupled storage and compute to take advantage of the cloud computing paradigm. Amazon Aurora [30] decouples the transaction and query processing layer from the storage layer. Redshift [18] has support for materialized views that are not continuously maintained. Snowflake [8] projects itself as a "virtual data warehouse," which can be spun up on demand and then brought down, and it provides modern database innovations such as columnar storage, vectorization and matching changes in query optimization. Napa provides continuous ingestion and high performance querying with tunable freshness. It further advances the idea of disaggregation by decoupling its architectural components: ingestion, aggregation (i.e., derivation of updates in materialized views), indexing and querying. As a result, the impact of a slowdown in indexing on the query performance is minimized by either trading off data freshness or incurring higher costs as shown in Section 9. This improvement is achieved by bringing in the notion of freshness as another tunable parameter to give Napa this unique capability.

Data analytics within Google An early attempt at building a large scale data management system at Google was Tenzing [6], which offered SQL over Map-Reduce [10] on data stored in Colossus [12, 14] and Bigtable [4]. Dremel [24] is a scan-based querying system that enabled large-scale querying of Google's log data using thousands of machines. Procella [5] is a recent scan-based system that improves upon Dremel by using advanced storage format to support filter expressions, zone maps, bitmaps, bloom filters, and partitions and sorting by keys. In contrast to these prior systems, Napa is a fully indexed system that is optimized for key lookups, range scans, and efficient incremental maintenance of indexes on tables and views. Napa can easily support both adhoc queries and highly selective and less diverse queries. Early systems used sharded MySQL, which over time forked into two systems. The data layer forked off into Mesa [19, 20], while the query layer became F1 Query [27, 29].

Comparison with Mesa Napa is designed to be a drop-in replacement to Mesa and hence is a multi-tenant, multi-homed, distributed, globally-replicated data management system, which is used by mission-critical applications within Google. Napa advances the state of the art in terms of configuration of databases to meet the

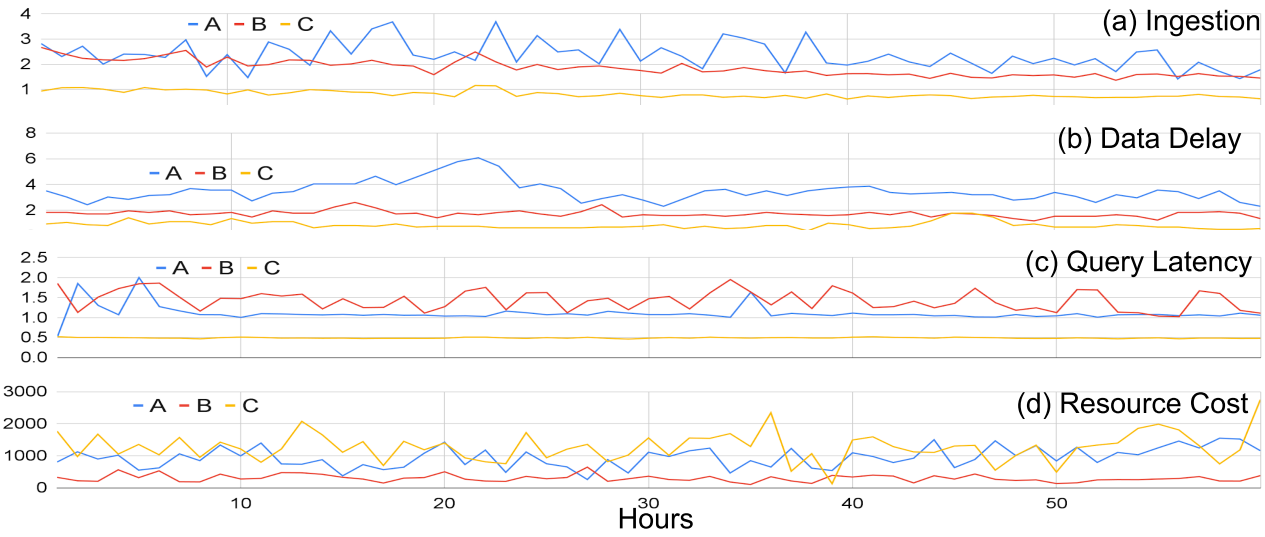

Figure 10: Production metrics from three client workloads (a) Ingestion load, (b) Data Delay, (c) Query Latency, (d) Resource costs per ingested unit.

