Online, Asynchronous Schema Change in F1

Ian Rae
University of
Wisconsin–Madison
ian@cs.wisc.edu

Eric Rollins
Google, Inc.
erollins@google.com

Jeff Shute
Google, Inc.
jshute@google.com

Sukhdeep Sodhi
Google, Inc.
sodhi@google.com

Radek Vingralek
Google, Inc.
radekv@google.com

ABSTRACT

We introduce a protocol for schema evolution in a globally distributed database management system with shared data, stateless servers, and no global membership. Our protocol is *asynchronous*—it allows different servers in the database system to transition to a new schema at different times—and *online*—all servers can access and update all data during a schema change. We provide a formal model for determining the correctness of schema changes under these conditions, and we demonstrate that many common schema changes can cause anomalies and database corruption. We avoid these problems by replacing corruption-causing schema changes with a sequence of schema changes that is guaranteed to avoid corrupting the database so long as all servers are no more than one schema version behind at any time. Finally, we discuss a practical implementation of our protocol in F1, the database management system that stores data for Google AdWords.

1. INTRODUCTION

Schema evolution—the ability to change a database's definition without the loss of data—has been studied in the database research community for more than two decades [17]. In this paper, we describe techniques used by Google's F1 database management system [21] to support schema evolution in a new setting. F1 is a globally distributed relational database management system that provides strong consistency, high availability, and allows users to execute queries via SQL. F1 is built on top of Spanner, a globally distributed data store [5]. Spanner has evolved over time to provide its own relational abstraction layer which includes some of the features that F1 provides; however, F1 was developed around an earlier version of Spanner, and as such, its implementation details, design goals, and assumptions differ from Spanner. We focus only on the features of Spanner relevant to our protocol for schema evolution and hence view Spanner as a key–value store.

The main features of F1 that impact schema changes are:

Massively distributed An instance of F1 consists of hundreds of individual F1 servers, running on a shared cluster of machines distributed in many datacenters around the globe.

Relational schema Each F1 server has a copy of a relational schema that describes tables, columns, indexes, and constraints. Any modification to the schema requires a distributed schema change to update all servers.

Shared data storage All F1 servers in all datacenters have access to all data stored in Spanner. There is no partitioning of data among F1 servers.

Stateless servers F1 servers must tolerate machine failures, preemption, and loss of access to network resources. To address this, F1 servers are largely stateless—clients may connect to any F1 server, even for different statements that are part of the same transaction.

No global membership Because F1 servers are stateless, there is no need for F1 to implement a global membership protocol. This means there is no reliable mechanism for determining currently running F1 servers, and explicit global synchronization is not possible.

These aspects of F1's architecture, together with its role as a primary data store for Google AdWords, impose several constraints on the schema change process:

Full data availability F1 is a critical part of Google's business infrastructure. Therefore, the availability of the data managed by F1 is paramount—any downtime of the AdWords F1 instance can be measured with a direct impact on Google's revenue, and modifications to the schema cannot take the system offline. Additionally, due to this revenue loss, it is unacceptable to take even a portion of the database offline during a schema change (e.g., locking a column to build an index).

Minimal performance impact The AdWords F1 instance is shared among many different teams and projects within Google that need to access AdWords data. As a result, the F1 schema changes rapidly to support new features and business needs created by these teams.

Proceedings of the VLDB Endowment, Vol. 6, No. 11
Copyright 2013 VLDB Endowment 2150-8097/13/09... $ 10.00.

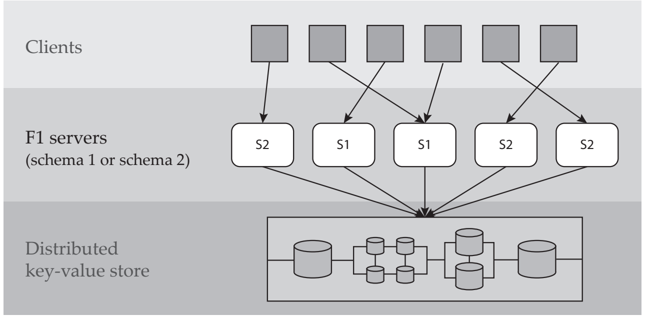

Figure 1: Overview of an F1 instance during a schema change. All servers share the same key–value store, but multiple versions of the schema can be in use simultaneously.

Typically, multiple schema changes are applied every week. Because schema changes are frequent, they must have minimal impact on the response time of user operations.

Asynchronous schema change Because there is no global membership in F1, we cannot synchronize the schema change across all F1 servers. In other words, different F1 servers may transition to using a new schema at different times.

These requirements influenced the design of our schema change process in several ways. First, since all data must be as available as possible, we do not restrict access to data undergoing reorganization. Second, because the schema change must have minimal impact on user transactions, we allow transactions to span an arbitrary number of schema changes, although we do not automatically rewrite queries to conform to the schema in use. Finally, applying schema changes asynchronously on individual F1 servers means that multiple versions of the schema may be in use simultaneously (see Figure 1).

Because each server has shared access to all data, servers using different schema versions may corrupt the database. Consider a schema change from schema S_1 to schema S_2 that adds index I on table R. Assume two different servers, M_1 and M_2, execute the following sequence of operations:

1. Server M_2, using schema S_2, inserts a new row r to table R. Because S_2 contains index I, server M_2 also adds a new index entry corresponding to r to the key–value store.

2. Server M_1, using schema S_1, deletes r. Because S_1 does not contain I, M_1 removes r from the key–value store but fails to remove the corresponding index entry in I.

The second delete leaves the database corrupt. For example, an index-only scan would return incorrect results that included column values for the deleted row r.

Our protocol attempts to prevent this corruption by addressing the general problem of asynchronous, online schema evolution in a distributed database system that has shared data access across all servers. We consider not only changes to the logical schema, such as the addition or removal of columns, but also changes to the physical schema like adding or removing secondary indexes. By ensuring that no more than two schema versions are in use at any given time, and that those schema versions have specific properties, our protocol enables distributed schema changes in a way that does not require global membership, implicit or explicit synchronization between nodes, or the need to retain old schema versions once a schema change is complete. As a result, the **main contributions** of this paper are:

- A description of the design choices we made when building a global-scale distributed DBMS on top of a key–value store and their implications for schema change operations.

- A protocol to execute schema changes asynchronously in a system with shared access to data while avoiding anomalies that lead to data corruption. This protocol allows for concurrent modification of the database by user transactions during a schema change.

- A formal model and proof of correctness of the proposed schema change protocol.

- A discussion of changes made to the system in order to implement the protocol.

- An evaluation of the implementation and efficacy of the protocol in a production system used by Google AdWords.

The rest of the paper is organized as follows. We explain the interface of the key–value store and the high-level design of F1's relational schema and operations in Section 2. In Section 3, we present a model for our supported schema changes and show how we design them to protect against various anomalies that can corrupt the database. We follow this with Section 4, where we describe how we implemented these schema changes in our production F1 system that has been serving Google's AdWords traffic for over a year, and we provide some information on the performance and overall user experience of the system in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2. BACKGROUND

F1 provides a relational view over data stored as keys and values in a key–value store. In this section, we separate the interface provided by the key–value store from its implementation, and we show how we map traditional relational database features into this unique setting.

