F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business

Jeff Shute, Stephan Ellner
{jshute,stephano}@google.com

Strata

February 28, 2013
Today's Talk

F1 - A Hybrid Database combining the
- Scalability of Bigtable
- Usability and functionality of SQL databases
Built on Spanner, Google's new globally replicated storage system

Key Ideas
- Scalability: Auto-sharded storage
- Availability & Consistency: Synchronous replication
- High commit latency, but can be hidden using
  - Hierarchical schema
  - Protocol buffer column types
  - Efficient client code

Can you have a scalable database without going NoSQL? Yes.
The AdWords Ecosystem

One shared database backing Google's core AdWords business

- advertiser
  - SOAP API
  - web UI
  - reports
  - log aggregation
  - spam analysis

- ad-hoc SQL users
  - ad servers
  - ad approvals
  - ad logs

Java / "frontend"

C++ / "backend"
**Our Legacy DB: Sharded MySQL**

**Sharding Strategy**
- Sharded by customer
- Apps optimized using shard awareness

**Limitations**
- **Availability**
  - Master / slave replication -> downtime during failover
  - Schema changes -> downtime for table locking
- **Scaling**
  - Grow by adding shards
  - Rebalancing shards is extremely difficult and risky
  - Therefore, limit size and growth of data stored in database
- **Functionality**
  - Can't do cross-shard transactions or joins
Demanding Users

Critical applications driving Google's core ad business
- 24/7 availability, even with datacenter outages
- Consistency required
  - Can't afford to process inconsistent data
  - Eventual consistency too complex and painful
- Scale: 10s of TB, replicated to 1000s of machines

Shared schema
- Dozens of systems sharing one database
- Constantly evolving - several schema changes per week

SQL Query
- Query without code
Our Solution: F1

A new database,

● built from scratch,
● designed to operate at Google scale,
● without compromising on RDBMS features.

Co-developed with new lower-level storage system, Spanner
What is Spanner?

Google's new globally distributed storage system

Scalability
- Transparent sharding, data movement

Replication
- Synchronous cross-datacenter replication (with Paxos)

Fault Tolerance

Transactions
- General transactions (ACID)
  - Standard row-level locking
  - Local or cross-machine (using 2PC)
Spanner Timestamps

**TrueTime**: Scalable synchronized clocks (with bounded error)
- Every write picks a commit timestamp *locally*
- These timestamps provide total ordering of commits *globally*

Result: External Consistency
- Commit order == Timestamp order == Read visibility order
- No clock skew anomalies

Consistent multi-versioned reads
- Repeatable reads at any timestamp
- Servers know when a timestamp is stable
- Consistent reads across servers, without locks
- Long-running queries or MapReduces with stable data
TrueTime Implementation

- "True" clock signal provided in hardware
- Worst-case local clock drift: 200 μs/sec
- Poll every 30s => error of 1-7 ms
"Global wall-clock time" with bounded uncertainty

TrueTime returns range \([\text{now.earliest}, \text{now.latest}]\).

Details: http://research.google.com/archive/spanner.html
F1

Architecture
- Sharded Spanner servers
  - data on GFS and in memory
- Stateless F1 server
- Worker pools for distributed SQL execution

Features
- Relational schema
  - Consistent indexes
  - Extensions for hierarchy and rich data types
  - Non-blocking schema changes
- Multiple interfaces
  - SQL, key/value R/W, MapReduce
- Change notifications
Hierarchical Schema

Relational tables, with hierarchical clustering. Example:

- **Customer:** Key (CustomerId)
- **Campaign:** Key (CustomerId, CampaignId)
- **AdGroup:** Key (CustomerId, CampaignId, AdGroupId)

**Rows and PKs**

1

1,3

1,4

1,3,5

1,3,6

1,4,7

2

2,5

2,5,8

**Storage Layout**

- Customer (1)
  - Campaign (1,3)
    - AdGroup (1,3,5)
    - AdGroup (1,3,6)
  - Campaign (1,4)
    - AdGroup (1,4,7)
- Customer (2)
  - Campaign (2,5)
    - AdGroup (2,5,8)
Clustered Storage

- Child rows under one root row form a cluster
- Cluster stored on one machine (unless huge)
- Transactions within one cluster are most efficient
- Very efficient joins inside cluster (can merge with no sorting)

**Rows and PKs**

```
1,3,5  1,3,6  1,4,7
```

**Storage Layout**

```
Customer (1)
  Campaign (1,3)
    AdGroup (1,3,5)
    AdGroup (1,3,6)
  Campaign (1,4)
    AdGroup (1,4,7)

Customer (2)
  Campaign (2,5)
    AdGroup (2,5,8)
```
Protocol Buffer Column Types

Protocol Buffers
- Structured data types with optional and repeated fields
- Open-sourced by Google, APIs in several languages

