F1: A Distributed SQL Database That Scales

Presentation by:
Alex Degtiar (adegtiar@cmu.edu)
15-799
10/21/2013
What is F1?

• Distributed relational database
• Built to replace sharded MySQL back-end of AdWords system
• Combines features of NoSQL and SQL
• Built on top of Spanner
Goals

• Scalability
• Availability
• Consistency
• Usability
Features Inherited From Spanner

- Scalable data storage, resharding, and rebalancing
- Synchronous replication
- Strong consistency & ordering
New Features Introduced

- Distributed SQL queries, including joining data from external data sources
- Transactionally consistent secondary indexes
- Asynchronous schema changes including database reorganizations
- Optimistic transactions
- Automatic change history recording and publishing
Architecture
Architecture - F1 Client

- Client library
- Initiates reads/writes/transactions
- Sends requests to F1 servers
Architecture - F1 Server

- Coordinates query execution
- Reads and writes data from remote sources
- Communicates with Spanner servers
- Can be quickly added/removed
Architecture

Diagram showing the architecture with F1 Client, Load Balancer, F1 Master, F1 Server, Slave Pool, Spanner, and CFS components connected in a network.
Architecture - F1 Slaves

- Pool of slave worker tasks
- Processes execute parts of distributed query coordinated by F1 servers
- Can also be quickly added/removed
Architecture - F1 Master

- Maintains slave membership pool
- Monitors slave health
- Distributes list membership list to F1 servers
Architecture - Spanner Servers

- Hold actual data
- Re-distribute data when servers added
- Support MapReduce interaction
- Communicates with CFS
Data Model

- Relational schema (similar to RDBMS)
- Tables can be organized into a hierarchy
- Child table clustered/interleaved within the rows from its parent table
  - Child has foreign key as prefix of p-key
## Data Model

### Traditional Relational

- **Customer** *(CustomerID, ...)*
- **Campaign** *(CampaignID, CustomerID, ...)*
- **AdGroup** *(AdGroupID, CampaignID, ...)*

Foreign key references only the parent record.

Joining related data often requires reads spanning multiple machines.

<table>
<thead>
<tr>
<th>Customer(1,...)</th>
<th>AdGroup(6,3,...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer(2,...)</td>
<td>AdGroup(7,3,...)</td>
</tr>
<tr>
<td>Campaign(3,1,...)</td>
<td>AdGroup(8,4,...)</td>
</tr>
<tr>
<td>Campaign(4,1,...)</td>
<td>AdGroup(9,5,...)</td>
</tr>
</tbody>
</table>

### Clustered Hierarchical

- **Customer** *(CustomerID, ...)*
- **Campaign** *(CustomerID, CampaignID, ...)*
- **AdGroup** *(CustomerID, CampaignID, AdGroupID, ...)*

Primary key includes foreign keys that reference all ancestor rows.

Related data is clustered for fast common-case join processing.

<table>
<thead>
<tr>
<th>Customer(1,...)</th>
<th>Campaign(1,3,...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdGroup (1,3,6,...)</td>
<td>AdGroup(1,3,7,...)</td>
</tr>
<tr>
<td>AdGroup (1,4,...)</td>
<td>Campaign(1,4,...)</td>
</tr>
<tr>
<td>AdGroup (1,4,8,...)</td>
<td>AdGroup(2,5,...)</td>
</tr>
</tbody>
</table>

Physical data partition boundaries occur between root rows.
Secondary Indexes

- Transactional & fully consistent
- Stored as separate tables in Spanner
- Keyed by index key + index table p-key
- Two types: Local and Global
Local Secondary Indexes

- Contain root row p-key as prefix
- Stored in same spanner directory as root row
- Adds little additional cost to a transaction
Global Secondary Indexes

- Does not contain root row p-key as prefix
- Not co-located with root row
  - Often sharded across many directories and servers
- Can have large update costs
- Consistently updated via 2PC
Schema Changes - Challenges