2.1 Key–value store

F1 is designed for a distributed key–value store with a simple data model that associates **keys** with **values**. F1 assumes the key–value store supports three operations: *put*, *del*, and *get*. *put* and *del* insert or delete a value with a given key, respectively, and *get* returns any stored values whose key matches a given prefix. Note that *put* and *del* reference exactly one key–value pair, while *get* may return multiple key–value pairs.

Additionally, F1's optimistic concurrency control adds two more requirements on the key–value store:

1. **Commit timestamps.** Every key–value pair has a last-modified timestamp which is updated atomically by the key–value store.

Example

first_name*	last_name*	age	phone_number
John	Doe	24	555-123-4567
Jane	Doe	35	555-456-7890

(a) Relational representation.

key	value
Example.John.Doe.*exists*	
Example.John.Doe.*age*	24
Example.John.Doe.*phone_ number*	555-123-4567
Example.Jane.Doe.*exists*	
Example.Jane.Doe.*age*	35
Example.Jane.Doe.*phone_ number*	555-456-7890

(b) Key–value representation.

Table 1: F1's logical mapping of the "Example" table (a) into a set of key–value pairs (b). Primary key columns are starred.

2. **Atomic test-and-set support**. Multiple *get* and *put* operations can be executed atomically.

We describe this in more detail in Section 2.5.

2.2 Relational schema

An F1 **schema** is a set of table definitions that enable F1 to interpret the database located in the key–value store. Each table definition has a list of columns (along with their types), a list of secondary indexes, a list of integrity constraints (foreign key or index uniqueness constraints), and a list of optimistic locks. Optimistic locks are required columns that cannot be read directly by client transactions; we describe them in more detail in Section 2.5. A subset of columns in a table forms the primary key of the table.

Column values can be either primitive types or complex types (specified in F1 as protocol buffers [10]). Primary key values are restricted to only primitive types.

We call a column **required** if its value must be present in every row. All primary-key columns are implicitly required, while non-key columns may be either required or optional.

2.3 Row representation

Rows are represented in the key–value store as a set of key–value pairs, one pair for each non-primary-key column. Each key logically includes the name of the table, the primary key values of the containing row, and the name of the column whose value is stored in the pair. Although this appears to needlessly repeat all primary key values in the key for each column value, in practice, F1's physical storage format eliminates this redundancy [21]. We denote the key for the value of column C in row r as $k_r(C)$.

In addition to the column values, there is also a reserved key–value pair with the special column *exists*. This key–value pair indicates the existence of row r in the table, and it has no associated value, which we denote as $\langle key, null \rangle$. An concrete example of a row is shown in Table 1.

F1 also supports **secondary indexes**. A secondary index in F1 covers a non-empty subset of columns on a table and is itself represented by a set of key–value pairs in the key–value store. Each row in the indexed table has an associated index key–value pair. The key for this pair is formed by concatenating the table name, the index name, the row's indexed column values, and the row's primary key values. We denote the index key for row r in index I as $k_r(I)$, and as in the case of the special *exists* column, there is no associated value.

2.4 Relational operations

F1 supports a set of standard relational operations:

- $insert(R, vk_r, vc_r)$ inserts row r to table R with primary key values vk_r and non-key column values vc_r. Insert fails if a row with the same primary key values already exists in table R.
- $delete(R, vk_r)$ deletes row r with primary key values vk_r from table R.
- $update(R, vk_r, vc_r)$ updates row r with primary key values vk_r in table R by replacing the values of a subset of non-key columns with those in vc_r. *update* cannot modify values of primary keys. Such updates are modeled by a *delete* followed by an *insert*.
- $query(\vec{R}, \vec{C}, P)$ returns a projection $\vec{C}$ of rows from tables in $\vec{R}$ that satisfy predicate P.

We use the notation $write(R, vk_r, vc_r)$ to mean any of *insert*, *delete*, or *update* when we wish to model the fact that some data has changed, but we do not care about the specific type of operation that changed it.

These relational operations are translated into changes to the key–value store based on the schema. For example, $delete(R, vk_r)$ deletes all key–value pairs corresponding to all columns and indexes defined on table R in schema S with primary key vk_r. Therefore, we subscript all operations with their related schema, such as $delete_S(R, vk_r)$.

Whenever we need to distinguish the transaction that issued a particular operation, we superscript the operation with the transaction identifier. For example, we say that $update^1_S(R, vk_r, vc_r)$ is an update operation issued by transaction T_1 using schema S. We introduce a shorthand notation $query(R, C, vk_r)$ for a query reading a single value of column C in row r with primary key vk_r in table R.

Each operation is guaranteed to take the database from one consistent state to another with respect to the schema on which the operation is based.

2.5 Concurrency control

F1 uses a form of timestamp-based optimistic concurrency control similar to that found in Percolator [14], which is also built on top of a distributed key–value store. F1's concurrency control is relevant to schema evolution because F1's schema contains an additional element on each table: **optimistic locks**.

A table may have many locks; however, each column in that table is associated with (or **covered by**) exactly one optimistic lock. Each row has its own instance of each of the optimistic locks defined in the schema, and these instances

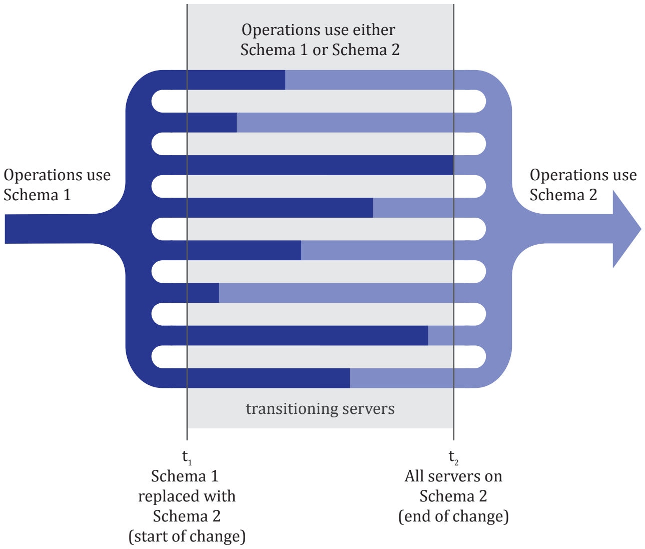

Figure 2: A view of the schema change process from Schema 1 to Schema 2 over time. Lines in the center of the figure represent individual F1 servers as they load Schema 2.

control concurrent access to that row's column values by multiple transactions.

When clients read column values as part of a transaction, they accumulate last-modified timestamps from the locks covering those columns; at commit time, these timestamps are submitted to the server and validated to ensure that they have not changed. If validation succeeds, the last-modified timestamps of all locks associated with columns modified by the transaction are updated to the current timestamp. This form of concurrency control can be shown to be conflict serializable by following the same argument as used in the proof of Theorem 4.17 in [25, pg. 173]. Transactions are limited to at most one logical write operation, which may modify many individual rows, and the write operation occurs atomically with a commit.

By default, as in the majority of database systems, F1 implements row-level locking [21]. However, because users can add new locks to a table and associate them with arbitrary columns in that table, F1 users can select a locking granularity ranging from row- to column-level locking as desired. Consequently, we must consider the correctness of schema changes which add or remove locks from the schema (identical to adding or removing required columns, discussed in Section 3.3), and the correctness of schema changes which modify the associations between columns and locks (discussed in Section 3.4).

