An overview of Google F1
(with an emphasis on schema change)

Ian Rae
ian@cs.wisc.edu
Talk outline

Introduction and background

Design overview

Schema changes

Conclusion
Introduction and background
Where Google makes its cash
AdWords overview

~97% of Google’s revenue is from advertising!

Need to track lots of info:
  
  Customer information
  Ad campaign preferences
  Displayed ads
  Clicked ads
  Follow-through purchases
  …
AdWords technology ecosystem

MySQL

???

???

???
The homegrown parallel RDBMS blues

Data partitioned across dozens of MySQL instances.

Have to manually repartition to add servers.

Developers make assumptions about where data lives.

Limited cross-machine transactions.
Revenue paranoia

Data *synchronously replicated* across multiple machines.

Can handle machine failure; what about *datacenter failure*?
AdWords technology ecosystem

???

???

F1

???

???

???
F1 design overview
What is F1?

F1 is a distributed, relational database designed for both OLAP and OLTP.

Full SQL support with ACID semantics for transactions.

Two main design goals

Fault tolerance

Scalability
Fault tolerance
F1 is globally distributed

A single F1 instance consists of thousands of servers in datacenters across the globe.

Data is synchronously replicated across datacenters.
F1 architecture

**F1 servers**
(query processing)

**Spanner**
(cross-datacenter storage)

**Colossus**
(distributed filesystem)
Spanner: next-generation BigTable

Spanner does the storage-related “heavy lifting” for F1.

Spanner uses Paxos and 2PC to synchronously replicate data across datacenters.

More Spanner features

Spanner supports **strict two-phase locking** for **pessimistic transactions**.

Spanner provides **guaranteed unique commit timestamps** for transactions.
F1 and Spanner

F1 uses Spanner mostly as a **key-value store**:
- Get(key prefix)
- Put(key, value)
- Delete(key)

Spanner pessimistic transactions are used to enable **atomic test-and-set of multiple values**.
Scalability
Stateless servers

All data is shared among all servers.

Servers can be added or removed with no data movement.

Clients can send a request to any server, even different requests that are part of the same transaction.
Transactions in F1

Use a form of **optimistic concurrency control**, with all state stored on the client (not F1 server).

Limited to **one atomic write** operation (**implicitly commits**).

Spanner **pessimistic** transactions also supported, but **not stateless**.
Optimistic lock columns

Every column is covered by a hidden optimistic lock column containing a last-modified timestamp.

When a column is updated, the commit timestamp of the updating transaction is stored in its covering lock column.
Configurable locking granularity

Users can specify which lock covers a column.

By default, all columns in a row are covered by a default lock.
Optimistic transactions: reads

When an optimistic transaction reads a column value, it also reads the corresponding lock timestamp.

<table>
<thead>
<tr>
<th>Lock name</th>
<th>Buffered timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock1</td>
<td>$ts_1$</td>
</tr>
<tr>
<td>Lock7</td>
<td>$ts_2$</td>
</tr>
<tr>
<td>Lock3</td>
<td>$ts_3$</td>
</tr>
</tbody>
</table>

Lock timestamps for all reads are buffered on the client for the duration of the transaction.
Optimistic transactions: write + commit

At commit, **all buffered timestamps** are validated against the **lock timestamps currently in the database**.

<table>
<thead>
<tr>
<th>Lock name</th>
<th>Buffered timestamp</th>
<th>Current timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock1</td>
<td>$ts_1$</td>
<td>$ts_1$</td>
</tr>
<tr>
<td>Lock7</td>
<td>$ts_2$</td>
<td>$ts_4$</td>
</tr>
<tr>
<td>Lock3</td>
<td>$ts_3$</td>
<td>$ts_3$</td>
</tr>
</tbody>
</table>

If there is a **mismatch**, the transaction **aborts**.
Optimistic transaction example

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Lock1</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>27</td>
<td>ts₂</td>
</tr>
</tbody>
</table>

**T1**: read (Age) -> get value 26, read lock1 and get $ts_1$

**T2**: write (Age) -> set value = 27, lock1 is updated to $ts_2$

**T1**: commit -> validate lock1 ($ts_1 \neq ts_2$), abort
Schema changes
F1 servers use a schema to interpret key-value pairs as rows and to translate relational operations into key-value operations.
Why is schema change in F1 important?

Data in F1 is critical to Google’s business.

Any downtime or corruption is measured in dollars!

The AdWords F1 instance is shared by many teams with hundreds of developers.

Schema changes requested daily.
Why is schema change in F1 hard?

Every F1 server has a **local cached copy** of the schema.

To **change the schema**, we need to **update all the caches**, but **synchronizing** across all F1 servers is **slow**.

Until the change finishes, no operations can execute → no money!
The goal for schema changes in F1

Enable changes to the logical and physical schema of an F1 instance in a way that is online and asynchronous.

**Online**
All data accessible, no downtime, and without large delays for transactions.

**Asynchronous**
Different servers transition to a new schema at different times.
A paper is available

A protocol for online, asynchronous schema change that permits no database corruption.

A formal model for reasoning about and proving the correctness of our protocol.

Some terminology

Schema elements
Any part of the schema, e.g., tables, columns, constraints, etc.