- F1 massively and widely distributed
- Each F1 server has schema in memory
- Queries & transactions must continue on all tables
- System availability must not be impacted during schema change
Schema Changes

- Applied asynchronously
- Issue: concurrent updates from different schemas
- Solution:
  - Limiting to one active schema change at a time (lease on schema)
  - Subdivide schema changes into phases
    - Each consecutively mutually compatible
Transactions

- Full transactional consistency
- Consists of multiple reads, optionally followed by a single write
- Flexible locking granularity
Transactions - Types

• Read-only: fixed snapshot timestamp
• Pessimistic: Use Spanner’s lock transactions
• Optimistic:
  o Read phase (Client collects timestamps)
  o Pass to F1 server for commit
  o Short pessimistic transaction (read + write)
    ▪ Abort if conflicting timestamp
    ▪ Write to commit if no conflicts
Optimistic Transactions: Pros and Cons

Pros
• Tolerates misbehaving clients
• Support for longer transactions
• Server-side retryability
• Server failover
• Speculative writes

Cons
• Phantom inserts
• Low throughput under high contention
Change History

- Supports tracking changes by default
- Each transaction creates a change record
- Useful for:
  - Pub-sub for change notifications
  - Caching
Client Design

- MySQL-based ORM incompatible with F1
- New simplified ORM
  - No joins or implicit traversals
  - Object loading is explicit
  - API promotes parallel/async reads
  - Reduces latency variability
Client Design

- NoSQL interface
  - Batched row retrieval
  - Often simpler than SQL
- SQL interface
  - Full-fledged
  - Small OLTP, large OLAP, etc
  - Joins to external data sources
Query Processing

- Centrally executed or distributed
- Batching/parallelism mitigates latency
- Many hash re-partitioning steps
- Stream to later operators ASAP for pipelining
- Optimized hierarchically clustered tables
- PB-valued columns: structured data types
- Spanner’s snapshot consistency model provides globally consistent results
SELECT agcr.CampaignId, click.Region, cr.Language, SUM(click.Clicks)
FROM AdClick click
JOIN AdGroupCreative agcr
    USING (AdGroupId, CreativeId)
JOIN Creative cr
    USING (CustomerId, CreativeId)
WHERE click.Date = '2013-03-23'
GROUP BY agcr.CampaignId, click.Region, cr.Language
Query Processing Example

- Scan of AdClick table
- Lookup join operator (SI)
- Repartitioned by hash
- Distributed hash join
- Repartitioned by hash
- Aggregated by group
Distributed Execution

- Query splits into plan parts => DAG
- F1 server: query coordinator/root node and aggregator/sorter/filter
- Efficiently re-partitions the data
  - Can’t co-partition
  - Hash partitioning BW: network hardware
- Operate in memory as much as possible
- Hierarchical table joins efficient on child table
- Protocol buffers utilized to provide types
Evaluation - Deployment

- AdWords: 5 data centers across US
- Spanner: 5-way Paxos replication
- Read-only replicas
Evaluation - Performance

- 5-10ms reads, 50-150ms commits
- Network latency between DCs
  - Round trip from leader to two nearest replicas
  - 2PC
- 200ms average latency for interactive application - similar to previous
- Better tail latencies
- Throughput optimized for non-interactive apps (parallel/batch)
  - 500 transactions per second
Issues and Future work

- High commit latency
- Only AdWords deployment show to work well - no general results
- Highly resource-intensive (CPU, network)
- Strong reliance on network hardware
- Architecture prevents co-partitioning processing and data
Conclusion

- More powerful alternative to NoSQL
- Keep conveniences like SI, SQL, transactions, ACID but gain scalability and availability
- Higher commit latency
- Good throughput and worst-case latencies
References

• Information, figures, etc.: J. Shute, et al., F1: A Distributed SQL Database That Scales, VLDB, 2013.

• High-level summary: http://highscalability.com/blog/2013/10/8/f1-and-spanner-holistically-compared.html