# F1 Lightning: HTAP as a Service

Jiacheng Yang Ian Rae Jun Xu Jeff Shute Zhan Yuan Kelvin Lau
Qiang Zeng Xi Zhao Jun Ma Ziyang Chen Yuan Gao Qilin Dong
Junxiong Zhou Jeremy Wood Goetz Graefe Jeff Naughton John Cieslewicz
Google LLC
f1-lightning-paper@google.com

## ABSTRACT

The ongoing and increasing interest in HTAP (Hybrid Transactional and Analytical Processing) systems documents the intense interest from data owners in simultaneously running transactional and analytical workloads over the same data set. Much of the reported work on HTAP has arisen in the context of "greenfield" systems, answering the question "if we could design a system for HTAP from scratch, what would it look like?" While there is great merit in such an approach, and a lot of valuable technology has been developed with it, we found ourselves facing a different challenge: one in which there is a great deal of transactional data already existing in several transactional systems, heavily queried by an existing federated engine that does not "own" the transactional systems, supporting both new and legacy applications that demand transparent fast queries and transactions from this combination. This paper reports on our design and experiences with F1 Lightning, a system we built and deployed to meet this challenge. We describe our design decisions, some details of our implementation, and our experience with the system in production for some of Google's most demanding applications.

**PVLDB Reference Format:**
J. Yang, I. Rae, J. Xu, J. Shute, Z. Yuan, K. Lau, Q. Zeng, X. Zhao, J. Ma, Z. Chen, Y. Gao, Q. Dong, J. Zhou, J. Wood, G. Graefe, J. Naughton, J. Cieslewicz. F1 Lightning: HTAP as a Service. *PVLDB*, 13(12): 3313-3325, 2020.
DOI: http://doi.org/10.14778/3415478.3415553

## 1. INTRODUCTION

The intense research and commercial interest in HTAP (Hybrid Transactional and Analytical Processing) systems demonstrates that data owners have a strong desire to process queries and transactions over the same data set. There has been a great deal of research and development relevant to HTAP systems (some of it starting long before the term "HTAP" was coined), and the literature is full of techniques and system descriptions. Much of that work assumes a "greenfield" approach to the problem, where the question is: what should an ideal HTAP system look like, and what technical advances are needed for good performance. In this paper we consider a related but different approach: a loosely coupled HTAP architecture that can support HTAP workloads under a variety of constraints.


*Proceedings of the VLDB Endowment,* Vol. 13, No. 12
ISSN 2150-8097.
DOI: http://doi.org/10.14778/3415478.3415553

Simply put, while supporting HTAP well is of critical importance, for us a greenfield approach was not the best option to enable HTAP processing in Google's ecosystem. In Google, we use multiple transactional data stores that serve large legacy and new workloads, and we have federated query engines that are loosely coupled with these systems. We want a single HTAP solution that can be enabled across the different options for transactional storage to avoid costly migrations and to permit flexibility in the design of transactional storage systems, and we want to benefit from separation of concerns by allowing transactional systems to focus on transaction processing and query engines to focus on query processing, with an emphasis on analytical queries.

Accordingly, we have designed, implemented, and deployed Lightning, a loosely coupled HTAP solution that we term "HTAP-as-a-service." By "HTAP-as-a-service" we mean that Lightning can transparently provide HTAP functionality accessible to applications merely by marking some tables in their schema in the transactional store as "Lightning tables." (These are the tables over which they expect to run HTAP workloads.) The actual logistics of running and supporting Lightning in production is handled by the HTAP service provider, rather than by the applications accessing Lightning or the transactional system provider.

All the work of creating a read-optimized copy of the data—keeping it consistent and fresh with respect to the transactional data, managing controlled replication, and optimizing and executing queries that may span transactional and Lightning tables—is handled by Lightning and its integration with a federated query engine. Users of the query engine benefit from improved efficiency and performance, many of them without even knowing Lightning exists. Furthermore, we note that this is also transparent to the transactional store—we did not need to modify the transactional store to provide the Lightning service, and in fact the Lightning service is developed and maintained by teams that do not own the development of the transactional storage systems.

It is our hope that the challenges we faced and the decisions we made are of interest and use to others. We give an overview of the system in section 3 and later highlight techniques we found useful, including a vectorized columnar implementation of merging and compaction (section 4.5); the use of a two-level schema to provide a great deal of freedom in how Lightning transparently speeds queries over its data (section 4.6); a change data capture component we call "changepump" that can be extended to new transactional data stores (section 4.8); methods for reliability and efficiency in a multi-data-center environment (section 4.9); and tight integration and co-design with a federated query engine (section 5). Finally, we also share some of the engineering practices we developed (section 6) and demonstrate that Lightning is successfully meeting its performance goals under real-world conditions at scale (section 7).

## 2. RELATED WORK

Many kinds of architectures and systems have proliferated across the HTAP landscape. In the HTAP survey [25], HTAP solutions are divided into "Single System for OLTP and OLAP" and "Separate OLTP and OLAP Systems." Many greenfield systems fall under "Single System for OLTP and OLAP." Among them, systems that use hybrid row-wise and columnar data organizations tend to have better performance than those using a single data organization for both ingestion and analytics. F1 Lightning is different from the single-system approach in that the analytic system is decoupled from the OLTP system because our users at Google cannot easily do a wholesale migration to an entirely new system.

The survey further categorizes HTAP solutions using "Separate OLTP and OLAP Systems" into two sub-categories: shared storage or decoupled storage for OLTP and OLAP. Solutions that adopt shared storage usually require modifications to the OLTP system—in fact, systems in this category are usually developed by the OLTP systems themselves to leverage an existing analytical query engine to accelerate analytic queries. F1 Lightning, as well as others that use decoupled storage, assumes no modifications can be made to the OLTP storage.

Many applications set up their HTAP architecture using loosely decoupled storage by maintaining a separate, offline ETL process. Using offline ETL to ingest data into columnar file formats tends to suffer from high lag between the OLTP data and the OLAP copy. F1 Lightning improves the data freshness via the integration with a Change Data Capture (CDC) mechanism and use of a hybrid memory-resident and disk-resident storage hierarchy. The abundance of customized solutions in this space also result in duplication due to the lack of standards. F1 Lightning provides a standard solution at Google via a *managed service*.

Next, we will introduce some specific systems and implementations with which F1 Lightning shares similar design principles and technologies.

SAP HANA Asynchronous Parallel Table Replication (ATR) [21] is a replication architecture that handles all OTLP workloads in a single modern server machine with a row-format store and uses a parallel log replay scheme to maintain scale-out OLAP replicas stored in column formats. Queries can be routed to the OLAP replicas according to users' preferred max acceptable staleness. There are differences between their scenario and our production environment, and these different constraints lead to different design choices. At a very high level, SAP HANA ATR exploits tighter coupling between OLTP and OLAP processing, whereas our system intentionally supports looser coupling, yielding two systems that are both of interest.

In more detail, firstly, the SAP architecture requires an interface for log shipping inside the OLTP engine, which in general cannot be achieved without modifying the OLTP system. Secondly, the OLTP database in SAP HANA ATR runs on a single machine, whereas OLTP databases at Google are geo-replicated distributed transactional systems. This means Lightning's change replication has to process a distributed change log and coordinate parallel and distributed log replication. Thirdly, SAP HANA ATR uses its own query system for query routing, whereas Lightning uses a federated query engine. Last but not least, Lightning's HTAP-as-a-service can be easily and transparently connect (through the federated query engine) applications to multiple OLTP systems, allowing users to achieve good HTAP performance without migrating their data out of their preferred OTLP system.

The tight coupling between OLTP and OLAP processing in the SAP HANA ATR approach helps achieve lower data delay than a loosely coupled design. An all-new system choosing its HTAP storage from the beginning would probably benefit from a more tightly coupled HTAP system like SAP HANA ATR.

TiFlash [6] is a columnar extension for TiDB [5]. It adds a columnar storage and vectorized processing layer to the row-store TiDb. It is tightly integrated with the TiDB query layer. TiFlash only keeps casual consistency with TiDB. This is different from Lightning which can provide strong snapshot consistency with the source database in the queryable window.

Oracle Database In-Memory [3] is another example of "Single System for OLTP and OLAP." Oracle DBIM accelerates HTAP workloads by maintaining an in-memory column store for active data that is kept transactionally consistent with the persisted row store. Using Oracle DBIM requires no changes to applications.

*Middleware-based database replication* systems [12,13,19,27,30] decouple the replication engine from underlying DBMS systems to perform ETL to extract data from heterogeneous databases. Existing work on middleware-based database replication mainly focuses on problems of consistency, isolation and the performance of the ETL process. These replication systems are not optimized for generating replicas for fast analytics, and they do not try to provide a transparent HTAP experience.