3. SCHEMA CHANGES

> *There is nothing as practical as a good theory.*
>
> Kurt Lewin

Soon after we implemented the schema change process in F1, we realized the need for a formal model to validate its correctness, since improperly executed schema changes could result in catastrophic data loss. While we developed our formal model, we found two subtle bugs in our production system, and we also found that some of the schema changes that we had not yet implemented could be supported safely. Finally, by allowing us to reason about our schema change protocol, having a formal model increased our confidence in the F1 schema change process. We describe the most salient features of the formal model in this section.

All servers in an F1 instance share a set of key–value pairs, called a **database representation**, that are located in a key–value store. To interpret these key–value pairs as rows, every F1 server maintains a copy of its instance's schema in its memory, and it uses this schema to translate relational operators into the operations supported by the key–value store. Accordingly, when a client submits an operation, the schema used for that operation is determined by the schema currently in the memory of the F1 server the client is connected to.

The canonical copy of the schema is stored within the database representation as a special key–value pair known to all F1 servers in the instance. When the canonical copy of the schema is replaced with a new version, it begins a **schema change**, which is a process that propagates the new schema to all servers in an F1 instance. Because F1 is a highly distributed system with hundreds of servers with no way to synchronize between them (either explicitly or implicitly), different servers may transition to the new schema at different times (see Figure 2), and multiple schema versions may be in use simultaneously. Accordingly, we say that a schema change has completed only when all F1 servers in the instance have loaded the new schema.

Since all F1 servers in an instance share a single key–value store, improperly executing asynchronous schema changes can corrupt the database. For example, if a schema change adds an index to the database, servers still operating on the old schema will fail to maintain the new index. If this occurs, any queries which perform index reads will return incorrect results.

The fundamental cause of this corruption is that the change made to the schema is, in some sense, too abrupt. Servers on the old schema have no knowledge of the index, while servers on the new schema use it for all operations as if it were fully maintained. Additionally, although we used adding an index as an example, this problem occurs for all fundamental schema change operations in our system.

To address this issue, we devised a protocol for safely executing schema changes that relies on the use of **intermediate states**. With our protocol, elements of the schema (such as tables, columns, indexes, and so on) can be placed in an intermediate state that restricts which operations can be applied to them. This enables us to decompose a single dangerous schema change into a sequence of safe schema changes that we show enforce correct behavior.

To simplify reasoning about the correctness of our implementation, we restrict servers in an F1 instance from using more than two distinct schema versions. In particular, our protocol expects that all servers use either the most recent schema version or a schema that is at most one version old (see Section 4.3 for a practical method of implementing this requirement). This is the fewest number of simultaneous schema versions that we can permit, since the asynchronous nature of our schema change process means that there is always a possibility of having a short window where more than one schema version is in use. It would also be possible to allow more than two schema versions in use simultaneously, but as this would greatly increase the complexity of reasoning about correctness and be of limited utility, we opted not to support it in our protocol.

In order to describe our protocol for safely executing distributed, asynchronous schema changes and to reason about its correctness, we must first discuss the elements found within an F1 schema and the states that can be applied to them.

3.1 Schema elements and states

As described in Section 2.2, an F1 schema has tables, columns, indexes, constraints, and optimistic locks. We describe these collectively as **schema elements** (or just **elements**), and each element in the schema has a state associated with it. There are two states which we consider to be non-intermediate: **absent** and **public**.

As one might expect, if an element is not present in the schema, it is absent. If an element is present in the schema, and it can be affected by or applied to all operations, it is public. Accordingly, these are the two states that are requested by users when elements are added to or removed from the schema.

However, F1 also has the notion of two internal, intermediate states: **delete-only** and **write-only**. The delete-only state is defined as follows:

Definition 1. A **delete-only table**, **column**, or **index** cannot have their key–value pairs read by user transactions and

1. if E is a table or column, it can be modified only by *delete* operations.
2. if E is an index, it is modified only by *delete* and *update* operations. Moreover, *update* operations can delete key–value pairs corresponding to updated index keys, but they cannot create any new ones.

As a result, when an element is delete-only, F1 servers will delete its associated key–value pairs as necessary (e.g., to remove entries from an index), but they will not permit the insertion of any new key–value pairs for that element.

The write-only state is defined for columns and indexes as follows:

Definition 2. A **write-only column or index** can have their key–value pairs modified by *insert*, *delete*, and *update* operations, but none of their pairs can be read by user transactions.

Therefore, this state allows data to be written, but not read (in the case of indexes, F1 servers will not use write-only indexes to accelerate seeks).

The write-only state is also defined for constraints:

Definition 3. A **write-only constraint**[1] is applied for all new *insert*, *delete*, and *update* operations, but it is not guaranteed to hold over all existing data.

In this case, F1 servers will enforce the constraint for new operations on the database, but reads against the database may see data which violates the constraint.

Although it may seem strange to have states where data can be inserted into the database but not read, or read in a manner that seems to violate expectations, recall that different F1 servers may be using different schema versions. We will show in the following sections that careful use of these states is important for ensuring a consistent view of the key–value pairs in the database across all F1 servers.

3.2 Database consistency

Intuitively, all data in the key–value store must correspond to some column or index entry in the schema; otherwise, the key–value store would contain some "garbage" data that is not part of the database, which is clearly undesirable. Additionally, the database must satisfy all constraints that are present in the schema. We now formalize this intuition into a definition of database consistency that enables us to evaluate whether a schema change can corrupt the database.

Definition 4. A database representation d is **consistent with respect to schema S** iff

1. **No column values exist without a containing row and table.** For every column key–value pair $\langle k_r(C), v_r(C)\rangle \in d$ there exists $\langle k_r(\mathit{exists}), \mathit{null}\rangle \in d$ and there exists table $R \in S$ containing column C.
2. **All rows have all public required column values.** For every required public column C in table $R \in S$, if there exists $\langle k_r(\mathit{exists}), \mathit{null}\rangle \in d$, there exists $\langle k_r(C), v_r(C)\rangle \in d$.
3. **No index entries exist without a corresponding index in the schema.** For every index key–value pair $\langle k_r(I), \mathit{null}\rangle \in d$ there exists table $R \in S$ containing index I.
4. **All public indexes are complete.** If there exists a public index I on $R \in S$, then there exists an index key–value pair $\langle k_r(I), \mathit{null}\rangle \in d$ for every row $r \in R$.[2]
5. **All index entries point to valid rows.** Conversely, for every index key–value pair $\langle k_r(I), \mathit{null}\rangle \in d$, there exists a column key–value pair $\langle k_r(C), v_r(C)\rangle \in d$ for every column C covered by index I.
6. **All public constraints are honored.** No key–value pair exists in d that violates a public constraint listed in S, and all key–value pairs that must be present according to public constraints in S exist in d.
7. **No unknown values.** There are no key–value pairs in database representation d except those postulated in Clauses 1 and 3 of this definition.

We denote the fact that database representation d is consistent with respect to schema S as $d \models S$. If d is not consistent with respect to schema S, we denote this as $d \not\models S$.

The consistency of a database representation d with respect to schema S can be violated in two ways:

1. Database representation d contains key–value pairs it should not according to schema S. We call such a violation an **orphan data anomaly**. Specifically, database representation d has orphan data with respect to S if it violates Clauses 1, 3, 5, or 7 of Definition 4.

[1] Section 2.2 defines the constraints we implemented in F1.

[2] As in most relational database systems, F1 does not index rows with missing values in the index key. We do not model this fact for the sake of simplicity.

2. Database representation d is missing a key–value pair it should contain according to schema S or it contains a key–value pair that violates a public constraint in schema S. We say a violation of this sort is an **integrity anomaly**. Specifically, database representation d has integrity anomalies if it violates Clauses 2, 4, or 6 of Definition 4.

Let op_S be any of *delete*, *update*, *insert*, or *query* executing under schema S. Every correctly implemented operation op_S preserves the consistency of any database representation d it is applied to with respect to the schema S it uses. However, it is not guaranteed to preserve consistency with respect to any other schema. Because multiple schema versions are in use during a schema change, failure to preserve consistency with respect to all schema versions easily corrupts the database representation.

To avoid this, we define the notion of a **consistency-preserving schema change**.

Definition 5. A schema change from schema S_1 to schema S_2 is **consistency preserving** iff, for any database representation d consistent with respect to both S_1 and S_2, it is true that

1. any operation op_{S_1} preserves the consistency of d with respect to schema S_2, and
2. any operation op_{S_2} preserves the consistency of d with respect to schema S_1.

In Definition 5, it may appear unnecessary to require that all operations preserve the consistency of the database representation with respect to the old schema S_1. After all, once the schema change completes, all F1 servers will use the new schema S_2. Therefore, as long as all operations preserve consistency with respect to S_2, the database representation will become consistent as soon as the schema change completes.

However, F1 servers using the old schema S_1 that operate on a representation which is not consistent with respect to S_1 may issue operations that yield database representations not consistent with respect to the new schema S_2 (an operation op_{S_1} has an undefined behavior if applied to a database representation $d \not\models S_1$). Consider a schema change from schema S_1 to schema S_2 that adds column C and a database representation d that is consistent only with respect to schema S_2. First, operation $insert_{S_2}(R, vk_r, v_r(C))$ updates database representation d by adding key–value pair $\langle k_r(C), v_r(C)\rangle$. The resulting database representation d' is still consistent with respect to schema S_2 (but not S_1!). Now, suppose operation $delete_{S_1}(R, vk_r)$ is later executed against database representation d'. It fails to remove key–value pair $\langle k_r(C), v_r(C)\rangle$ from database representation d' because column C is not present in schema S_1, making key–value pair $\langle k_r(C), v_r(C)\rangle$ an orphan with respect to schema S_2. Hence the resulting database representation d'' is not consistent with respect to schema S_2, and moreover, the corruption will persist after the schema change completes.

Therefore, we must require in Definition 5 that, during a consistency-preserving schema change, all operations preserve consistency with respect to both the old schema S_1 and the new schema S_2 at all times during a schema change. This property also means that Definition 5 is symmetric:

Claim 1. *A schema change from schema S_1 to schema S_2 is consistency preserving iff a schema change from schema S_2 to schema S_1 is consistency preserving.*

Consistency-preserving schema changes ensure that the database is not corrupted; however, many common schema changes are not consistency preserving.

3.3 Adding and removing schema elements

The most common schema changes requested by F1 users are those that add and remove elements, such as tables, columns, and constraints, from the schema. For ease of explanation, we group tables, columns (including optimistic locks), and indexes together as **structural schema elements** (or simply **structural elements**). Structural elements can be thought of as the elements in the schema that determine the set of allowed key–value pairs in the key–value store. We do not address optimistic locks explicitly in this section since they are semantically identical (from a schema change perspective) to required columns.

Due to the fact that multiple schema versions are in use simultaneously, adding or removing any structural element without using any intermediate states has the potential to corrupt the database.

Claim 2. *Any schema change from schema S_1 to schema S_2 that either adds or drops a public structural element E is not consistency preserving.*

Proof. Consider a schema change from schema S_1 to schema S_2 that adds structural element E and a (possibly empty) database representation $d \models S_1, S_2$.

We examine all possibilities for new structural element E:

E **is a table.** Suppose we apply operation $insert_{S_2}(E, vk_r, vc_r)$ to database representation d, creating database representation d'. $d' \not\models S_1$ because $insert_{S_2}(E, vk_r, vc_r)$ adds key–value pairs which are orphan data with respect to schema S_1, violating Clause 1 in Definition 4.

E **is a column in table** R**.** Suppose we apply operation $insert_{S_2}(R, vk_r, v_r(E))$ to d, creating database representation d''. Similarly, $d'' \not\models S_1$ because the key–value pairs corresponding to E are also orphan data, again violating Clause 1 in Definition 4.

E **is an index on table** R**.** Finally, suppose we apply operation $insert_{S_2}(R, vk_r, vc_r)$ to d, creating database representation d''' such that row r contains all columns indexed by E. $d''' \not\models S_1$ and $d''' \not\models S_2$ because it violates Clauses 3 and 4 in Definition 4, respectively.

The same result follows for structural element drops by Claim 1. □

However, it is possible to prevent these anomalies by judicious use of the intermediate states we described in Section 3.1. These intermediate states, when applied in the proper order, can ensure that no orphan data or integrity anomalies ever occur in the database representation, allowing us to execute consistency-preserving schema changes.

Claim 3. *A schema change from schema S_1 to schema S_2 is consistency preserving iff it avoids orphan data and integrity anomalies with respect to both S_1 and S_2.*

Proof. Follows directly from the definitions of orphan data and integrity anomalies, which together cover all clauses of Definition 4. □

We now discuss the exact intermediate states needed to support addition and removal of optional structural elements, required structural elements, and constraints in turn.

3.3.1 Optional structural elements

Adding and removing public optional elements can cause orphan data anomalies, since some servers have knowledge of elements that other servers do not, and these elements can be freely modified by user transactions. We can eliminate these anomalies by ensuring that elements pass through an intermediate schema that has them in a delete-only state before they are set to public (in the case of additions) or removed (in the case of drops). This enables us to add a new, delete-only element to the schema in a way that does not compromise the consistency of the database representation.

Claim 4. *Consider a schema change from schema S_1 to schema S_2 that adds a delete-only structural element E, with any database representation d such that $d \models S_1$. Then $d \models S_2$, and no operation op_{S_1} or op_{S_2} on E in d can cause orphan data or integrity anomalies with respect to S_1 or S_2.*

Proof. Because $d \models S_1$ and E is absent in S_1, there are no key–value pairs corresponding to E in d. Element E is not public in S_2 and hence no such pairs are required in order to establish that $d \models S_2$.

Assume that E is an index. Operations using S_2 do not add any new key–value pairs relative to operations using S_1; they only delete key–value pairs belonging to E, should they exist. Since neither schema allows the insertion of any key–value pairs corresponding to E, any pair of operations using S_1 and S_2 trivially avoids orphan data anomalies.

Similarly, no integrity anomalies can occur because neither S_1 nor S_2 impose any new requirements or constraints on key–value pairs in the database, as E is optional.

The reasoning is identical if E is a table or column. □

Once the optional structural element is in the schema in the delete-only state, it can be promoted to public without causing further anomalies.

