Online, Asynchronous Schema Change in F1

Google

July 16, 2024

1 Introduction

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
- *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.

F1 is built on top of Spanner. Features:

- **Massively distributed**
- **Relational schema**
- **Shared data storage**: All F1 servers in all datacenters have access to all data stored in Spanner.
- **Stateless servers**
- **No global membership**

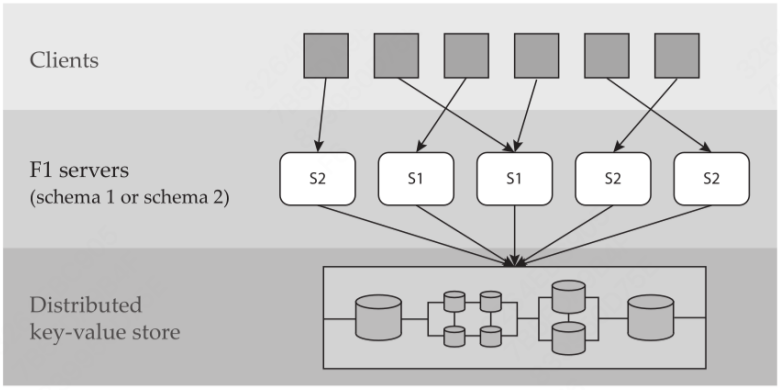

Figure 1: Overview of an F1 instance during a schema change. All servers share the same key–value store, but multiple versions of the schema can be in use simultaneously.

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 remvoe the corresponding index entry in I.

2 Background

2.1 Key-value Store

F1 assumes the key-value store supports three operations: put, del and get. 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.

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

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)
- a list of optimistic locks.

Optimistic locks are required columns that cannot be read directly by client transactions; 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). 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

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. 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 $\langle key, null \rangle$.

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.

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

- $insert(R, vk_r, vc_r)$ 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 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. 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. 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.

2.5 Concurrency Control

F1's concurrency control is relevant to schema evolution because F1's schema contains an additional element on each table: **optimistic locks**.

Each column in the table is associated with exactly one optimistic lock. Each row has its own instance of each of the optimistic locks defined in the schema, and these instances 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

3 Schema Changes

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.

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 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.

3.1 Schema Elements and States

Definition 3.1. 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 a 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.

Definition 3.2. 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.

Definition 3.3. A **write-only constraint** is applied for all new $insert$, $delete$, and $update$ operations, but it is not guaranteed to hold over all existing data.

3.2 Database Consistency

Definition 3.4. A database representation d is **consistent w.r.t. schema** S iff

1. **No column values exist without containing row and table**. For every column key-value pair $\langle k_r(C), v_r(C)\rangle \in d$, there exists $\langle k_r(exists), null\rangle \in d$ and there exists table $R \in S$ containing column C.

4 Problems

5 References

References