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

Jeff Shute, Mircea Oancea, Stephan Ellner,
Ben Handy, Eric Rollins, Bart Samwel,
Radek Vingralek, Chad Whipkey, Xin Chen,
Beat Jegerlehner, Kyle Littlefield, Phoenix Tong

SIGMOD
May 22, 2012

Today's Talk

F1 - A Hybrid Database combining the

- Scalability of Bigtable
- Usability and functionality of SQL databases

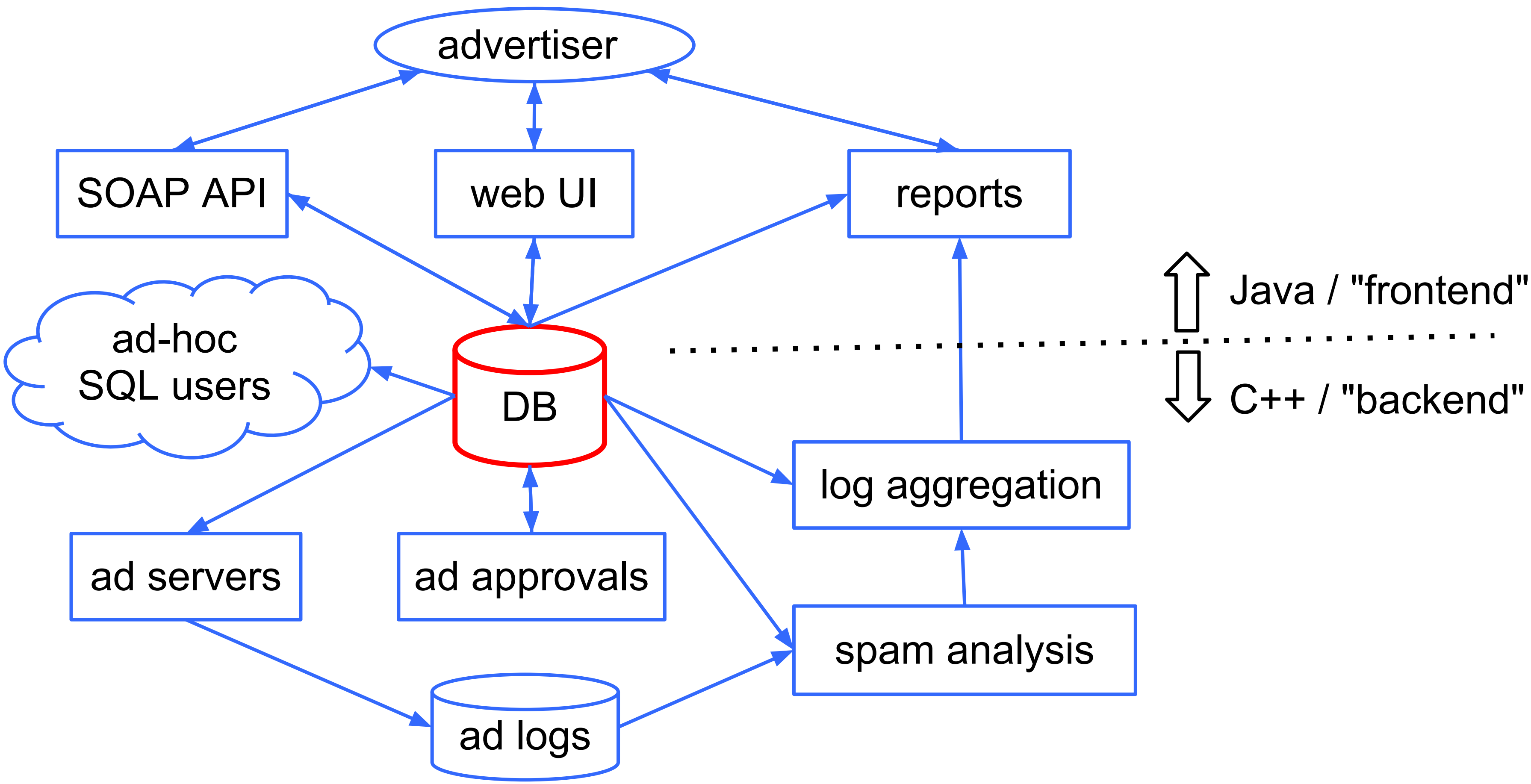
Key Ideas

- Scalability: Auto-sharded storage
- Availability & Consistency: Synchronous replication
- High commit latency: Can be hidden
  - 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