Claim 5. *Consider a schema change from schema S_1 to schema S_2 that promotes an optional structural element E from delete-only to public, with any database representation d such that $d \models S_1$. Then $d \models S_2$, and no operation op_{S_1} or op_{S_2} on E in d can result in orphan data or integrity anomalies with respect to S_1 or S_2.*

Proof. Because $d \models S_1$ and E is delete-only in S_1, there may or may not be key–value pairs corresponding to E in d; however, since E is optional, the presence of such pairs is allowed—but not required—in S_2, and so $d \models S_2$.

Delete operations using schema S_1 will delete the key–value pairs corresponding to structural element E if present, because the element is in a delete-only state in S_1. Similarly, operations using schema S_2 will delete the key–value pairs corresponding to structural element E if present, because E is public. Thus, all operations will avoid orphan data anomalies by deleting key–value pairs corresponding to structural element E as necessary.

Integrity anomalies cannot occur since element E is optional. □

Accordingly, if a structural element is optional, it can be safely added to or dropped from the schema with only a single intermediate schema and the following state transitions (the order is reversed for drops):

$$\text{absent} \rightarrow \text{delete only} \rightarrow \text{public}$$

However, aside from simply following the above state transitions in reverse, drops have an additional step: they must delete the key–value pairs associated with the removed structural element. When the only schema in use has the element in the delete-only state, a **database reorganization process** must remove the element's key–value pairs from the database representation. To be consistent, this must occur before a schema change begins which removes the element from the schema; otherwise, some servers would be using a schema which prohibits such key–value pairs.

We have shown how it is possible to add and remove optional elements with only the delete-only intermediate state; however, to support adding and removing required structural elements and constraints, an additional state is needed.

3.3.2 Required structural elements

Required structural elements require the presence of specific key–value pairs by definition. Accordingly, in addition to the orphan data anomalies that affect optional elements, adding and removing these elements in a public state can also lead to integrity anomalies. In this section, we demonstrate how the write-only state can be used in conjunction with the delete-only state to execute these schema changes in a consistency-preserving manner with the following state transitions (in the case of drops, the order of states is reversed, and the database reorganization takes place before the transition to absent):

$$\text{absent} \rightarrow \text{delete only} \rightarrow \text{write only} \xrightarrow{\text{db reorg}} \text{public}$$

We showed previously in Section 3.3.1 that the transition from absent to delete-only was free of anomalies; accordingly, we now show that the transition from delete-only to write-only cannot cause anomalies.

Claim 6. *Consider a schema change from schema S_1 to schema S_2 that promotes an index or required column E from delete-only to write-only, with any database representation d such that $d \models S_1$. Then $d \models S_2$, and no operation op_{S_1} or op_{S_2} on E in d can result in orphan data or integrity anomalies with respect to S_1 or S_2.*

Proof. Schemas S_1 and S_2 contain identical elements except for element E. Since both S_1 and S_2 contain E in internal states, both schemas allow key–value pairs corresponding to E in d, and neither schema requires that the pairs be present. Consequently, $d \models S_1, S_2$.

We show that no operations on E in d can cause orphans. First, consider the case where structural element E is an index on table R. Assume that there is an orphan key–value pair $\langle k_r(E), \mathit{null} \rangle$ for some row $r \in R$. The orphan could have been formed only by *delete* or *update*. However, delete_{S_1} and update_{S_1} cannot form orphans because E is delete-only in schema S_1. Similarly, delete_{S_2} and update_{S_2} cannot form orphans because E is write-only in schema S_2.

Since neither S_1 nor S_2 require the presence of key–value pairs corresponding to E, integrity anomalies cannot occur.

The same reasoning applies if E is a required column. □

This allows us to add an index or required column in a write-only state without causing orphan data or integrity anomalies. Once an element is in the write-only state, all F1 servers will ensure that it is properly maintained for new data; however, data which existed prior to the schema change may not be consistent. Accordingly, before we can promote the index or required column to public, we must execute a database reorganization process that backfills all missing key–value pairs corresponding to the new element. With this done, the element can then be transitioned to public without causing anomalies.

Claim 7. *Consider a schema change from schema S_1 to schema S_2 that promotes structural element E from write-only to public, with any database representation d such that $d \models S_1, S_2$*[3]*. Then no operation op_{S_1} or op_{S_2} on E in d can cause orphan data or integrity anomalies with respect to S_1 or S_2.*

Proof. We can show that the schema change from schema S_1 to schema S_2 does not result in orphan data anomalies following the same argument as in the proof of Claim 6.

We proceed to show that the schema change from schema S_1 to schema S_2 does not result in integrity anomalies. S_1 and S_2 contain identical elements except for element E; therefore, only E can be involved with any integrity anomalies.

Because $d \models S_2$, it cannot have any integrity anomalies with respect to S_2 at the start of the change. It remains to show that operations executed during the schema change do not form any new integrity anomalies. If E is an optional column, no integrity anomalies can ever occur since E imposes no requirements on the presence of key–value pairs. If E is an index or a required column, *insert* and *update* operations are required to update the key–value representations of E because this is enforced by both the write-only and public states. This prevents the formation of any new integrity anomalies. □

As the promotion of an element from write-only to public can result in neither orphan data nor integrity anomalies, it is a consistency-preserving schema change. We can apply this result to all schema changes which add or drop a structural element by showing that they can be implemented as a sequence of schema changes that maintain database consistency in the presence of concurrent operations.

Theorem 1. *Consider a schema change from schema S_1 to schema S_2 that either adds or drops structural element E, with any database representation d_1 such that $d_1 \models S_1$. Then there is a sequence of consistency-preserving schema changes and at most one database reorganization that transitions all servers to schema S_2 and modifies d_1 to database representation d_2 such that $d_2 \models S_2$.*

Proof. We consider a schema change that adds index I; the other cases are similar. We add index I using the following sequence of consistency-preserving schema changes and a single database reorganization:

1. A schema change from schema S_1 to schema S' that adds index I as delete-only. It follows from Claim 4 and $d_1 \models S$ that the schema change is consistency preserving and $d' \models S'$, where d' is the database representation at the end of the schema change from S_1 to S'.

2. A schema change from schema S' to schema S'' that promotes index I from delete-only to write-only. It follows from Claim 6 and $d' \models S'$ that the schema change is consistency preserving and $d'' \models S''$.

3. A database reorganization that inserts key–value pairs corresponding to index I for all rows in d_1, resulting in database representation $d''' \models S'', S_2$. Because $d''' \models S_2$, it cannot have any integrity anomalies with respect to S_2.

4. A schema change from schema S'' to schema S_2 that promotes index I from write-only to public. This is consistency preserving by Claim 7 and $d''' \models S'', S_2$.

The claim for dropping index I follows from performing the same sequence of schema changes with the states applied in the reverse order, using Claim 1. In this case, database reorganization happens prior to the final schema change, and it drops the index key–value pairs instead of adding them. □

Accordingly, we can add and remove any structural schema element in a manner that preserves the consistency of the database representation with respect to all schemas involved.

3.3.3 Constraints

F1 supports foreign key and index uniqueness integrity constraints. Adding and removing these constraints can cause integrity anomalies; for example, if a uniqueness constraint is added to a column without using intermediate states, servers using the old schema will allow the insertion of duplicate values, causing a violation of the constraint from the perspective of servers on the new schema. These anomalies can be prevented by first adding the constraint in the write-only state with the following state transitions (again, the order is reversed for drops):

$$\text{absent} \rightarrow \text{write only} \rightarrow \text{public}$$

Because this is similar to the write-only state used for required structural elements, we omit a detailed discussion and proof of this process in the interest of space. The same mechanism could be used to enforce arbitrary integrity constraints in addition to those supported by F1.