Several works have been done to use *Change Data Capture* (CDC) to incrementally extract changes from source databases into a separate storage for analytics. These systems typically have improved change propagation delay and change replication efficiency over the traditional ETL approach that reimports the entire dataset. LinkedIn Databus [17] is a source-agnostic distributed change data capture system that feeds changes from source-of-truth systems to downstream applications to build specialized storage and indexes for different purposes. Databus is similar in spirit to Lightning's Changepump. Unlike Lightning, Databus does not provide a complete service solution for hybrid transactional and analytic processing. Instead, it is a component for building data replicas. We think Databus could be a useful component in building a complete HTAP solution.

Regarding HTAP solutions that are tightly coupled with a SQL query engine, Wildfire [11] and SnappyData [24] are two more recent HTAP systems that are designed to use the Spark computational engine. Wildfire combines SparkSQL [9] with the Wildfire storage system that uses columnar data organization built on Parquet [4]. It adds the Spark API for OLTP and extends SparkSQL for OLAP queries. The query runs on Spark executor, with operations pushed down into Wildfire storage servers. SnappyData also uses the Spark ecosystem to provide a unified interface for transactional, interactive analysis and streaming processing. It uses a mix of Spark RDD [32] and a transactional store (Apache Gemfire [2]) as its storage layer. These two systems have storage systems that are optimized for their computational engine. Unlike Lightning, they are still native HTAP systems that would require users to migrate data from their existing transactional storage.

## 3. SYSTEM OVERVIEW

Our HTAP solution is composed of three main parts: an OLTP database that acts as the source of truth and exposes a change data capture or other change replay interface; *F1 Query* [28], a distributed SQL query engine; and *Lightning*, which maintains and serves read-optimized replicas.

At Google, there are two primary OLTP databases: *F1 DB* [29] and *Spanner* [16]. F1 DB was implemented as a relational database layer on top of Spanner, and it is used by several major Google products including AdWords, Payments, and Shopping. Many applications at Google also directly use Spanner.

F1 Query is a federated query engine that executes queries written in a dialect of SQL internally called GoogleSQL (open sourced as

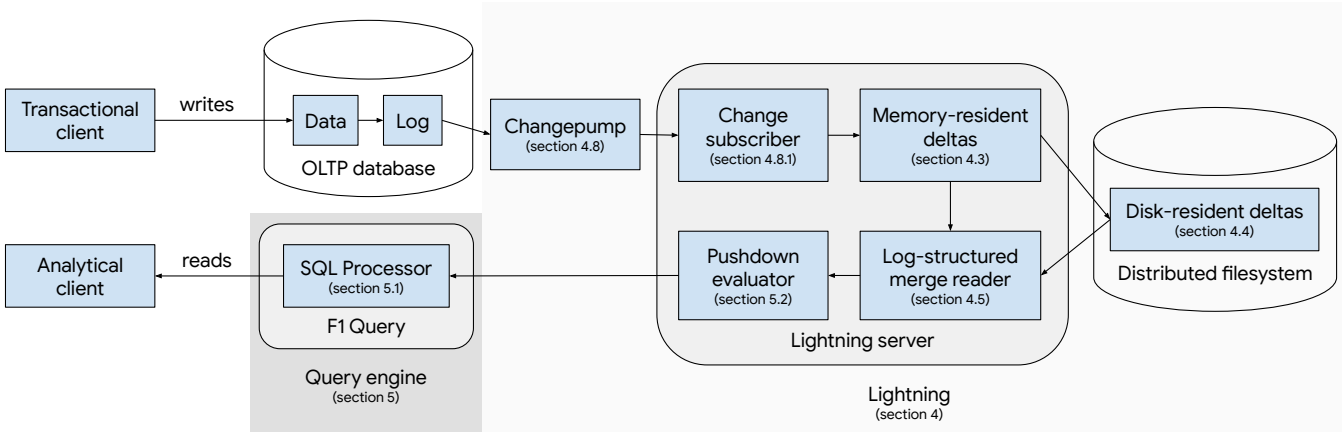


**Figure 1:** An overview of how data flows through an HTAP system with Lightning. Details on Changepump and the Lightning server architecture are described in section 4, and details on Lightning's integration with F1 Query are described in section 5.

ZetaSQL [7]). In contrast to the query layer native to the OLTP system (e.g., Spanner SQL [10]), F1 Query is a federated engine that supports many different internal data sources, including F1 DB, Spanner, Mesa [20], ColumnIO [23], and BigTable [14]. Users are able to write queries that seamlessly join across these systems, and F1 Query executes in excess of 100 billion queries per day on behalf of users and applications.

There is a natural tension between optimizing for analytic queries and optimizing for transaction processing. The Ads F1 DB, for example, stores tables in a hierarchical schema with entities interleaved in physical storage in order to avoid distributed transactions [29]. This hierarchical schema imposes some constraints on partitioning. The database keeps a modest number of partitions for the root table in order to keep the write latency low, but large child tables would benefit from more partitions for high-throughput reads. Small tables, on the other hand, may have too many partitions for low-latency reads because they inherit the same partitioning from the root table.

In general, F1 DB and Spanner optimize for OLTP workloads like writes and point-lookup queries by using efficient row-oriented storage and indexes, and users design their schemas to maximize write throughput. Unfortunately, this means that, while F1 Query is able to run analytical queries over these datasets, analytical query performance directly on this data is often suboptimal. F1 Query counteracts this by running distributed analysis queries with many workers, but this incurs substantial computational resource costs in order to provide reasonable latency (see Figure 2 and Figure 1 for the cost and latency comparison).

In response to these issues, some teams set up pipelines for copying F1 DB tables into ColumnIO files or other formats for further analysis. This approach has several drawbacks. First, it results in fragmented engineering investment as well as duplicate storage (in fact, in some cases, multiple teams each keep their own copies of the same data, so the replication factor can be much greater than two). Second, since ColumnIO files do not support in-place updates, copies have to be periodically restated as a whole, which is very inefficient, and datasets maintained this way also tend to have poor data freshness as new changes are only visible after the next restatement. Third, users need to explicitly change their queries to read from the copy, and they may need to modify their queries due to schema interoperability issues or other semantic differences between the F1 DB schema and the ColumnIO schema. Finally, access permissions need to be kept in sync between the original database and extracted data, adding maintenance overhead and the risk of security holes.

To address these issues, we developed Lightning, an HTAP system that replicates data from an OLTP database into a format optimized for analytical queries. Database owners are able to enable Lightning on a table-by-table basis or enable Lightning for the entire database. For each enabled table, *Changepump*, a component of Lightning, uses the change data capture mechanism exposed by F1 DB (or, in the case of Spanner, an internal log shipping interface) to detect new changes. Changepump then forwards those changes to partitions managed by individual *Lightning servers*, each of which maintain Log-Structured Merge (LSM) trees [26] backed by a distributed file system (see Figure 1 for an overview). When Lightning ingests these changes, it transforms the change data from the row-oriented, write-optimized format used by the OLTP database into a column-oriented format optimized for analysis. Lastly, Lightning also supports asynchronous maintenance of additional structures like secondary indexes and rollups. This further improves query performance without impacting transaction throughput.

Lightning serves ingested data in a manner that is snapshot consistent with respect to the original OLTP database. Both F1 DB and Spanner support multi-version concurrency control using timestamps, and every change committed to Lightning retains its original commit timestamp. Lightning guarantees that reads at a specific timestamp will produce results identical to reads against the OLTP database at the same timestamp, and all changes that it ingests will be represented with full fidelity and semantics equivalent to the source database. F1 Query takes advantage of this to transparently accelerate query performance by automatically rewriting eligible queries to use Lightning; that is, when a query requests to read F1 DB and Spanner at a particular timestamp, if that data is available in Lightning, the query will instead read from Lightning and benefit from improved performance without any explicit action on the part of the end user. This rewrite occurs on a table-by-table basis, meaning it's possible for a single query to read some tables from Lightning and other tables directly from the OLTP database.

Adding Lightning to our query ecosystem achieves the following:

- **Improved resource efficiency and latency for analytic queries:** Lightning stores data optimized for read-only analysis queries rather than transactional processing.
- **Simple configuration and deduplication:** Instead of engineers developing ad-hoc solutions with bespoke ETL pipelines, Lightning standardizes the process and allows database owners to enable HTAP with a simple configuration change.

- **Transparent user experience:** Users get faster analytic queries without changing their SQL text or even being aware of the HTAP replica.

- **Data consistency and data freshness:** Analytic queries run over a snapshot consistent with a recent version in the OLTP database. Changes are automatically replicated from the OLTP database with low delay.

- **Data security:** F1 Query authorizes queries based on the access permissions of the original OLTP database before rewriting queries to use Lightning. This ensures that only users authorized to read the original data will be able to use the read-optimized replica. Access control features commonly used in the OTLP database like logical views work the same way with Lightning.

- **Separation of concerns:** Lightning is an independent system that is not maintained by either the F1 DB or Spanner teams. This allows the teams managing those systems to focus on supporting efficient transactional updates, while the Lightning team can focus on optimal analytic query performance.

- **Extensibility:** Lightning can be extended to support new transactional databases with little effort. The original implementation of Lightning supported only F1 DB. Subsequent development efforts have extended it to support Spanner. Even though F1 DB and Spanner have the same underlying storage, they have different schemas, different client APIs, and different change data capture mechanisms. In principle, Lightning can be extended to operate on any OLTP database that provides a change data capture mechanism.

In the following sections, we describe the architecture of Lightning and how it interacts with F1 Query in more detail.

# 4. LIGHTNING ARCHITECTURE

Lightning consists of the following components:

- **Data storage:** The data storage layer is responsible for continuously applying changes to the Lightning replica. It creates read-optimized files stored in a distributed file system, provides an API that allows query engines to read stored data with semantics identical to the OLTP database, and handles background maintenance operations like data compaction.

- **Change replication:** A change replication system is responsible for tailing a transaction log provided by the OLTP database and partitioning changes for distribution to relevant data storage servers. The change replication system is responsible for tracking which changes have been applied, replaying historical changes as needed, and triggering backfills when new tables are added.

- **Metadata database:** State for the data storage and change replication components is stored in a metadata database.

- **Lightning masters:** Lightning masters coordinate actions across the other servers and maintain Lightning-wide state.

In the following sections, we describe the read semantics that Lightning provides, then go into detail on how these components are designed. In the interest of space, we focus primarily on the data storage and change replication components.

## 4.1 Read semantics

Lightning supports multi-version concurrency control with snapshot isolation. All queries against Lightning-resident tables specify a read timestamp, and Lightning returns data consistent with the OLTP database as of that timestamp. This was primarily motivated by existing OLTP databases at Google, which use this model, and the consequent expectations of application developers, who have built a large ecosystem of products assuming this model holds.

Since Lightning applies the OLTP database's change log asynchronously, there is a delay before a change made in the OLTP database is visible to queries over Lightning. Additionally, Lightning supports an upper bound on how far in the past an individual query can read. This upper bound could be infinite (i.e., Lightning could store all changes), but in practice most queries are focused on recent data, and limiting the amount of data in Lightning saves cost.

We refer to timestamps that can be queried through Lightning as *safe timestamps*. The *maximum safe timestamp* indicates that Lightning has all ingested all changes up to that timestamp, and the *minimum safe timestamp* indicates the timestamp of the oldest version that can be queried. In essence, Lightning has a single-version snapshot of the database as of the minimum safe timestamp and a multi-version record from that point up to the maximum safe timestamp. We call the range of safe timestamps that Lightning maintains the *queryable window*. In production at Google, the queryable window is typically ten hours.

## 4.2 Tables and deltas

Lightning stores data organized into *Lightning tables*. Database tables, indexes, and views are all treated as physical tables in Lightning. Each Lightning table is divided into a set of partitions using range partitioning. Each partition is stored in a multi-component Log-Structured Merge (LSM) tree. We refer to each component in the LSM tree as a *delta*.

Deltas contain *partial row versions* for their corresponding Lightning table. Each partial row version is identified by the primary key of the corresponding row and a timestamp of when that version was committed in the OLTP database. There are three types of versions that Lightning stores, corresponding to the mutation that was made to the source data:

- **Inserts:** Inserts contain values for all columns. The first version of each row is an insert.

- **Updates:** Updates contain values for at least one non-key column, and they omit values for columns that were not modified.

- **Deletes:** Deletes do not contain any values for non-key columns. Deletes are used as tombstones to indicate that rows should be removed from reads after a specific timestamp.

In order to simplify delta maintenance, Lightning is intentionally permissive about the content of deltas. Individual deltas may contain multiple versions for the same key, and there may be duplicate versions across different deltas for the same partition. Within a delta, partial row versions are uniquely identified by $\langle \text{key}, \text{timestamp} \rangle$ pairs, and in order to support fast seeks to a specific timestamp, deltas are ordered by key ASC, timestamp DESC.

When a table is bootstrapped, Lightning runs an offline process to generate the partitioning of the table and creates an initial delta for each partition reading and converting from the OTLP source.

## 4.3 Memory-resident deltas

When changes are ingested by Lightning, the resulting partial row versions are first written to a memory-resident, row-wise B-tree. Similar to the write-optimized store used by C-Store [31] or the $C_0$ LSM-tree [26], this allows for high update rates at the cost of some read efficiency.

A memory-resident delta has at most two active writers, with many readers. For each partition, Lightning uses a thread to apply new changes from the OLTP transaction log. Additionally, a background thread periodically runs a garbage collection process to remove versions that are no longer in Lightning's queryable window. Lightning handles these concurrency needs using copy-on-write in the B-tree structure.

Once data is written to a memory-resident delta, it is immediately available for querying, subject to the consistency protocol provided by Changepump. However, writes to memory-resident deltas are not durable, and Lightning does not maintain its own write-ahead log. In the case of system failure, changes stored in memory may be lost. Lightning recovers from this state by replaying from the log of the OLTP database.

It can often be impractical to replay many hours of changes from the transaction log. To reduce the amount of data that must be replayed, Lightning periodically checkpoints memory-resident deltas to disk. Because we want the checkpointing operation to be fast and cheap, Lightning writes the content of the memory-resident B-tree to disk as-is without transformations. Checkpoints are not directly queryable and must be loaded into memory to be read.

When deltas become too large, either due to a per-delta size limit or to server-wide memory pressure, Lightning writes them to disk. Unlike checkpointing, this operation includes a transpose where Lightning converts the row-wise memory-resident data into a read-optimized columnar format, analogous to the tuple mover in the C-Store architecture. Existing reads will continue to be served from the memory-resident delta until the write is complete, at which point they will transparently switch to reading from disk.

## 4.4 Disk-resident deltas

Disk-resident deltas, which contain the bulk of Lightning's data, are stored in read-optimized columnar files. We built an abstraction layer with a common interface that allows Lightning to use many different file formats to store deltas. This allows us to experiment with new formats when they become available, and it ensures that we can transition to better, more efficient file formats without a substantial engineering effort. Lightning supports several internal columnar file formats, but here we will introduce just one columnar file format currently used in production.

Each delta file stores two parts: a data part and an index part. The data part stores row versions in a PAX (Partition Attributes Across) [8] layout where rows are first divided into row bundles then stored column-wise within a row bundle. The index part contains a sparse B-tree index on the primary key where the leaf entry tracks the key range of each row bundle. The index is much smaller than the data and is usually cached in Lightning servers.

This layout obtains a good balance between range scan performance and point lookup performance. Since we use Lightning to serve hybrid workloads, it is important to optimize performance for all traffic patterns.

## 4.5 Delta merging

Lightning reads partitions at a particular timestamp requested by the query. However, since Lightning stores partial row versions that may be scattered across several deltas (either memory- or disk-resident), each read must merge deltas and combine row versions in order to form complete rows.

Delta merging consists of two logical operations: *merging* and *collapsing*. Merging deduplicates changes in the source deltas and copies distinct versions to the new delta. Because the source deltas and the new delta do not necessarily use a common schema, merging performs schema coercion while copying rows. Collapsing combines multiple versions of the same key into a single version.

This process uses a vectorized version of standard LSM logic that combines a merge and aggregate operator. As a preprocessing step, Lightning first enumerates the deltas that need to participate in the merge; if there are predicates on the primary key, then deltas without versions matching those predicates may be omitted. Once the deltas that must participate in the merge are identified, Lightning performs a $k$-way merge in two stages that are applied repeatedly: *merge plan generation* and *merge plan application*.

In merge plan generation, Lightning reads a block of keys from each of the $k$ inputs and identifies which versions to collapse and in which order in a structure we call the *merge plan*. The first step is to identify the range of keys that can be collapsed in this round. Since multiple row versions for a single primary can be contained within a single delta, Lightning must take care that it collapses only complete histories with no holes.