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 - multiple 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

Underlying Storage - Spanner

Descendant of Bigtable, Successor to Megastore

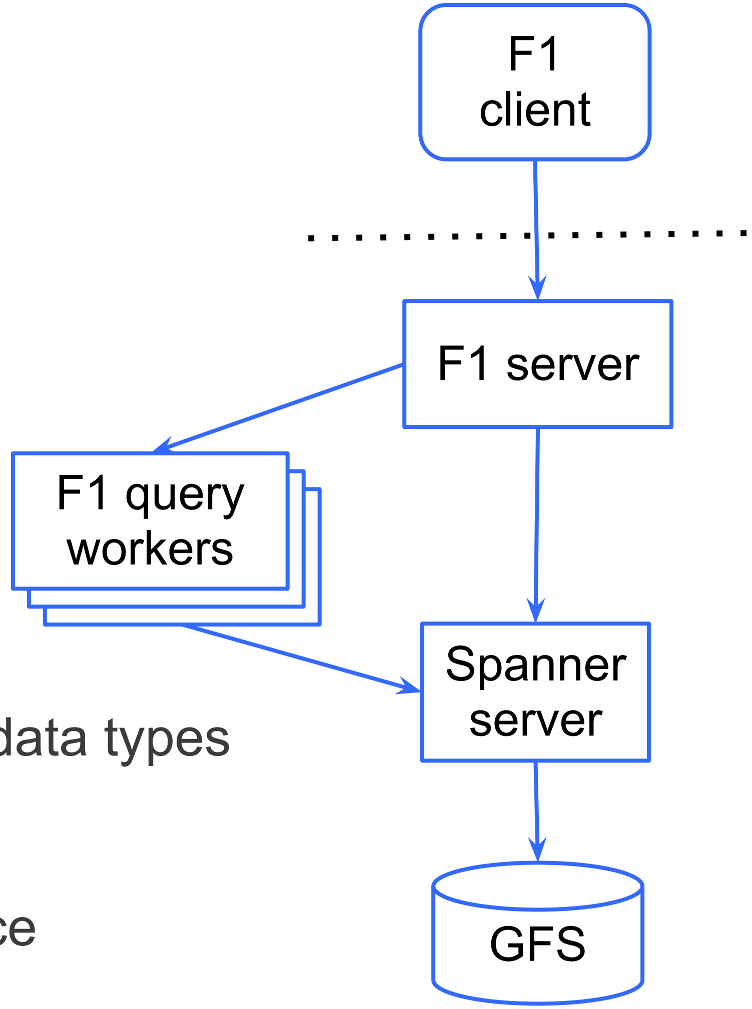
Properties

- Globally distributed
- Synchronous cross-datacenter replication (with Paxos)
- Transparent sharding, data movement
- General transactions
  - Multiple reads followed by a single atomic write
  - Local or cross-machine (using 2PC)
- Snapshot reads

F1

Architecture

- Sharded Spanner servers
  - data on GFS and in memory
- Stateless F1 server
- Pool of workers for query execution



Features

- Relational schema
  - Extensions for hierarchy and rich data types
  - Non-blocking schema changes
- Consistent indexes
- Parallel reads with SQL or Map-Reduce

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

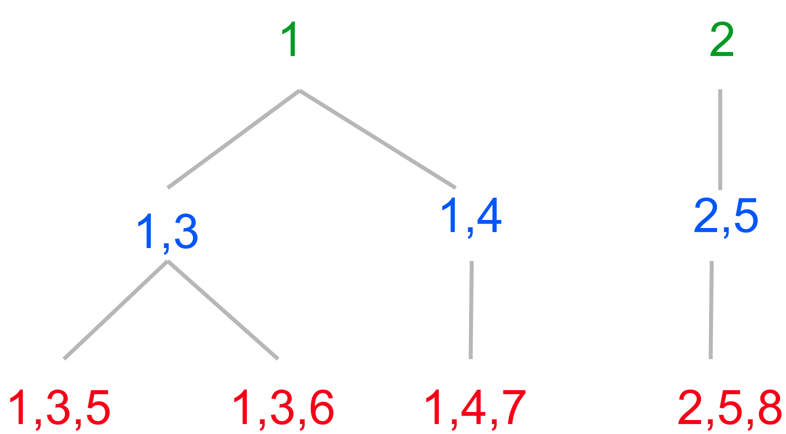
- Very high commit latency - 50-100ms
- Reads take 5-10ms - much slower than MySQL
- High throughput

Hierarchical Schema

Explicit table hierarchies. Example:

- Customer (root table): PK (CustomerId)
- Campaign (child): PK (CustomerId, CampaignId)
- AdGroup (child): PK (CustomerId, CampaignId, AdGroupId)

Rows and PKs



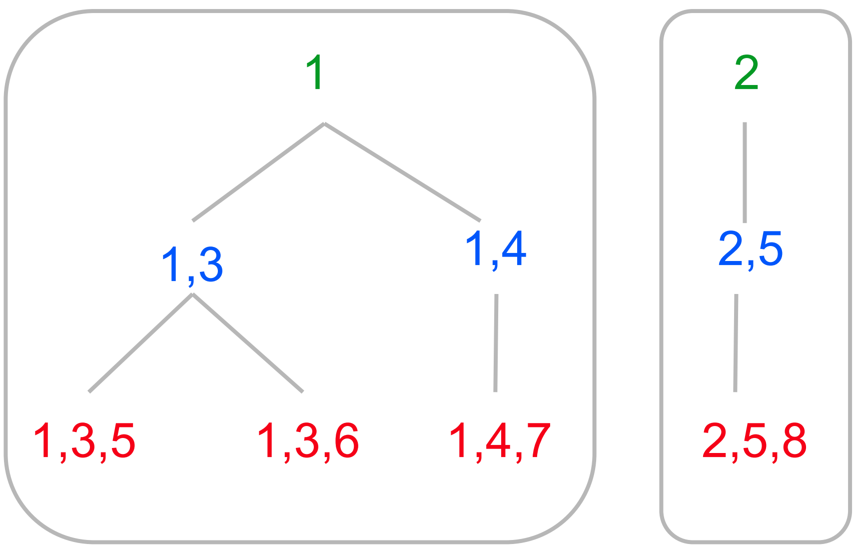
Storage Layout

```
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 clusters (can merge with no sorting)

Rows and PKs



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

- Treated as blobs by underlying storage
- 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 Query

Google™

- Parallel query engine implemented from scratch
- Fully functional SQL, joins to external sources
- Language extensions for protocol buffers

```
SELECT CustomerId
FROM Customer c PROTO JOIN c.Whitelist.feature f
WHERE f.feature_id = 302
  AND f.status = 'STATUS_ENABLED'
```

Making queries fast

- Hide RPC latency
- Parallel and batch execution
- Hierarchical joins

Coping with High Latency

Preferred transaction structure

- One read phase: No 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

ORM Anti-Patterns

- Obscuring database operations from app developers
- Serial reads
  - `for` loops doing one query per iteration
- Implicit traversal
  - Adding unwanted joins and loading unnecessary data

These hurt performance in all databases.

They are disastrous on F1.

Our Client Library

- Very lightweight ORM - doesn't really have the "R"
  - Never uses Relational joins or traversal
- 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

Google™

Development

- Code is slightly more complex
  - But predictable performance, scales well by default
- Developers happy
  - Simpler schema
  - Rich data types -> lower impedance mismatch

User-Facing Latency

- Avg user action: ~200ms - on par with legacy system
- Flatter distribution of latencies
  - Mostly from better client code
  - Few user actions take much longer than average
  - Old system had severe latency tail of multi-second transactions

Current Challenges

- Parallel query execution
  - Failure recovery
  - Isolation
  - Skew and stragglers
  - Optimization

- Migrating applications, without downtime
  - Core systems already on F1, many more moving
  - Millions of LOC

Summary

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

This gave us

- Better scalability
- Better availability
- Equivalent consistency guarantees
- Equally powerful 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, and didn't lose any key database features along the way.