end client's freshness, cost and query performance tradeoffs. Napa has superior consistency semantics in that it provides a single database-level queryable timestamp, which means that a user can reference and query multiple tables and views in a consistent manner. Napa also supports views with full SQL generality; a subset of them can be continuously maintained. Mesa used a custom framework for creating and maintaining materialized views and their indexes. In contrast, Napa uses F1 Query, an SQL engine, both for processing user queries and for maintaining tables, materialized views, and indexes. Napa has made significant improvements over Mesa in terms of query latency and cost of running the system.

LSM-based Indexing Systems B-trees [3, 16] are the principal index structures in many traditional database management systems. Napa uses a variant of B+-trees that exploits the fact that Napa tables have multi-part keys. Additionally, min/max keys (per-column min/max values) are stored along with each non-leaf block to enable effective pruning. Log-structured merge-trees (LSM) [25] adapt B-tree indexes for high update rates. Napa belongs to a class of LSM systems that trade high write throughput for fast reads. Writes are written to level files which get compacted to form larger level files. Reads merge these at run-time. The efficiency of the LSM data structure is measured by "write amplification" or the number of times an input row is written to disk (across all levels). Since merging is a sequential process, we also use more sequential I/O as opposed to issuing random I/Os. RocksDB [11] and PebblesDB [26], based on LevelDB [13], are examples of write-intensive key-value stores that use the LSM scheme.

11 CONCLUSION

Napa is an analytical data management system that serves critical Google dashboards, applications, and internal users. In some ways, it is comparable to other relevant systems: high scalability, and availability through replication and failover, high user query load, and large data volumes. In other ways, it has a combination of characteristics that is perhaps unique: relying on materialized views to ensure that most queries are sub-second look-ups, maintaining views while ingesting trillions of rows, and giving clients the flexibility to tune the system for data freshness, query latency, and costs.

Whereas current data management storage systems typically rely on scans sped up by columnar storage, parallelism, and compression, Napa relies heavily on views to guarantee robust query performance. Views are optimized for continuous high-bandwidth insertions using log-structured merge-forests. Napa provides clients with flexible configurable parameters for query performance, freshness, and costs. The Queryable Timestamp (QT) provides a live indicator of the client's database production performance against the said requirements. Napa's configurability means that the same system under different configurations can serve cost-conscious clients as well as demands around high performance and freshness.

While Napa serves Google well, it continues to evolve in dimensions related to automatically suggesting views, making the tuning efforts self-driven, and supporting emerging applications. New challenges continue to emerge as new applications and users are frequently added to the Napa ecosystem.

ACKNOWLEDGMENTS

Napa is the result of the efforts of many people, especially our former and current team members. We thank the F1 Query and EngProd teams, as well as the following for their immense contributions to Napa: Alejandro Estrella Balderrama, Pablo Boserman, Dennis Frostlander, Afief Halumi, Taihua He, Mingsheng Hong, Rohit Khare, Sugeeti Kochhar, Ioannis Koltsidas, Lina Kulakova, Romit Kusarye, Andrew Lamb, Ruocheng Li, Sandeep Mariserla, Jeff Shute, Zhaozhe Song, Dao Tao, Dexin Wang, Chad Whipkey, Adam Xu, Yin Ye, Lingyi You, David Zhou, Ed Zhou and Min Zhou. We are also grateful to Ashish Gupta and Shiv Venkataraman for providing strategic vision and guidance to the Napa team.

REFERENCES

[1] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. 2009. Asynchronous view maintenance for VLSD databases. In *SIGMOD*. Providence, RI, 179–192.