To illustrate this more concretely, suppose Lightning is merging two inputs, $D_1$ and $D_2$, and it has read a block from each input (we use $K_i$ to represent an arbitrary key where $K_i < K_j$ for all $i < j$, abbreviate "timestamp" as $\mathrm{ts}$ and "operation" as $\mathrm{op}$):

| | |
|---|---|
| | $\langle K_1, \mathrm{ts}: 75, \mathrm{op}: \mathrm{UPDATE}\rangle$ |
| $\langle K_1, \mathrm{ts}: 100, \mathrm{op}: \mathrm{UPDATE}\rangle$ | $\langle K_1, \mathrm{ts}: 25, \mathrm{op}: \mathrm{INSERT}\rangle$ |
| $\langle K_1, \mathrm{ts}: 50, \mathrm{op}: \mathrm{UPDATE}\rangle$ | $\langle K_2, \mathrm{ts}: 150, \mathrm{op}: \mathrm{UPDATE}\rangle$ |
| $\langle K_2, \mathrm{ts}: 125, \mathrm{op}: \mathrm{UPDATE}\rangle$ | $\langle K_2, \mathrm{ts}: 100, \mathrm{op}: \mathrm{UPDATE}\rangle$ |
| *Delta* $D_1$ | *Delta* $D_2$ |

Lightning can only collapse versions whose key and timestamp are less than than $\langle K_2, \mathrm{ts}: 125\rangle$, and only collapsed rows whose key is less than $K_2$ can be included in the output of this round. This is because $D_1$ may have additional versions for $K_2$ whose timestamp is between 100 and 125, but this will not be determined until the next batch is read from $D_1$. Therefore, the upper bound on the range of versions that may be collapsed in a single round is the minimum of the maximum keys across all blocks being considered.

Once the upper bound is identified, Lightning compares key values to generate a single sorted stream. Each unique key is assigned a slot in the output buffer. Multiple versions for the same key are identified as *collapsing groups* and assigned the same slot. Finally, versions for the boundary key are assigned a slot in an *escrow buffer*, where they will be collapsed and retained for the next round.

After generating the merge plan, Lightning applies it column-by-column, copying and aggregating row values into the appropriate buffers. Lightning then flushes the output buffer, and uses the escrow buffer as an additional input in the next round. That is, a two-way merge will have a third input containing a single row version for all rounds except the first, but the logic is otherwise unchanged.

Finally, this process repeats with additional rounds until all inputs are exhausted.

## 4.6 Schema management

Since Lightning is replicating an OLTP database and transparently serving queries, it must handle schema evolution with identical semantics. Lightning monitors changes to the source database schema (described later in section 4.8.3) and automatically applies changes with minimal data movement and processing.

| Id (INT) | TS (INT) | OP (ENUM) | Name (STRING) | Address (STRUCT) |
|---|---|---|---|---|
| 1 | 150 | UPDATE | *NotSet* | {city: "Madison" state: "WI"} |
| 1 | 125 | UPDATE | *NotSet* | {city: "Milwaukee" state: "WI"} |
| 1 | 100 | INSERT | John Smith | {city: "Seattle" state: "WA"} |
| 2 | 50 | INSERT | Jane Doe | {city: "San Jose" state: "CA"} |

**(a)** Partial row versions conforming to a logical schema.

| Logical column | Physical column |
|---|---|
| Id | Id |
| TS | TS |
| OP | OP |
| Address | Address |
| Address.City | City |
| Address.State | State |

**(b)** A mapping between logical and physical columns.

| Id (INT) | TS (INT) | OP (INT) | Name (STRING) | Address (STRING) | City (STRING) | State (STRING) |
|---|---|---|---|---|---|---|
| 1 | 150 | UPDATE | *NotSet* | {city: "Madison" state: "WI"} | Madison | *NotSet* |
| 1 | 125 | UPDATE | *NotSet* | {city: "Milwaukee" state: "WI"} | Milwaukee | WI |
| 1 | 100 | INSERT | John Smith | {city: "Seattle" state: "WA"} | Seattle | WA |
| 2 | 50 | INSERT | Jane Doe | {city: "San Jose" state: "CA"} | San Jose | CA |

**(c)** Partial row versions conforming to a physical schema.

**Figure 2:** An example of how partial row versions with "Id" as the primary key column are stored in Lightning's two-level schema design. In this case, types like structs and enums become strings and integers in the physical schema, respectively, and individual struct fields are stored separately as columns in addition to the full serialized struct. *NotSet* values are markers that indicate a value has not changed since the previous version.

To achieve this, Lightning uses a two-level schema abstraction. The first level is the *logical schema*, which maps from the OLTP schema into a Lightning table schema. The logical schema contains complex types such as protocol buffers and GoogleSQL structs, as well as types that logically map to simpler types, such as dates and enum values that map to integers. For a particular logical schema, Lightning then generates one or more *physical schemas*. The physical schema contains only primitive types, such as integers, floats, and strings. Lightning's file format interface operates only at the level of the physical schema. This means that file format implementers do not need to understand the semantics of complex types, reducing the engineering cost of implementing new file formats.

Logical schemas and physical schemas are connected via a *logical mapping*. The mapping specifies how to transform a logical row to a physical row and vice versa. Data is converted from logical rows to physical rows during ingestion and back again at read time as part of the LSM stack. An example of how data is stored under this design is shown in Figure 2.

One use case for mappings is to implement alternative storage layouts for the same logical data. For example, when storing a protocol buffer, we have two options. Lightning could store it as a serialized byte string, or Lightning could decompose its fields into individual columns. The former is better for queries that read most or all of the fields, and the latter is better for queries that read only a few fields. Lightning can even store both layouts, to spend storage space in order to get optimal read performance for both extremes, and it can use different layouts in memory than it uses on disk.

Mappings also facilitate metadata-only schema changes. There are many common schema changes that Lightning can accommodate without explicitly rewriting data on disk. For example, if a schema change adds a column to a table, there's no need to modify deltas that were written prior to that schema change; Lightning can simply generate default values at read time. Similarly, if a schema change drops a column, Lightning actually cannot remove that data immediately since it still needs to be accessible to queries at a timestamp prior to the drop operation, but this data should not appear in queries executing under the new schema.

Accordingly, whenever a schema change occurs, Lightning creates a new logical schema. Deltas created after the schema change natively write using the new physical schema. For old deltas, Lightning analyzes the differences between the original logical schema and the new logical schema to construct *schema-adapted logical mappings*. Schema-adapted logical mappings specify how Lightning can adapt from physical rows conforming to the old schema into logical rows conforming to the new schema (and vice versa). At read time, the LSM stack applies the schema-adapted mappings to seamlessly convert to the expected schema.

Not all schema changes can be handled using the mapping layer. When a new table is created in the OLTP database, Lightning needs to generate an initial snapshot. Changes like this are performed by background task workers that run in separate processes.

Finally, over time, a table may have many metadata-only schema changes, with many associated transformations. However, applying many transformations imposes a performance overhead on each query. In order to reduce this overhead, Lightning applies schema transformations and moves old data to a newer schema as part of the compaction process described in the next section.

## 4.7 Delta compaction

Lightning is constantly ingesting changes and creating new deltas. However, the continuous build-up of deltas increases resource usage and hurts read performance. In order to keep these costs manageable, Lightning runs periodic *delta compaction* operations to rewrite smaller deltas into a single larger delta.

Lightning runs four types of delta compaction: active compaction, minor compaction, major compaction and base compaction. Active

compaction performs delta compaction on memory-resident deltas. Minor compaction and major compaction runs compaction on multi-version disk-resident deltas. Base compaction generates a new snapshot of data at a timestamp before the minimum queryable timestamp. In the event of a high rate of updates, Lightning servers need to frequently flush data to disk and generate new multi-version deltas to free up memory. In order to keep the total number of deltas small, compaction speed must be able to catch up with delta flushing speed. We designed a size-based delta compaction policy to generate compaction tasks. The key idea is that Lightning runs fast and slow compaction in two different compaction tasks, i.e. minor compaction and major compaction. Minor compaction compacts small and likely fresh deltas, while major compaction handles large and old deltas. When Lightning selects deltas to compact for a compaction task, it uses the delta size as a criterion. Starting with a small size limit, Lightning exponentially increases the limit until it finds two or more deltas eligible for compaction. This allows Lightning to quickly reduce the number of small deltas while avoiding repeatedly rewriting large amounts of data in consecutive compaction tasks.

Of the four compaction tasks, only active compaction, which is cheap and fast to run, is done in Lightning servers. All the other three tasks are scheduled by Lightning servers, but executed on dedicated task workers without need to compete for resources with read operations and other critical work on Lightning servers. Once a compaction task is done, Lightning servers asynchronously load the latest compacted deltas to replace old deltas.