Structural elements
Tables
Columns
Indexes
Locks
User-visible states of schema elements

Absent
Doesn’t exist!

Public
Available for all operations.
Ensuring correctness

Use **intermediate states** that restrict allowed operations on an element.

Decompose **incompatible** schema changes into a **series of changes** that are **pair-wise compatible**.
An illustration

All servers on \textit{old} schema.

Both schemas in use!

All servers on \textit{new} schema.
Supported schema changes

Add/drop structural elements
- Table add + drop
- Column add + drop
- Index add + drop
- Lock add + drop

Concurrency control
- Change lock coverage

Add/drop constraints
- Change column type
- Make column unique/non-unique
- Foreign key add + drop
- Make column required/optional
- Change protocol buffer definition
- ...
Adding and dropping structural elements
Index add corruption

Change from schema S to S', **adding index I** on table R.

Corruption!
Intermediate states for structural elements

Delete only
Updated by delete operations; cannot be read.

Write only
Updated by delete and insert operations; cannot be read.
Index add revisited

Change from schema $S_1$ to $S_4$, adding index $I$ on table $R$. 
Index add: absent to delete only

Index $I_1$ doesn’t exist.

Index $I_2$ exists, updated only by deletes.
Index $I_2$ is not used for reads.

Index is always empty, but unused.
Index add: delete only to write only

Index I exists, updated only by **deletes**.
Index I is **not used** for reads.

Index I exists, updated by **deletes & inserts**.
Index I is **not used** for reads.

All servers delete entries, so no dangling entries are possible.
Index add: backfill

Index I exists, updated by **deletes & inserts**. Index I is **not used** for reads.

A MapReduce starts to **backfill** index I.

All servers maintain index for new rows.
Index add: write only to public

Index I is **completely backfilled**.

Index I is **public and ready to use**.
Adding and dropping constraints
Constraint corruption

Change from schema $S$ to $S'$, making column $C$ unique.

**Problem:** servers on schema $S$ can insert duplicates into column $C$ that servers on schema $S'$ don’t expect!

**Bonus problem:** how do we verify that column $C$ only contains unique values?
Intermediate states for constraints

Write only
Constraint applies to inserts and updates, but is not guaranteed to hold for reads.
Constraint add revisited

Change from schema $S_1$ to $S_3$, making column $C$ unique.
Constraint add: absent to write only

Column C is **not unique**.

Column C **cannot** have duplicates **inserted**. Reads **may show duplicates**.
Constraint add: verification

Column C **cannot** have duplicates **inserted**. Reads **may show duplicates**.

A MapReduce starts to **verify** that column C contains only unique values.

No server allows new duplicates to be inserted.
Constraint add: write only to public

Column C is **verified unique**.

Column C is unique for **reads and writes**.
Concurrency control
Concurrency corruption

Change from schema $S$ to $S'$, changing the lock coverage of column $C$ from $L_1$ to $L_2$.

**Problem:** servers on schema $S$ don’t validate writes to column $C$ by servers on schema $S'$. 
Concurrency corruption example

**T1**: read(C) using $S$ → read $ts_1$ from $L_1$

**T2**: write(C) using $S'$ → update $L_2$ to $ts_2$

**T1**: write(C) using $S$ → validate $ts_1$ against $L_1$ (works!)

Corruption!
Intermediate states for lock coverage

Dual coverage
A column is covered by two locks.
Dual coverage semantics

On a **read**, the timestamp returned is the **maximum** of **both locks**.

On a **write**, the timestamp is **validated** against **both locks**.
Lock coverage change revisited

Change from schema $S_1$ to $S_3$, changing lock coverage of column $C$ from $L_1$ to $L_2$. 
Coverage change: $L_1$ to dual coverage

Column $C$ is covered by $L_1$.

Column $C$ is covered by $L_1$ and $L_2$.

$L_1$ handles concurrency control.
Coverage change: dual coverage to $L_2$

Column $C$ is covered by $L_1$ and $L_2$.

Column $C$ is covered by $L_2$.

$L_2$ handles concurrency control.
More concurrency corruption

Suppose both $L_1$ and $L_2$ have the same timestamp.

**T1**: read(C) using $S_1$ -> read $ts_1$ from $L_1$

**T2**: write(C) using $S_1$ -> update $L_1$ to $ts_2$

**T1**: write(C) using $S_3$ -> validate $ts_1$ against $L_2$ (works!)
Coverage change: propagation

Column C is covered by $L_1$ and $L_2$.

A MapReduce sets $L_2 = \max(L_1, L_2)$.

Timestamps propagate from $L_1$ to $L_2$. 
Some implementation details
Schema leases

**Canonical schema** file is stored in Spanner.

Once per **lease period**, F1 servers **reload** the canonical schema if needed.

If a server cannot read the schema, it **terminates and restarts**.
Modifications to the schema are first committed to source control, not a live F1 instance.

Schema change process periodically applies modifications present in source control as a batch.
More details in the paper!

Paper has a lot of other stuff
Formal model and proofs
Concurrency control schema changes
Details on overlapping state transitions
Discussion of MapReduces needed
More implementation details
...

BUT WAIT THERE'S MORE!
Conclusion

F1 is a globally distributed, fault-tolerant relational database that serves as the main data store for Google AdWords.

Check out the papers for more details:
