F1 - The Distributed SQL Database Supporting Google's Ad Business

Bart Samwel
bsamwel@google.com

ETH Zurich
Zurich, 25 October 2016
What's This Talk About?

- Part 1: Distributed databases and the challenges of "web scale"
- Part 2: F1, the distributed SQL database from Google
Distributed Databases and the challenges of "web scale"
Traditional Relational Databases

- A typical traditional database setup:
  - Relational (SQL) databases with ACID transactions
  - Master/slave replication
  - Scale by partitioning or "sharding" data over multiple databases
- Transaction = unit of database work.
  - Set of related reads + writes followed by a "commit".
- An ACID transaction is:
  - **Atomic:** All-or-nothing.
  - **Consistent:** Database moves from one consistent state to another.
  - **Isolated:** Transactions (seem to) happen one after the other.
  - **Durable:** After transaction commits, the data is safe.
Web Challenge #1: Scaling

- Web company data sets are **huge** and tend to **grow exponentially**¹
- Traditional SQL database model takes a lot of effort to scale: adding servers, repartitioning data. Not impossible, but hard.
- Move towards distributed database systems with equal peers
  - All servers created equal, dynamic partitioning
  - Workload can shift dynamically between servers
  - Add more servers = handle more data and more requests
- Rise of "NoSQL"
  - Non-relational data stores without full ACID transaction support.
  - Can be key/value, document stores or other non-relational formats.
  - Examples: Google Bigtable, Apache HBase, MongoDB, Amazon DynamoDB.

¹For companies that do well.
Web challenge #2: Multiple Datacenters

- Simple setups use master/slave replication even with NoSQL systems and across datacenter boundaries
  - Only one datacenter can commit writes
  - commit != fully replicated across datacenters. Too slow.
- Web companies want to run multiple symmetric data centers that each serve fraction of web requests.
- Multimaster replication:
  - Every datacenter can process write requests
  - Often eventually consistent
  - Prevent or resolve conflicts
- Example systems:
  - Amazon Dynamo (!= DynamoDB)
  - Apache Cassandra
More Observations

- Tradeoff between consistency and latency.
  - Latency matters. High page load time => users go somewhere else
- Availability is king.
  - If you aren't available then you can't serve your users / sell things
  - Continue operating even if data centers can't talk to each other
- Trade-offs result in:
  - Complexity for applications
  - "Interesting" failure recovery
Google F1
What is Google F1?

F1 - A Hybrid Database combining the
● Scalability of Bigtable
● Usability and functionality of SQL databases
Built on Spanner, Google's 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

- SOAP API
- web UI
- reports
- DB
- log aggregation
- spam analysis
- ad servers
- ad approvals
- ad logs
- ad-hoc SQL users

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 globally distributed storage system (OSDI, 2012)

Scalable: transparent sharding, data movement

Replication
- **Synchronous** cross-datacenter replication
  - Paxos protocol: majority of replicas must acknowledge
- Master/slave replication with consistent snapshot reads at slaves

ACID Transactions
- Standard 2-phase row-level locking
- Local or cross-machine (using two-phase commit, 2PC)
- Transactions serializable because of TrueTime (see paper)
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 history & notifications
Hierarchical Schema

Relational tables, with hierarchical clustering. Example:

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

**Rows and PKs**

- 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)

**Storage Layout**
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
 /   \
1,3   1,4
   /   /  \
1,3,5 1,3,6 1,4,7
```

**Storage Layout**

```
Customer (2)
  Campaign (2,5)
      AdGroup (2,5,8)
```

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

```
1,3
1,4
1,3,5
1,3,6
1,4,7
2,5
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 as 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

**SELECT** CustomerId, Whitelist
FROM Customer

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

**SELECT** CustomerId, Whitelist
FROM Customer

**SELECT** CustomerId, f.*
FROM Customer c
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 SQL Query Engine

- Fully functional SQL - joins, aggregations, indexes, etc.
- Highly parallel global scans
  - Complex plans: arbitrary joins, partitioning and shuffling, DAGs
  - In-memory and streaming whenever possible
- Local joining in Spanner when possible (hierarchical schema!)
- 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

```
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, less "foreign key chasing"

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>✔️ (per shard)</td>
<td></td>
<td>✔️ (global)</td>
</tr>
<tr>
<td>SQL Query</td>
<td>✔️ (per shard)</td>
<td></td>
<td>✔️ (global)</td>
</tr>
<tr>
<td>Schema mgmt.</td>
<td>✔️ (downtime required)</td>
<td></td>
<td>✔️ (nonblocking)</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>
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.
Internships at Google:
- Paid internships for Bachelor, Master, and PhD students
- Available in all major offices (London, Dublin, Zurich, Mountain View)
- Work on real Google products
- Information at google.com/about/careers/students

Careers at Google: google.com/about/careers
Research at Google: research.google.com
Bart Samwel: bsamwel@google.com