[2] A. Ailamaki, D. J. DeWitt, and M. D. Hill. 2002. Data page layouts for relational databases on deep memory hierarchies. *VLDBJ* 11, 3 (2002), 198–215.

[3] R. Bayer and E. M. Mccreight. 1972. Organization and Maintenance of Large Ordered Indexes. In *Software Pioneers*. Vol. 1. Springer-Verlag, Berlin, Heidelberg, 173––189.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4:1–4:26.

[5] B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. McCormick, A. Mokashi, P. Harvey, H. Gonzalez, D. Lomax, S. Mittal, R. Ebenstein, N. Mikhaylin, H.-C. Lee, X. Zhao, T. Xu, L. Perez, F. Shahmohammadi, T. Bui, N. Mckay, S. Aya, V. Lychagina, and B. Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *PVLDB* 12, 12 (2019), 2022–2034.

[6] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. 2011. Tenzing: A SQL Implementation On The MapReduce Framework. *PVLDB* 4, 12 (2011), 1318–1327.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. 2013. Spanner: Googleś Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8:1–8:22.

[8] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. San Francisco, CA, 215–226.

[9] J. Dean and L. A. Barroso. 2013. The tail at scale. *CACM* 56, 2 (2013), 74–80.

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

[11] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. 2017. Optimizing Space Amplification in RocksDB. In *CIDR*. Chaminade, CA, 9.

[12] A. Fikes. 2010. Storage Architecture and Challenges. https://cloud.google.com/files/storage_architecture_and_challenges.pdf.

[13] S. Ghemawat and J. Dean. 2011. LevelDB. https://github.com/google/leveldb/.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. 2003. The Google file system. In *SOSP*. Bolton Landing, NY, 29–43.

[15] Goetz Graefe. 2006. Implementing Sorting in Database Systems. *ACM Comput. Surv.* 38, 3 (Sept. 2006), 10–es.

[16] G. Graefe. 2011. Modern B-Tree Techniques. *Foundational Trends in Databases* 3, 4 (April 2011), 203–402.

[17] G. Graefe and H. A. Kuno. 2010. Adaptive indexing for relational keys. In *ICDE Workshop*. Long Beach, CA, 69–74.

[18] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD*. Melbourne, Victoria, Australia, 1917–1923.

[19] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. 2014. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. *PVLDB* 7, 12 (2014), 1259–1270.

[20] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Govind Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. 2016. Mesa: a geo-replicated online data warehouse for Google's advertising system. *CACM* 59, 7 (2016), 117–125.

[21] S. Idreos, M. L. Kersten, and S. Manegold. 2007. Database Cracking. In *CIDR*. Asilomar, CA, 68–78.

[22] B. R. Iyer. 2005. Hardware assisted sorting in IBM's DB2 DBMS. In *COMOD*. Goa, India, 9.

[23] D. E. Knuth. 1998. *The Art of Computer Programming, Volume III: Sorting and Searching, 2nd edition*. Addison-Wesley-Longman, Boston, MA.

[24] S. Melnik, A. Gubarev, J. Jing Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. 2011. Dremel: interactive analysis of web-scale datasets. *CACM* 54, 6 (2011), 114–123.

[25] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. OŃeil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[26] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *SOSP*. Shanghai, China, 497–514.

[27] B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte, F. Weigel, D. Wilhite, J. Yang, J. Xu, J. Li, Z. Yuan, C. Chasseur, Q. Zeng, I. Rae, A. Biyani, A. Harn, Y. Xia, A. Gubichev, A. El-Helw, O. Erling, Z. Yan, M. Yang, Y. Wei, T. Do, C. Zheng, G. Graefe, S. Sardashti, A. M. Aly, D. Agrawal, A. Gupta, and S. Venkataraman. 2018. F1 Query: Declarative Querying at Scale. *PVLDB* 11, 12 (2018), 1835–1848.

[28] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*. Tucson, AZ, 23–34.

[29] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. 2013. F1: A Distributed SQL Database That Scales. *PVLDB* 6, 11 (2013), 1068–1079.

[30] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. ACM, Chicago, IL, 1041–1052.