3.4 Changing lock coverage

Ordinarily, database concurrency control is not affected by changes to the schema. However, recall that F1 represents optimistic locks in the schema as named objects. Users are able to modify the granularity of locking on a per-table basis by changing which optimistic locks cover which columns. As the mechanics of adding or dropping a lock are equivalent to adding or dropping a required column, which we discussed in Section 3.3.2, we focus here on lock coverage changes.

Lock coverage schema changes modify which lock handles concurrency control for a given column. Note that lock drops implicitly cause a corresponding lock coverage change, since all columns must be covered by some lock. Changes to lock coverage can allow non-serializable schedules if they are improperly implemented.

Claim 8. *A schema change from schema S_1 to S_2 which changes the coverage of column C from lock L_1 to lock L_2 allows non-serializable schedules.*

Proof. Suppose we have a schedule with transactions T_i and T_j ($i \neq j$):

$$\begin{aligned} &query^i_{S_1}(R, C, vk_r) \\ &< write^j_{S_2}(R, vk_r, v_r(C)) \\ &< write^i_{S_1}(R, vk_r, v_r(C)) \end{aligned}$$

[3] Requiring $d \models S_2$ implies that the database reorganization must be executed prior to the schema change from S_1 to S_2.

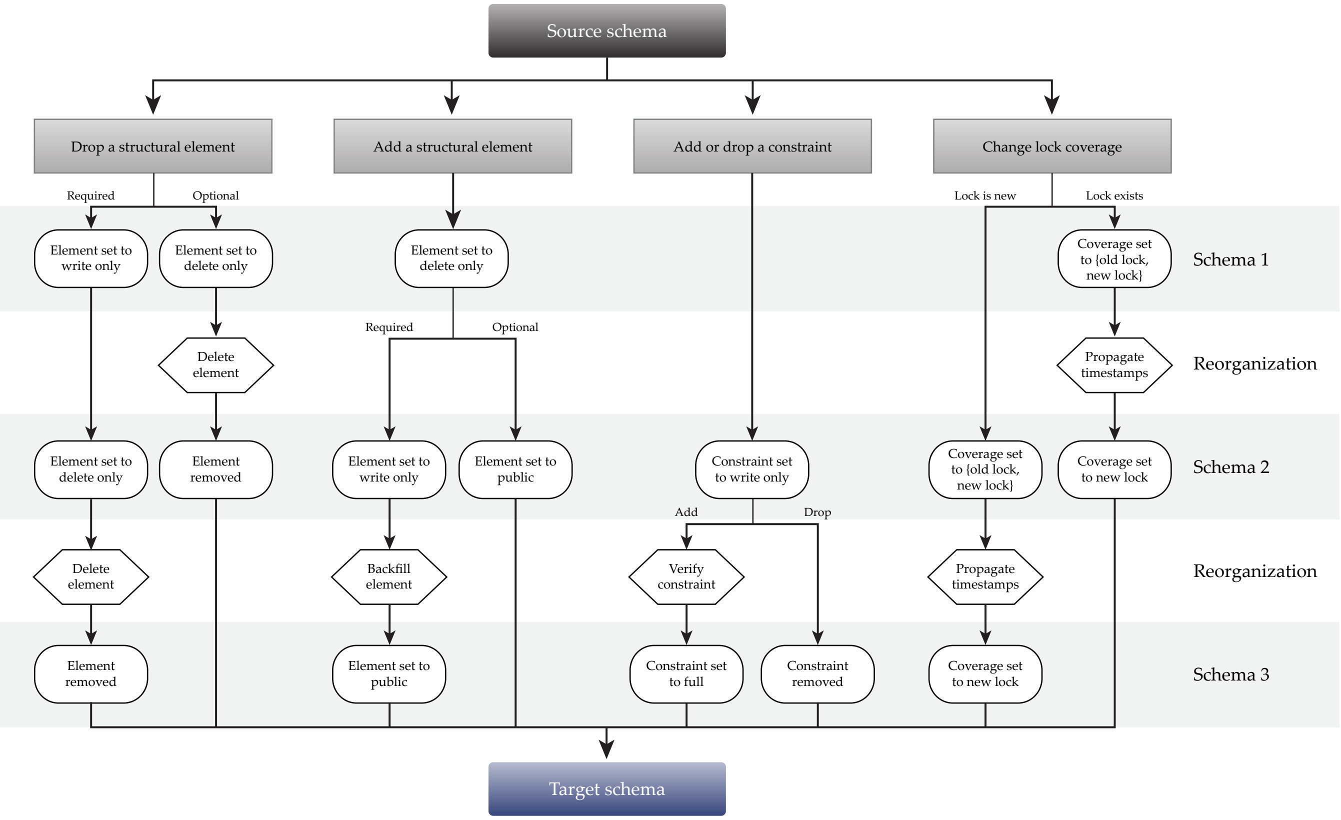

Figure 3: Intermediate states used during schema changes. State transitions occurring in the same row of the figure are combined into a single schema change, and if no state transitions in a given row are necessary, that schema change is skipped.

If T_i commits, this schedule is not serializable: there is a Read–Write conflict between transaction T_i and T_j. However, since the transactions are using different schemas with different lock coverage for C, this schedule will be (erroneously) permitted. Operations using S_1 read and update the lock timestamp from L_1, while operations using S_2 read and update the lock timestamp from L_2. Therefore, transaction T_j's write will not update L_1's timestamp, allowing transaction T_i's validation (and subsequent commit) to succeed despite the fact that the value of C has changed. □

We correct this problem by allowing columns to be temporarily covered by multiple locks. When changing a column's lock coverage from one lock to another, we insert an internal state in which the column is covered by both the old lock and the new lock. We refer to this as the **dual-coverage** state, and operations executing against a schema with a column in the dual-coverage state must read, validate, and update all locks associated with that column.

It would appear that it suffices to split a lock coverage schema change into two schema changes, with a schema having dual coverage as an intermediate step. However, such a composition of schema changes still allows non-serializable schedules. Consider column C covered by lock L_1 in schema S_1, locks L_1 and L_2 in schema S_2, and lock L_2 in schema S_3. Suppose locks L_1 and L_2 have the same timestamp, t_1. This can occur when a row is created as the result of an *insert* operation. Now consider the following schedule with transactions T_i and T_j $(i \neq j)$:

$$\begin{aligned} & \mathit{query}^i_{S_1}(R, C, vk_r) \\ < \; & \mathit{write}^j_{S_1}(R, vk_r, v_r(C)) \\ < \; & \mathit{write}^i_{S_3}(R, vk_r, v_r(C)) \end{aligned}$$

In the first operation, T_i receives timestamp t_1 for lock L_1. In the second operation, T_j updates the timestamp for lock L_1 to timestamp t_2 $(t_2 > t_1)$. When T_i submits its write operation using S_3, timestamp t_1 is validated by the server against the current lock listed in S_3, which is L_2. L_2's timestamp has not changed as a result of the above schedule, so t_1 passes validation, permitting a non-serializable schedule.