## 4.8 Change replication

We implemented a change-tailing service called *Changepump*. Changepump provides a unified interface across different transactional sources, abstracting the details of individual change data capture interfaces away from the main Lightning data storage layer, and it provides a scalable and efficient way of feeding transactional changes into its clients, namely change subscribers. Changepump provides several benefits for Lightning.

First, it hides the details of individual OLTP databases from the rest of the system. For each supported transactional source, we implement an adapter inside Changepump that converts from that system's change data capture format into a unified format. This makes it relatively straightforward to support new data sources.

Second, it adapts from a transaction-oriented change log into a partition-oriented change log. The change log record of a single transaction may span many different Lightning partitions. Each Lightning partition is managed independently. In order to maintain a partition, Lightning servers just want a series of changes that must be applied to that partition, regardless of the original scope of each transaction. Changepump servers produce streams of these changes.

Finally, Changepump is responsible for maintaining transactional consistency. It tracks the timestamps of all changes that have been applied to Lightning servers and emits checkpoints that advance the maximum safe timestamp of each partition. This controls when data becomes queryable in Lightning servers.

### *4.8.1 Subscriptions*

Individual Lightning servers manage many partitions for many different tables. For each partition, Lightning maintains a *subscription* to Changepump. The subscription specifies the partition's table and key range, and Changepump is responsible for delivering those changes to the Lightning server.

Subscriptions have a *start timestamp*. Changepump will only return changes that commit after that timestamp. The timestamp may be in the past, in which case Changepump will replay historical changes as needed. This allows Lightning servers to resume their subscriptions at the appropriate point in the case of system failure.

### *4.8.2 Change data*

Changepump subscriptions return two kinds of data: *change updates* and *checkpoint timestamp updates*. Change updates contain *change rows* in the subscribed table key ranges. Each change row includes the row's primary key, the values modified in the transaction, and the operation (insert, update, delete). Changes from the same primary key are delivered in ascending timestamp order, but there are no strict ordering guarantees on cross-row deliveries. This means that the maximum safe timestamp may vary for different primary keys.

Because maintaining per-key timestamp state is expensive, Changepump implements a checkpointing mechanism. Changepump periodically sends a checkpoint timestamp to each subscriber indicating that all changes prior to that timestamp have been delivered. This permits Lightning servers to advance their maximum safe timestamp using the checkpoint rather than keeping state for each primary key.

For efficiency reasons, Changepump does not deliver a checkpoint timestamp along with every single change update. Changepump batches change reading from the change log and divides the work into parallel substreams that are loosely coordinated, meaning that checkpoint generation has a non-trivial cost. How often Changepump sends a checkpoint timestamp is a trade-off between Lightning data freshness and change processing efficiency (see section 7.1 for details on this trade-off).

### *4.8.3 Schema changes*

Lightning uses two mechanisms to detect schema changes: *lazy detection* and *eager detection*.

Every change that Changepump receives from the OLTP database is annotated with the schema version used to generate that change. For lazy detection, we simply check each change to see if it refers to a schema that Lightning has not previously seen. If we see such a change, we pause change processing for that partition until we have loaded and analyzed the new schema.

Lazy detection has a drawback since it can increase change processing delay around each schema change, which impacts the freshness of data seen by queries running over Lightning. In order to reduce this cost, we also have an eager detection model, where a background thread polls the OLTP database to see if any new schema changes have occurred.

Using both lazy and eager detection ensures that Lightning is likely to detect schema changes soon after they occur, without any noticeable interruption in change processing (see Figure 4 for the average data processing delay). The Lightning safe time can be advanced only when both the schema and the data is caught up.

### *4.8.4 Sharding*

Although Changepump is logically a single service, in practice it is implemented as a sharded service where many servers handle a single change log. This means that a single subscription may internally connect to multiple Changepump servers. The Changepump client library merges multiple such connections into a single change stream, making this process invisible to the Lightning server implementation.

Sharding like this is necessary because Changepump servers are partitioned differently than Lightning servers. Changepump servers are partitioned with the goal of roughly-equal partitions in terms of the number of new writes each partition sees, whereas Lightning servers are partitioned with the goal of roughly-equal partitions in terms of the total number of rows contained in each partition.

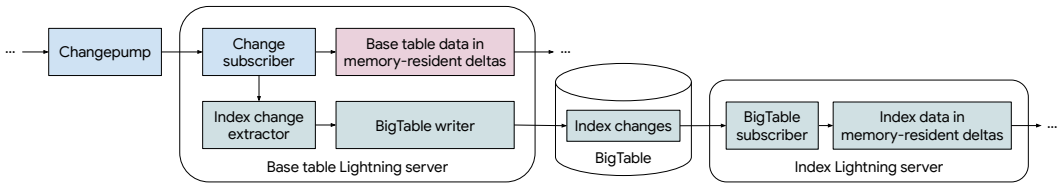


**Figure 3:** Flow of data for a secondary index. Lightning servers maintaining base table partitions read changes from Changepump, generate the corresponding index mutations, and write those mutations to BigTable. Lightning servers maintaining index partitions watch for changes in BigTable and apply them to their memory-resident deltas. Once data is in memory, compaction and reads are as shown in Figure 1.

Changepump also exploits physical data locality in the change log to reduce the cost of reading changes for multiple tables, while Lightning servers partition each table independently.

### 4.8.5 Caching

Changepump servers also contain a memory-resident cache of recent change records. This cache is used to improve change delivery throughput and to reduce expensive I/O operations over the OLTP database's change log. There are two cases that this cache improves.

First, Lightning keeps multiple replicas per Lightning partition for fault tolerance. These replicas all subscribe to the same changes from Changepump. The cache allows changes to be retrieved from the change log once and then shared by these different replicas.

Second, when a Lightning server restarts, it needs to catch up its memory-resident state to the latest timestamp. The Changepump cache allows recent changes to be replayed without reading the source change log, which reduces the time needed for server recovery.

### 4.8.6 Secondary indexes and views

In addition to replicating base tables, Lightning can also maintain secondary indexes and materialized views containing single-table aggregates. From the perspective of Lightning's storage engine, indexes and views are no different than standard tables; we think of them as *derived tables*. However, since derived tables are not present in the original OLTP database, Lightning uses a different method of replicating their changes.

Instead of subscribing to changes from Changepump servers, partitions for derived tables subscribe to changes from a generated "change log." This change log is actually written by the Lightning servers maintaining partitions of the base table. Whenever the base table changes, the Lightning servers will compute and emit the relevant changes for each derived table.

As astute readers may have noticed, Lightning needs to solve a shuffling problem for the derived table change log. Maintainers of derived table partitions only want to subscribe to changes for specific partitions, thus they expect the change log to be sorted by the derived table key. The base table partitions that produce the derived table changes, however, may be maintained by many different Lightning servers.

To solve this problem, Lightning uses BigTable [14] as a shuffle and storage medium. Lightning servers maintaining base table partitions write derived table changes into a BigTable sorted by the key order of each derived table, and Lightning servers maintaining derived table partitions will scan the BigTable for changes to their local key ranges. These servers also write checkpoint timestamps for derived tables according to the checkpoint timestamps received from Changepump.

Because Lightning processes the change log of already committed transactions, it never sees rollbacks, and therefore writes for derived table changes are idempotent. Servers restarting and servers loading the same partition can write the same change log for the derived table for redundancy, overwriting idempotent rows in the BigTable.

Currently, Lightning supports only a limited set of query patterns for materialized view maintenance, in particular simple aggregations. These views tend to be the most common type used at Google, and therefore this fulfills the needs of the majority of our users. Extending support to more complex materialized views, such as joins, is left for future work.

### 4.8.7 Online repartitioning

Periodically, Lightning may need to repartition existing tables in order to ensure even load across all partitions. To facilitate this, Lightning supports dynamically changing the range partitioning on Lightning tables without impacting data ingestion or queries. The range repartitioning scheme is mostly a metadata-only operation, which means that Lightning can repartition frequently to adapt to fast changing write traffic and fast changing data size.

Lightning splits a partition when 1) the total size of a partition becomes much larger than the target partition size, or 2) when write load approaches ingestion bandwidth of a single partition.

For the first case, Lightning calculates the split points of new partitions to rebalance to the target partition size. For the second case, Lightning peeks at the change log to find changes that have not yet been processed, then selects split points that evenly divide those future change rows.

Once it has the split points for new partitions, Lightning registers new, inactive partitions. This is a metadata-only operation—the new partitions start off by sharing the deltas of the source partition. After the new partitions are registered, Lightning servers start Changepump subscriptions for their key ranges and maintain them as usual.

Lightning does not stop maintaining the source partition while the new partitions are being registered. Since there's some delay associated with starting the new Changepump subscriptions, Lightning does not make the new partitions active until they have caught up. At that point, Lightning will atomically deactivate the source partition and activate the new partitions. The source partitions will continue to be used for any queries that were running at the time it was deactivated, but Lightning will delete them once those queries complete.

Since the new partitions were created by sharing data in the source partition, they may contain some rows that are out of range for the new partition boundaries. Lightning handles this by applying a filter at read time to ensure that only the expected rows are returned from each partition. The excess rows will eventually be removed as part of the next compaction process that involves the new partitions.

Lightning also supports merging partitions following a similar mechanism. When $M$ partitions are to be merged into $N$ partitions, we generate new partitions as a metadata-only operation using the union of the source partitions. The rest of the process is the same.

Partition merging is triggered to combine small partitions into larger ones to reduce the overhead of having too many partitions. The size of a partition can shrink if many rows in it are deleted and those deletes age out of the queryable window.

## 4.9 Fault tolerance

Lightning is a critical production service at Google, backing core Google products like AdWords and Payments where outages cause direct revenue loss. Accordingly, Lightning is designed to cope with failures at various layers. In extreme cases, Lightning supports table-level failover back the OLTP database through a configurable blacklisting mechanism.

### *4.9.1 Coping with query failures*

There are two kinds of common query-time failures. In one, a server assigned to read a partition cannot be reached due to scheduled or unplanned server maintenance. In the other, the assigned server hits errors reading data from the storage due to transient network or I/O hiccups. We use intra- and cross-cluster replication to handle these failures.

Within a data center, Lightning assigns each partition to more than one Lightning server. These Lightning servers all subscribe for the same changes from Changepump, and they maintain their memory-resident deltas independently. They share the same set of disk-resident deltas and checkpoints of memory-resident deltas (data on disk is replicated by the distributed file system).

Only one of the replicas will transpose memory-resident deltas to disk-resident columnar deltas to avoid duplicate work. When a replica finishes writing, it notifies other replicas to update their LSM tree.

Queries can be served by any of these server replicas as they all hold the complete state. Queries will be load balanced among these replicas. Lightning can also transiently increase more replicas for partitions with heavier query traffic (hot spots).

Lightning also arranges scheduled server updates such that at most one replica of each partition is restarted at the same time to maintain high data availability. Additionally, when a server restarts, it attempts to load changes from its replicas if possible; this is cheaper than reading the same changes from Changepump, even accounting for the Changepump cache.

The complete Lightning stack is deployed in multiple data centers. An individual data center has its own set of Changepump servers, Lightning servers, task workers, and Lightning masters. These servers operate independently of the other data centers, maintaining complete copies of the data in each data center. Keeping independent LSM trees in each data center incurs less coordination overhead, and each data center can maintain its state with minimal cross-data-center network cost.

All data centers share the same metadata database. The metadata database is a multi-homed, highly reliable database service, so we assume it is always available and does not have a single point of failure. Table schemas and table partitioning is shared across data centers. If change replication gets stuck in one partition in a data center, reads for that partition fail over to the same partition in another data center served by different servers.

The transactional database replicated by Lightning is usually multi-homed too. However, Lightning can choose to replicate in a completely different location than the transactional storage, and it can replicate to more locations than the original database is stored in. This means that this multi-homing capability is not just about fault tolerance—it is also the preferred way for database owners to expand serving capability and improve data locality for query serving by bringing the storage closer to where applications run.

### *4.9.2 Coping with ingestion failures*

The fault tolerance mechanisms above reduce the risk of failing queries when data has already been ingested into Lightning. However, there can be failures that prevent the ingestion of data in the first place. For example, Changepump servers may crash, or an outage in the OLTP system's change data capture functionality may prevent Lightning from recognizing new changes.

For the former, failure of a single Changepump server is handled by connecting to a different Changepump server in the same data center. Any Lightning server can connect to any Changepump server, and we run several Changepump replicas in each data center.

For the latter, failure to replicate a piece of data within the OLTP system's change data capture functionality can increase the read latency significantly in regions that have not finished the replication. Changepump uses a load-balancing channel to connect to the OLTP system's CDC component. The channel redirects Changepump to read from another healthy region of the OLTP system when it detects an unhealthy region (at a slightly increased cost of network bandwidth and latency). Lightning master also monitors the Changepump lag from all data centers on every partition—whenever a partition's delay in one data center is larger than other data centers by a configurable threshold, the Lightning master will restart the partition in the outlier data center by copying previously ingested data from a healthy data center. This allows the blockage of incoming data to be cleared faster than catching up individual changes through CDC in the slow data center.

Global failures in the change data capture component of the underlying OLTP system are rare. However, when that happens, it can block the Lightning data replication globally as there is no healthy data center to fast-recover from. For that, Lightning provides a table-level failover mechanism.

### *4.9.3 Table-level failover*

Lightning supports a table-level blacklisting mechanism. When a table is blacklisted, F1 Query will automatically service queries over that table using the data stored in the OLTP database instead of Lightning.

There are a few circumstances where a table may be blacklisted. For example, Lightning runs an offline verifier that continuously checks the consistency of the Lightning replica with the data stored in the OLTP database. If data corruption is detected, Lightning can blacklist the affected table to prevent queries from returning incorrect results. Additionally, if a table experiences a high rate of changes that Lightning is unable to ingest, Lightning will blacklist that table in order to avoid degrading the system-wide maximum safe timestamp. When the table has recovered, Lightning will automatically remove it from the blacklist.

As previously mentioned in section 4, most analytic queries can still be run on the OLTP system directly, albeit at a much higher cost (see Table 2). Accordingly, while it is not desirable to run all traffic over the OLTP system, this table-level fall back mechanism gives us a powerful way to localize and isolate failures from the rest of the system, improving overall availability and reliability.

There is a trade-off involved, as some users may prefer to serve stale data with better performance rather than fall back to fresher data. To handle this, Lightning provides database owners with the option of configuring how stale the data must be before failover occurs. More sophisticated failover configuration can be added too: for instance, users can configure Lightning so that high-priority traffic that needs fresher data fails over during an outage while low-priority traffic stays on Lightning reading stale data to avoid competing with high-priority traffic over scarce OLTP resources.

# 5. F1 QUERY INTEGRATION

Google's technology stack is highly layered, with different layers such as query and storage managed by separate systems and often developed by completely separate teams. While this architecture has many advantages at the organizational level, it can cause increased costs associated with unnecessary data serialization and conversion at API boundaries. In order to avoid this, we standardized on F1 Query as the interface for Lightning—all Lightning reads are serviced by F1 Query—and we carefully co-designed two major features: *transparent query rewrites* and *subplan pushdown*.

## 5.1 Transparent query rewrites

As we have mentioned, Lightning is completely transparent to the user or application issuing queries—the entity issuing the query continues querying against the OLTP database without modifying their queries, and they may not even know that Lightning is in the picture. Lightning accomplishes this by integrating with F1 Query's snapshot isolation mechanism.

Read-only queries sent to F1 Query execute against snapshots of the database. Users are allowed to select a specific timestamp if desired, or they may omit the timestamp and let F1 Query pick a recent timestamp for them. The read timestamp picked by F1 Query is called the *query safe timestamp*. Most queries use the query safe timestamp by default.

When Lightning is enabled, F1 Query always picks Lightning's database-level maximum safe timestamp as the query safe timestamp. This allows all queries without explicit timestamps to use Lightning, at the cost of reducing data freshness. We actively monitor the health of the production instance to ensure that freshness is not unacceptably reduced. If a user manually specifies a timestamp, Lightning is still used provided that timestamp is within Lightning's queryable window.

F1 Query always generates logical plans as if it was querying the OLTP database directly. Generating logical plans in this way simplifies that part of the system, and it ensures that query planning behavior is semantically identical to querying against the OLTP database, including critical functionality like authorization checks. Then, if the query is running at a timestamp that can be served by Lightning, F1 Query considers Lightning as an additional access path for each table during physical planning. Lightning-only indexes and views are exposed to the optimizer at this stage as well. If Lightning is chosen as the access path, F1 Query runs additional physical rules, such as subplan pushdown, to optimize for Lightning's characteristics.

## 5.2 Subplan pushdown

Operator pushdown is a common technique to improve query performance. As a federated query engine, F1 supports operator pushdown for several existing data sources. However, the pushdown on most data sources is typically restricted to simple filters that are expressible in the data source's limited filter pushdown API.

F1 Query has recently developed a vectorized query evaluation engine to replace its original row-oriented evaluator. We observed up to an order of magnitude improvement in performance by transitioning to a column-oriented evaluator, and we have also taken this opportunity to implement several optimizations for encoded columns (similar to those in Procella [15]) and refactor the code to make it more modular.

As a result of this effort, we were able to embed the vectorized evaluator directly in Lightning servers. This enables the F1 Query optimizer to consider a rich set of options for pushdown into Lightning servers—in principle, any operation that can be evaluated by F1 Query can also be evaluated by Lightning.

At query planning time, the F1 Query optimizer splits the query plan into parts that will be executed in stateless F1 workers, and parts that will be pushed down to Lightning servers. For parts that are pushed down, the serialized execution plan is sent to Lightning servers as part of the read request. Currently, F1 Query pushes down leaf subtrees that require no data shuffling, such as filters, partial aggregations, and projections.

The results of the Lightning server execution are serialized and transmitted to the F1 workers during distributed execution. Since Lightning reuses the evaluation engine from F1, it uses the same column-oriented in-memory data format and wire format as F1 Query. This format is similar to Apache Arrow [1] and is standardized across the F1 ecosystem. Using this format enables F1 Query workers to directly deserialize data from Lightning servers as if they were an F1 Query worker, with no data conversion.

# 6. ENGINEERING PRACTICES

In this section, we describe a few engineering practices we followed during the development of the Lightning system. These practices have not only proven to be helpful to Lightning, but they are now also adopted more widely in the development of F1 Query and other data infrastructure projects at Google.

## 6.1 Reusable components

During Lightning's development, we created and contributed to many general-purpose system components that are now also adopted in other query, storage, and data processing systems.

One reason that makes Lightning components reusable is that libraries developed for Lightning have minimal dependencies on system-specific data structures or APIs. Lightning is developed by the same team that built F1 Query. While it might be appealing to use existing data structures in F1 Query to build libraries, that hard dependency would make it more difficult for other systems at Google to adopt it. Instead, we developed standalone libraries, like Lightning's column-wise in-memory format and F1 Query's vectorized evaluation engine, with reuse in mind. These libraries are now used across Google.

Another advantage of this organization is that it encourages defining proper API boundaries for higher level system components. For example, Changepump was originally designed for Lightning, but it later became a more generalized change subscription service that is also used by TableCache [22], an in-memory read through cache over F1 DB, and by Lightning's internal metadata database change propagation. Furthermore, Lightning's storage layer is being used to build other storage systems with alternate data ingestion designs.

## 6.2 Correctness verification

Since Lightning manages business-critical databases at Google, correctness is a key requirement. We built several correctness verification frameworks to help catch issues during development, and we continue to use those validation tools as part of our regular release process to catch software bugs.

To verify data integrity, we built a data verifier that periodically scans the entire database and compares each row in the OLTP database with each row in Lightning. The data verifier limits its parallelism to avoid overloading the OLTP system; detecting diffs can take from hours up to a few days on very large databases.

To catch bugs in the integration with the query system, we use a query replay tool. The replay tool extracts all queries from the production log, runs these queries against the transactional database and Lightning using the same snapshot timestamp, and compares normalized query results. We normalize queries by deduplicating all queries that only differ in constant values, selecting a consistent

order, and filtering out queries that have side effects or use non-deterministic functions. The query replay tool replays one query per distinct normalized query pattern.

Query replay is helpful to find bugs that are not caused by data corruption in persistent storage. For example, many bugs related to pushdown execution can be caught only by query replay. The query replay tool also extracts latency and resource metrics from running queries, which provides useful signals to detect performance regressions. Regressions in the query system are usually caught within a week.

# 7. COST AND BENEFITS

In this section, we discuss the operational costs involved in Lightning as well as its benefits to Google's real-world production systems. We use Lightning to execute only read-only SQL queries that are not part of a transaction, and it is generally cheaper to run non-transactional SQL queries on Lightning (see Table 2). Transactional workloads that make modifications still run directly over the OLTP system and thus are excluded from the cost comparison. The total computational cost of running those transactional workloads is a very small fraction of the cost of running analytic queries in databases that enable Lightning.

However, data replication is not free, and in particular, the resource cost of Lightning comes in two parts: the replication layer and the storage layer.

When new tables are added to Lightning, there is a one-time cost to create an initial snapshot of the table. After that point, the replication layer incrementally applies changes, with ongoing cost roughly linear with respect to the size of writes in the OLTP database, assuming the OLTP database's change data capture feature supports partitioning and linear-time change tailing. This cost is similar to the cost of maintaining replicas using log shipping, except that it only pays the log shipping cost for tables that are replicated by Lightning.

In storage, Lightning keeps a copy of the replicated table in a column-oriented format. This overhead is similar to some dedicated HTAP storage systems that keep both a row-oriented and a column-oriented store. In addition to speeding up analytical queries, Lightning can also be used for geo-replication and isolating read-only workloads from read/write workloads, which can be used to further justify the additional storage.

For read-intensive applications, the large amount of resources saved by efficient analytical query computation can make Lightning a net resource win, particularly if these HTAP replicas displace replicas of the OLTP system data. It is still generally the case that computational resources such as CPU and RAM are more expensive than storage media such as HDD and SSD, so it makes sense to pay the storage cost if it can significantly reduce the computational cost. In the following sections, we present some metrics gathered from production at Google that show the benefits of Lightning on real-world workloads.

## 7.1 Safe timestamp delay

Figure 4 shows Lightning safe timestamp delay in the Ads instance. Lightning's observed safe timestamp delay depends on two explicit architectural decisions.

First, Lightning tries to minimize the impact on the OLTP database where it extracts changes. Because Lightning replicates a large-scale, geo-replicated OLTP database, it has several options for replicating changes. For example, Lightning could replicate changes as soon as they are committed to a single replica. However, this leads to substantial hotspotting on replicas that handle writes, with limited load on other replicas. Instead, Lightning replicates changes only after they have been committed to all replicas. This


**Figure 4:** Delay of the Lightning maximum safe timestamp relative to an OLTP database in production at Google.

allows individual Lightning replicas to replay changes from their local OLTP replica and more evenly distributes the load, but it incurs increased delay. Similarly, Lightning batches reads for changes into discrete time windows instead of reading the change log for every single transaction, which reduces the total number of reads that must be issued to the OLTP database.

Second, in general, Lightning prefers data availability over data freshness (subject to the maximum delay discussed in section 4.9.3). For example, the advertised safe timestamp delay of a Lightning instance is determined by the delay of the slowest table partition, even if there are partitions with lower delay. Similarly, F1 Query runs Lightning queries at the oldest safe timestamp advertised across all data centers. Using these conservative timestamps ensures that queries are able to read from any replica, which minimizes the number of queries that must read from the OLTP database in case of localized system failures.

## 7.2 Hybrid query workload latency

F1 Query supports running queries of all workloads: OLTP queries, OLAP queries, and ETL queries. All of these workloads use Lightning when applicable, and Lightning's data storage is tuned to optimize query performance for hybrid workloads. Table 1 shows the performance characteristics of a single Lightning instance on various workload types.

OLTP queries are typically point lookup queries and executed in F1 Query's central mode running on a single machine. Lightning has fast point lookup performance thanks to the PAX file layout with sparse indexing, effective caching, and an efficient LSM component.

Distributed queries on F1 Query and Lightning may run in hundreds of distributed F1 Query workers and Lightning servers. These queries are efficient thanks to fast range scans, secondary indexes, and views. There are also system-generated queries that run common query patterns over a fixed set of tables that benefit from caching.

Beyond machine-generated queries, Lightning also runs ad-hoc analysis queries sent from human users. These queries tend to have more diverse query patterns and hence varied query latencies; they often contain full table scans without an applicable secondary index and join data from other data sources, such as files stored in a distributed file system. These queries can have runtimes of many minutes depending on the exact query pattern and data sources read.

ETL queries run in F1 Query's batch mode, which runs F1 Query's execution kernel using the MapReduce [18] framework for distributed computation. These are resource-intensive queries that tend to scan entire tables, compute joins, transform the data, and write the results into the distributed file system. Several expensive ETL queries may be chained to run together in a single pipeline. These queries run from minutes up to several hours. Even though scanning from the data source is only a part of the query cost, serving those scans using Lightning tends to save a large amount of resources and reduce ETL latency due to the columnar file format and subplan pushdown.