Column data types are mostly Protocol Buffers
- Stored like blobs in Spanner
- SQL syntax extensions for reading nested fields
- Coarser schema with fewer tables - inlined objects instead

Why useful?
- Protocol Buffers pervasive at Google -> no impedance mismatch
- Simplified schema and code - apps use the same objects
  - Don't need foreign keys or joins if data is inlined
### SQL on Protocol Buffers

```sql
SELECT CustomerId, Whitelist
FROM Customer
```

<table>
<thead>
<tr>
<th>CustomerId</th>
<th>Whitelist</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>feature</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>feature_id: 18</td>
</tr>
<tr>
<td></td>
<td>status: ENABLED</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>feature</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>feature_id: 269</td>
</tr>
<tr>
<td></td>
<td>status: ENABLED</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>feature</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>feature_id: 302</td>
</tr>
<tr>
<td></td>
<td>status: ENABLED</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

```sql
SELECT CustomerId, f.*
FROM Customer c
PROTO JOIN c.Whitelist.feature f
WHERE f.feature_id IN (269, 302)
AND f.status = 'ENABLED'
```

<table>
<thead>
<tr>
<th>CustomerId</th>
<th>feature_id</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>269</td>
<td>ENABLED</td>
</tr>
<tr>
<td>123</td>
<td>302</td>
<td>ENABLED</td>
</tr>
</tbody>
</table>
F1 Query Engine

- Fully functional SQL - joins, indexes, etc.
- Highly parallel global scans
  - Complex plans: arbitrary joins, partitioning and shuffling
  - In-memory and streaming whenever possible
- Local joining in spanner when possible
- Reading data at timestamp T
  - Always consistent
  - Always current data (a few seconds old)

One query engine used for
- User-facing applications (OLTP)
- Live reporting
- Analysis (OLAP)
- Joining to external data sources with stats and logs data
Example Distributed Query Plan

```sql
SELECT *
FROM Campaign JOIN Customer USING (CustomerId)
WHERE Customer.Info.country = 'US'
```
F1 Change History

- Every F1 transaction writes a Change History row
  - Keys, before & after values (as protocol buffer)
- Publish notifications to subscribers
  - "Customer X changed at time T"

Subscriber
- Checkpoint time CT per Customer
- Read changes in (CT, T)
- Process changes, update checkpoint

Uses
- Incremental extraction, streaming processing of changes
- Caching data in clients
  - Force catch-up to T, with no invalidation protocol
How We Deploy

Five replicas needed for high availability
- Why not three?
  - Assume one datacenter down
  - Then one more machine crash => partial outage

Geography
- Replicas spread across the country to survive regional disasters
  - Up to 100ms apart

Performance
- Very high commit latency - 50-100ms
- Reads have extra network hops - 5-10ms
- High throughput - 100s of kQPS
Coping with High Latency

Preferred transaction structure
- One read phase: Avoid serial reads
  - Read in batches
  - Read asynchronously in parallel
- Buffer writes in client, send as one RPC

Use coarse schema and hierarchy
- Fewer tables and columns
- Fewer joins

For bulk operations
- Use small transactions in parallel - high throughput

Avoid ORMs that add hidden costs
Adjusting Clients

Typical MySQL ORM:
• Obscures database operations from app developers
• for loops doing one query per iteration
• Unwanted joins, loading unnecessary data

F1: ORM without the "R"
• Never uses relational joins
• All objects are loaded explicitly
  • Hierarchical schema and protocol buffers make this easy
  • Don't join - just load child objects with a range read
• Ask explicitly for parallel and async reads
Results: Development

- Code is slightly more complex
  - But predictable performance, scales well by default
- Developers happy
  - Simpler schema
  - Rich data types -> lower impedance mismatch
- One system for OLTP and OLAP workloads
  - No need for copies in bigtable
Results: Performance

User-Facing Latency
- Avg user action: ~200ms - on par with legacy system
- Flatter distribution of latencies

SQL Query Latency
- Similar or faster than MySQL
- More resources -> more parallelism -> faster
## No Compromise Storage

<table>
<thead>
<tr>
<th>Feature</th>
<th>Sharded MySQL</th>
<th>NoSQL (Bigtable)</th>
<th>F1 &amp; Spanner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistent reads and ACID</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>SQL Query</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>Schema mgmt.</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>Indexes</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>&quot;Infinite&quot; scaling</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>MapReduce</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>High Availability</td>
<td>Mostly</td>
<td></td>
<td>✔️</td>
</tr>
</tbody>
</table>
Summary

We moved a large and critical application suite from MySQL to F1.

This gave us
- Better scalability
- Better availability
- Strong consistency guarantees
- More scalable SQL query

And also similar application latency, using
- Coarser schema with rich column types
- Smarter client coding patterns

In short, we made our database scale, without giving up key database features along the way.
Questions...

Jeff Shute
jshute@google.com

Stephan Ellner
stephano@google.com