We prevent concurrency anomalies as shown above by using a database reorganization. After all servers have transitioned to a schema with all lock coverage changes in the dual-coverage state, we execute a reorganization process that propagates lock timestamps by atomically setting:

$$\mathit{timestamp}(L_2) = \max(\mathit{timestamp}(L_1), \mathit{timestamp}(L_2))$$

for every row with a column in the dual-coverage state.

With this modification, we show that a pair of schema changes can accomplish lock coverage without admitting any non-serializable schedules.

Claim 9. *Any schema change from schema S_1 to schema S_2 that changes lock coverage of column C from lock L_1 to lock L_2 can be replaced with a reorganization and a sequence of schema changes that avoid non-serializable schedules.*

PROOF. Let S' be a schema where column C is covered by both locks L_1 and L_2. We replace the schema change from schema S_1 to S_2 with the following: a schema change from S_1 to S', a reorganization process that propagates the timestamp of L_1 instances to L_2 instances if needed, and a schema change from S' to S_2.

During the schema change from S_1 to S', all operations effectively validate against L_1. Although L_2 is present and maintained by operations using S_2, it is not yet used for concurrency control. Therefore, the only schedules permitted are those that would be permitted if C was covered by L_1, which correspond to serializable schedules.

After the first schema change, but before starting the schema change from S' to S_2, the reorganization process ensures that $\mathit{timestamp}(L_2) \geq \mathit{timestamp}(L_1)$ in every row. Consequently, a schedule that is rejected when not performing a lock coverage change and using only lock L_1 (i.e., a non-serializable schedule) cannot be admitted when using either dual coverage (both locks L_1 and L_2), or only L_2. Therefore, only serializable schedules can be admitted during a schema change from S' to S_2. □

We summarize the intermediate states required for each schema change and the transitions between them in Figure 3. Note that state transitions for multiple elements can be combined into a single schema change.

4. IMPLEMENTATION

We implemented support for these schema changes in a production F1 instance that serves all of Google's AdWords traffic. Our implementation has been used to successfully perform schema changes with no downtime across hundreds of individual F1 servers. Because this instance of F1 is shared among many different groups in Google, our implementation has some details unique to this environment, and we describe these now.

4.1 Spanner

The discussion in Section 3 focused on the correctness of the schema change protocol given a generic interface for a key–value store. We intentionally designed this interface to be as generic as possible to enable broad application of our schema change protocol.

In practice, F1 is built on top of Spanner [5], and several of Spanner's features have influenced F1's implementation. Here, we highlight two Spanner features that are relevant to F1 schema changes: garbage collection and write fencing.

Garbage collection. Spanner has a schema which describes the allowed set of key–value pairs in the database. Any key–value pairs that are not allowed cannot be accessed and are eventually garbage-collected. This enables us to drop structural elements without database reorganization, since removing them from the Spanner schema is identical to proactively deleting all associated key–value pairs.

Write fencing. Write operations against the key–value store are formed by the F1 server using its current schema. If a single write operation takes a long time to commit, the write may take effect after multiple schema changes have occurred, which violates our requirement that operations be based on a schema that is no more than one version old. Spanner allows us to place a deadline on individual write operation so that the writes do commit past the deadline.

Of these two features, only write fencing is required for correctness; garbage collection is useful only for performance and ease of implementation.

4.2 Schema change process

F1 does not implement typical DDL operations (e.g., ALTER TABLE or DROP COLUMN) for performing schema changes. Applying changes individually using DDL statements is impractical, because much of the overhead (e.g., reorganizations) can be amortized if multiple schema changes are batched together.

Instead, F1 represents the entire database schema as a protocol buffer-encoded file [10, 21]. A version of this file is generated from sources stored in a version control system; when users need to modify the schema, they update the sources in the version control system to include their desired change. This allows us to batch several updates to the schema into one schema change operation, and it provides us with a log of all schema changes.

Twice per week, administrators take the schema from the version control system and apply it to the running F1 instance. Before the new schema can be applied, an analysis process determines which intermediate states and reorganizations are required in order to safely perform the schema change. Once the intermediate schemas and reorganizations are determined, an execution process applies them in order while ensuring that no more than two schema versions can be in use at any time (see Section 4.3).

A single intermediate schema may apply state transitions to many different elements in the schema. For details on how we overlap these state transitions, see Figure 3.

4.3 Schema leases

Our schema change protocol requires that F1 servers use at most two different schemas concurrently. Because F1 does not maintain global server membership, we cannot contact servers directly to check which schema version they are running or to update them with a new schema. Additionally, even if global membership data were available, F1 would need a method for dealing with unresponsive servers.

We address this by granting each F1 server a **schema lease**, with typical values of multiple minutes. F1 servers renew their lease by re-reading the schema from a well-known location in the key–value store every lease period. If a server is unable to renew its lease, it terminates—because F1 servers are run in a managed cluster execution environment, they will be automatically restarted on a healthy node.

Additionally, user transactions are allowed to span lease renewals; however, we limit individual write operations to use only schemas with active leases. This is necessary because write operations are translated into operations on the key–value store at the time they are submitted. If an individual write operation takes longer than two lease periods to execute, it could violate our requirement that only the two most recent schema versions can be used at any time.

We use write fencing to ensure that no write operation can commit if the schema it is based on has an expired lease.

This allows us to maintain the following invariant:

INVARIANT 1. *If schema S is written at time t_0 and no other schema is written between times t_0 and t_1 ($t_1 > t_0 + \mathit{lease\_period}$), then at time t_1 each F1 server either uses S or is unable to commit transactions.*

As a consequence, we ensure that at any moment F1 servers use at most two schemas by writing a maximum of one schema per lease period. The schema change execution process must wait at least one schema lease period before applying a new intermediate schema. Therefore, a typical schema change involving no reorganization takes 10 to 20 minutes. We could limit the number of simultaneously used schemas to $k + 1$ by writing schemas at a rate not exceeding k per lease period, this would have complicated reasoning about the system's correctness.

We optimize the lease renewal process in several ways. First, we replicate the canonical copy of the schema in several locations in Spanner, and we use Spanner's atomic test-and-set to update them all atomically. This provides fault tolerance and improved read performance.

Second, we optimize lease renewal by caching the commit timestamp of the currently loaded schema on the F1 servers. When an F1 server renews its lease, it first examines the commit timestamp of the canonical schema version. If the read timestamp is identical to the cached timestamp, the F1 server does not need to read the schema itself from Spanner.

4.4 Data reorganization

Some schema changes, like adding an index, require corresponding updates to the database representation. We perform these updates with a **background reorganizer** that takes several factors into account:

- It is impractical to assume that the entire reorganization can be done atomically. Therefore, the reorganizer's operation must be resumable and idempotent.
- All data must be available while the reorganizer is executing. As a result, the reorganizer must tolerate concurrent access to the data being modified.
- Although not required for correctness, the reorganizer should avoid re-executing a data change that has been performed by a user transaction (e.g., re-adding an index key–value pair that already exists).

We built our reorganizer using the MapReduce framework [8]. The MapReduce controller partitions the database and assigns partitions to map tasks. The map tasks scan all rows in their assigned partitions at a snapshot timestamp corresponding to the start of the schema change, updating each row to conform to the new schema if necessary. Depending on the schema change, the map task will either add key–value representations of an added structural element (such as a new index or lock) or remove representations of a dropped structural element (such as a dropped column).