**Table 1:** Latency distributions for various workloads running on Lightning. All workloads are served by the same Lightning instance.

| | 50th | 90th | 95th | 99th |
|---|---|---|---|---|
| Single-node queries | 8 ms | 50 ms | 101 ms | 1695 ms |
| Distributed queries | 0.15 s | 1.3 s | 2.4 s | 9.9 s |
| Ad-hoc analysis queries | 13 s | 16 min | 12 min | 50 min |
| ETL queries | 7 min | 49 min | 78 min | 163 min |

**Table 2:** CPU efficiency improvement of queries over Lightning against queries over write-optimized transactional storage. "Small" queries used less than 1 CPU second, "medium" queries used between 1 s and 100 s, and "large" queries used more than 100 s.

| | Small | Medium | Large |
|---|---|---|---|
| Data source CPU time speed-up | 2.3x | 11.8x | 7.6x |
| F1 server CPU time speed-up | 1.5x | 16.9x | 3.8x |

Using a single Lightning instance to serve diverse workloads saves the data owner and the query engine from making decisions for which storage engine to choose for each query pattern.

## 7.3 CPU efficiency comparison

Table 2 compares CPU efficiency when executing the same query on Lightning and on the OLTP database (F1 DB in this case).

We use our query replay tool to run each distinct production query on Lightning and on F1 DB at the same snapshot timestamp. These replays are limited to non-ETL queries since ETL queries all have side-effects like materializing results. We compare the total CPU time for running the same set of queries in two storage systems. The total CPU time is weighted by the number of occurrences of the query pattern appeared in production.

We report a comparison of the CPU spent in the Lightning servers and in the shared F1 Query workers separately. We also group queries into buckets in terms of its CPU cost: small, medium and large. Small queries are those which used less than 1 CPU second. Medium are queries whose CPU time is between 1 s and 100 s. Large are queries whose CPU time is greater than 100 s.

We can see that Lightning is more CPU efficient in all three query buckets and the CPU efficiency in both data source servers and F1 Query servers is improved.

On small queries, the CPU improvement is not as significant because the OLTP database is already optimized for low-latency point reads commonly seen during transactional processing. Despite this, Lightning still shows a noticeable improvement.

The CPU efficiency improvement is most significant on medium queries. The savings on Lightning servers comes from the columnar file layout that helps reduce I/O and data processing for reading structured protocol buffers, range partitioing that enables sequential data access for scans, and vectorized data processing in both storage access and pushdown operations. Pushing down operations enables CPU saving on both F1 Query servers and on Lightning servers. In F1 Query servers, these operations no longer need to be evaluated, and pushing those operations down to Lightning can reduce the amount of data that needs to be serialized and transmitted over the network in the case of filters and aggregates.

The CPU savings on large queries is smaller than medium queries. This is because large queries typically read a large number of bytes and stream those bytes to F1 and to the client. When queries read entire serialized protocol buffers, Lightning's column-oriented storage does not help. The cost for large queries is normally dominated by data serialization and network transfers; although Lightning reduces this cost by sharing a common data representation between F1 Query and Lightning servers, it still plays a large role.

# 8. FUTURE WORK

Of course substantial room for future work remains. Currently, Lightning supports two OLTP systems at Google, both with change data capture support. A future direction could be to extend Lightning beyond transactional sources or to transactional systems without a change data capture component. For those, Lightning may need a mode of change replication with looser consistency guarantees and possibly higher delay.

Another fundamental question is how truly decoupled systems like Lightning can be from existing systems. Lightning is loosely coupled with any transactional system it serves, but tightly coupled with one query engine (F1 Query). For us this decision was reasonable because the GoogleSQL interface to F1 Query makes it simple to insert Lightning between legacy applications and their OLTP storage—if an application uses GoogleSQL, then it is already largely query-engine agnostic. Also, unlike changing OLTP systems, changing query engines does not require data to be migrated to the new system. Still, it is interesting to speculate if it would be possible or beneficial to make a system like Lightning itself query-engine agnostic as well as OLTP-engine agnostic.

# 9. CONCLUSION

HTAP is a broad and important subfield of data management. We believe it is not a simple, single-dimensional area, where approaches or systems can be arranged in a total order of quality by a few metrics. Rather, it is a complex space in which systems and approaches must be evaluated in multiple dimensions with trade offs. Based on the goals of a particular target deployment, some of these dimensions will be more important than others.

We found ourselves in a scenario where important attributes included the ability to transparently improve HTAP performance over multiple OLTP systems without modifying the OLTP systems or migrating users to new ones, working seamlessly with an existing federated query engine, and supporting geo-replicated operation with stringent correctness and performance requirements. We call the system we built for this scenario Lightning and describe what it does as "HTAP-as-a-service."

In this paper, we showed how Lightning enables high-performance analytic queries over hybrid query workloads on top of existing transactional storage systems. We have deployed Lightning for business-critical transactional databases, including AdWords and Payments, in Google's production environment at scale, and achieved up to orders of magnitude savings in computational resources and query latency without compromising on query semantics.


# Acknowledgements

We would like to thank Haris Volos and interns Wangda Zhang and Michael Whittaker for their work during early development. Thanks to our many collaborators across Google, especially the Spanner, F1 DB, and Mesa teams. Great thanks also to early Lightning adopters at Google for their patience and feedback. Thank you to James Balfour, Race Wright, and Jeff Korn for their feedback on this paper. And finally a huge thanks to our F1 Query and F1 SRE colleagues for their support and feedback and without whom the ecosystem within which Lightning runs would not be possible.

# 10. REFERENCES


[1] Apache arrow. https://arrow.apache.org/.

[2] Apache geode. https://geode.apache.org/.

[3] Oracle database in-memory. https://www.oracle.com/a/tech/docs/twp-oracle-database-in-memory-19c.pdf.

[4] Parquet. https://parquet.apache.org/.

[5] Tidb. https://github.com/pingcap/tidb.

[6] Tiflash. http://www.hpts.ws/papers/2019/flash.pdf.

[7] Zetasql. https://github.com/google/zetasql/.

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

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[10] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, and et al. Spanner: Becoming a sql system. In *SIGMOD*, page 331–343, 2017.

[11] R. Barber, C. Garcia-Arellano, R. Grosman, R. Mueller, V. Raman, R. Sidle, M. Spilchen, A. J. Storm, Y. Tian, P. Tözün, et al. Evolving databases for new-gen big data applications. In *CIDR*, 2017.

[12] M. A. Bornea, O. Hodson, S. Elnikety, and A. Fekete. One-copy serializability with snapshot isolation under the hood. In *ICDE*, pages 625–636, 2011.

[13] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *SIGMOD*, page 739–752, 2008.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *TOCS*, 26(2):4:1–4:26, 2008.

[15] B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. Mccormick, A. Mokashi, P. Harvey, H. Gonzalez, D. Lomax, S. Mittal, and et al. Procella: Unifying serving and analytical data at youtube. *PVLDB*, 12(12):2022–2034, 2019.

[16] 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. Spanner: Google's globally-distributed database. In *OSDI*, pages 261–264, 2012.

[17] S. Das, C. Botev, K. Surlaker, B. Ghosh, B. Varadarajan, S. Nagaraj, D. Zhang, L. Gao, J. Westerman, P. Ganti, et al. All aboard the databus! linkedin's scalable consistent change data capture platform. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[18] J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool. *CACM*, 53(1):72–77, 2010.

[19] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 117–130, 2006.

[20] 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. Mesa: Geo-replicated, near real-time, scalable data warehousing. *PVLDB*, 7(12):1259–1270, 2014.

[21] J. Lee, S. Moon, K. H. Kim, D. H. Kim, S. K. Cha, and W.-S. Han. Parallel replication across formats in sap hana for scaling out mixed oltp/olap workloads. *PVLDB*, 10(12):1598–1609, 2017.

[22] G. N. B. Manoharan, S. Ellner, K. Schnaitter, S. Chegu, A. Estrella-Balderrama, S. Gudmundson, A. Gupta, B. Handy, B. Samwel, C. Whipkey, L. Aharkava, H. Apte, N. Gangahar, J. Xu, S. Venkataraman, D. Agrawal, and J. D. Ullman. Shasta: Interactive reporting at scale. In *SIGMOD*, pages 1393–1404, 2016.

[23] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *PVLDB*, 3(1-2):330–339, 2010.

[24] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. Snappydata: A unified cluster for streaming, transactions and interactice analytics. In *CIDR*, 2017.

[25] F. Özcan, Y. Tian, and P. Tözün. Hybrid transactional/analytical processing: A survey. In *SIGMOD*, page 1771–1775, 2017.

[26] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[27] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 155–174, 2004.

[28] B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte, and et al. F1 query: Declarative querying at scale. *PVLDB*, 11(12):1835–1848, 2018.

[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. F1: A distributed SQL database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[30] A. Simitsis, P. Vassiliadis, and T. Sellis. Optimizing etl processes in data warehouses. In *ICDE*, pages 564–575, 2005.

[31] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, and et al. C-store: A column-oriented dbms. In *PVLDB*, page 553–564, 2005.

[32] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, page 2, 2012.