Each map task reads the representation of each row assigned to it and determines whether it has been updated by a user transaction since the the reorganization has began. If so, the map task does not modify the row any further, following the Thomas write rule [24]. Otherwise, it adds or removes key–value pairs as needed.

4.5 Schema repository

Since schema changes occur frequently in our production F1 instance, it is important that each F1 server does not delay or fail a large number of user operations when moving from one schema to another. To manage this transition, each server stores a **schema repository** in its volatile memory.

	Query latency (ms)		
Percentile	No schema change	Schema change	Increase
50%	2.84	2.94	3.5%
90%	79.82	74.23	-7.0%
99%	170.33	178.05	4.5%

Table 2: The effect of schema changes on query latencies.

The schema repository stores multiple schema versions along with the lease times associated with them[4]. The schema repository maintains several invariants:

1. New write operations use the most recent schema version available in the repository.
2. After a new schema version is loaded by the repository, pending write operations are allowed to complete using their previously assigned schema version.
3. All submitted write operations are terminated through write fencing when the lease on their associated schema version expires.

Write operations terminated due to lease expiry are resubmitted by the server, but this causes wasted work. We can reduce the likelihood of failing write operations by renewing schema leases early; however, more frequent lease renewal increases load on both F1 and Spanner. In our experience, we have found renewing a lease when it has half of the lease period remaining to be a reasonable setting.

5. IMPACT ON USER TRANSACTIONS

Schema changes in F1 often involve data reorganizations, which perform global database updates. One of the design goals of the implementation of schema change in F1 is to minimize the impact of such reorganizations on user operation response times.

We analyzed the logs of the AdWords production system in order to evaluate the impact schema changes had on user transactions. The production system consists of approximately 1500 F1 servers evenly distributed in five datacenters. The logs contain records of 19 schema changes that span a period of approximately 10 weeks.

In this period, over one billion user operations occurred when no schema change was in progress, and over 50 million operations occurred during a schema change. These operations consisted of approximately 75% queries and 25% writes (inserts, deletes, and updates). Query and write latencies outside and during schema change can be found in Tables 2 and 3, respectively.

6. RELATED WORK

Mariposa [22] suggested that schema changes in distributed systems should not rely on global membership, and other work has brought up the idea that such changes should be online and non-blocking [11, 19]. Aspects of our work are similar to a "soft schema change," which allows old transactions to finish with a schema version that has been replaced [19]. Our protocol is per operation instead of per

[4] Transactions can use at most two schema versions simultaneously. However, read-only queries against database snapshots use the schema version of the snapshot.

	Write latency (ms)		
Percentile	No schema change	Schema change	Increase
50%	107.16	179.26	67.3%
90%	497.27	616.08	23.9%
99%	965.69	1076.69	11.5%

Table 3: The effect of schema changes on write latencies.

transaction, and we generalize their intermediate state for indexes into our write-only state.

Much of the work on schema evolution is complementary to our own. In particular, researchers have proposed special syntax for schema evolution [6,13,18], models for capturing the effect of schema changes and automatically rewriting queries [2,4,6,7,9,13,23], and methods for executing temporal queries [15,16]. Although we have not implemented these features, they can be used in conjunction with our work.

Schema evolution in distributed systems has largely focused on data warehouses and federated systems [1,3,12,20]. This work addresses the difficult problem of integrating the schemas of disparate systems with independent data into the appearance of a unified database with a single overall schema. In contrast, our work addresses issues that arise when multiple versions of a schema are used to interpret and modify a common set of data.

Finally, Spanner also has support for schema changes [5]. However, their implementation relies on the ability to perform a synchronous change through synchronized clocks and global membership information.

7. CONCLUSION

We examined schema evolution in F1, a distributed relational database built on top of a distributed key–value store. Our method allows users to evolve the schema with no loss of availability, no global synchronization, and no risk of data corruption. We implemented our protocol in production servers, and we have successfully executed hundreds of schema changes in a critical system under constant use.

Creating a formal model of the schema change process has had several positive effects on our production system. In particular, it highlighted two subtle bugs in our implementation, and it showed that some schema changes which were not supported by our system could be easily implemented.

8. ACKNOWLEDGMENTS

The authors would like to thank the members of F1 and Spanner teams at Google and Jeff Naughton for their feedback on early versions of the paper. The last author would like to dedicate this paper to the memory of his advisor, Yuri Breitbart.

9. REFERENCES

[1] Bellahsene, Z. Schema Evolution in Data Warehouses. *KAIS*, 4(3):283–304, 2002.

[2] Bernstein, P. A. et al. Model Management and Schema Mappings: Theory and Practice. In *VLDB*, pp. 1439–1440. 2007.

[3] Blaschka, M., et al. On Schema Evolution in Multidimensional Databases. In *DaWaK*, pp. 802–802. 1999.

[4] Bonifati, A., et al. Schema Mapping Verification: The Spicy Way. In *EDBT*, pp. 85–96. 2008.

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

[6] Curino, C., et al. The PRISM Workbench: Database Schema Evolution Without Tears. In *ICDE*, pp. 1523–1526. 2009.

[7] Curino, C. A., et al. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *VLDB*, 4(2):117–128, 2010.

[8] Dean, J. et al. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1):107–113, 2008.

[9] Domínguez, E., et al. Model–Driven, View–Based Evolution of Relational Databases. In *DEXA*, pp. 822–836. 2008.

[10] Google Inc. Protocol Buffers - Google's data interchange format, 2012. URL `http://code.google.com/p/protobuf/`.

[11] Løland, J. et al. Online, Non-blocking Relational Schema Changes. In *EDBT*, pp. 405–422. 2006.

[12] McBrien, P. et al. Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach. In *CAiSE*, pp. 484–499. 2006.

[13] Papastefanatos, G., et al. Language Extensions for the Automation of Database Schema Evolution. In *ICEIS*. 2008.

[14] Peng, D. et al. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*. 2010.

[15] Pereira Moreira, V. et al. Schema Versioning: Queries to The Generalized Temporal Database System. In *DEXA*, pp. 458–459. 1999.

[16] Rizzi, S. et al. X-Time: Schema Versioning and Cross-Version Querying in Data Warehouses. In *ICDE*, pp. 1471–1472. 2007.

[17] Roddick, J. Schema evolution in database systems: an annotated bibliography. *ACM SIGMOD Record*, 21(4):35–40, 1992.

[18] Roddick, J. SQL/SE – A Query Language Extension for Databases Supporting Schema Evolution. *ACM SIGMOD Record*, 21(3):10–16, 1992.

[19] Ronstrom, M. On-line Schema Update for a Telecom Database. In *ICDE*, pp. 329–338. 2000.

[20] Rundensteiner, E., et al. Maintaining Data Warehouses over Changing Information Sources. *CACM*, 43(6):57–62, 2000.

[21] Shute, J., et al. F1: A Distributed SQL Database That Scales. *VLDB*, 6(11), 2013.

[22] Stonebraker, M., et al. Mariposa: a wide-area distributed database system. *VLDB*, 5(1):48–63, 1996.

[23] Terwilliger, J., et al. Automated Co-evolution of Conceptual Models, Physical Databases, and Mappings. *ER*, 6412:146–159, 2010.

[24] Thomas, R. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM TODS*, 4(2):180–209, 1979.

[25] Weikum, G. et al. *Transactional Information Systems*. Morgan Kaufmann, 2002.