Spanner & F1

Iztok Savnik, FAMNIT

Januar, 2024.

Literature

James C. Corbett et al., Spanner: Google’s Globally-Distributed Database, OSDI, 2012.

Jeff Shute et al., F1: A Distributed SQL Database That Scales, VLDB, 2013.

Robert Morris, Lecture: Spanner, MIT 6.824, Distributed Systems, 2020.

Outline

- Spanner
- Distributed database system F1

Spanner

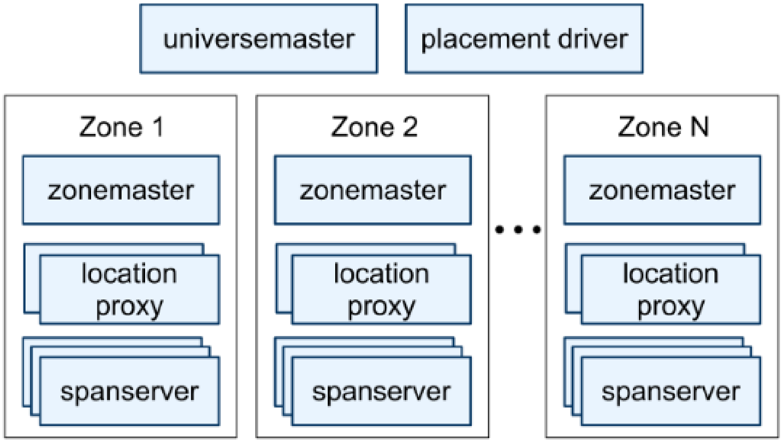
Introduction

- Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database.
  - First system to distribute data at global scale and
  - Support externally-consistent distributed transactions.
- A novel time API that exposes clock uncertainty is critical to provide:
  - External consistency
    - If T1 commits before T2 starts, then ts(T1)$<$ts(T2), and T2 must see T1's writes, globally.
  - Non-blocking reads in the past,
  - Lock-free read-only transactions, and
  - Atomic schema changes.

Introduction

- Shards data across many sets of Paxos state machines in data-centers spread globally.
  - Replication is used for global availability and geographic locality;
    - Clients automatically failover between replicas.
    - Managing cross-datacenter replication is main focus.
  - Spanner automatically:
    - Reshards data across machines on the changed amount of data or number of servers.
    - Migrates data across machines to balance load and in response to failures.
  - Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database.

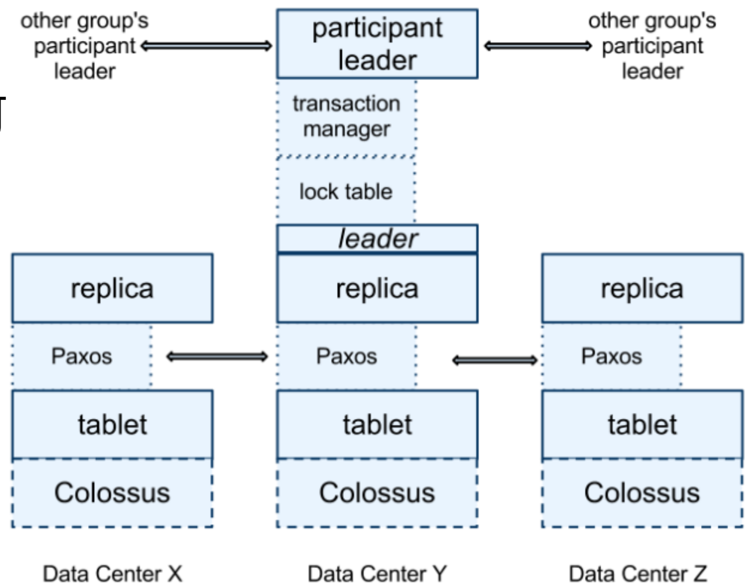
Implementation



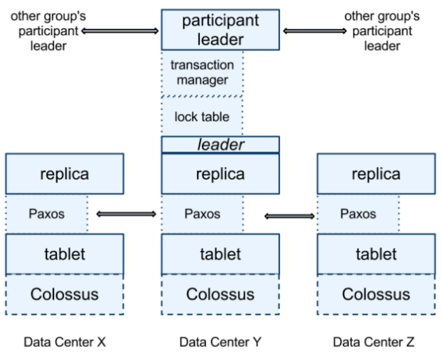
- Spanner deployment is called a *universe* (there are only a few universes)
- Spanner is organized as a set of *zones.*
  - Unit of physical isolation: one or more zones in a data center (DC).
    - Analog of a deployment of Bigtable servers.
  - Unit of administrative deployment.
  - Locations across which data can be replicated.
  - 1 zonemaster – [100,1000*n] spanservers, n~10
    - zonemaster assigns data to spanservers;
    - spanservers serve data to clients.
  - *Universemaster*: console displaying status of zones; debugging.
  - *Placement driver*: automated movement of data across zones on the timescale of minutes.

Spanserver Software Stack

- How replication and distributed txns are layered?
  - Onto BigTable-based storage manager.
- Each sserver responsible for 100-1000 tablets
- Tablet = A bag of mappings:
  - (key:string,TS:int64)→string
  - Similar to BigTable tablet
  - Multi-version database (not KV)
  - Tablet stores
    - B-tree-like files and a WAL (log)
- For replication, each sserver
  - Implements single Paxos state machine on each tablet

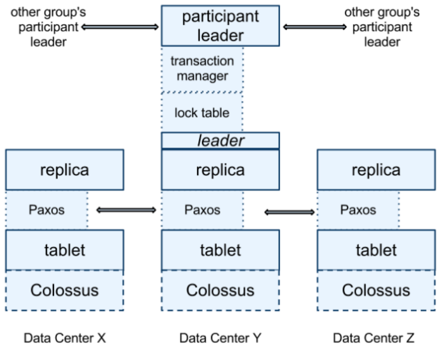


Spanserver Software Stack



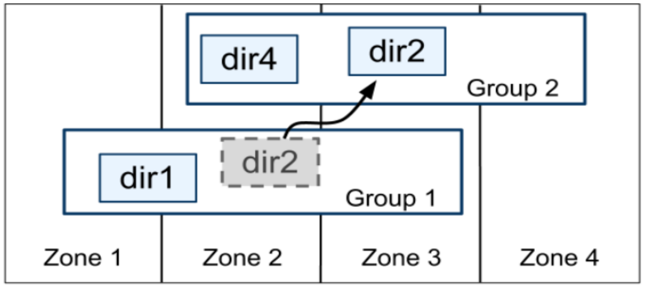
- Paxos implementation:
  - Each state machine stores its metadata and log in its corresp. tablet.
  - Long-lived leaders with time-based leader leases (10s)
  - Logs every Paxos write twice: tablet's and Paxos log
  - Writes to slave replicas are applied by Paxos in a timestamp order
- Paxos implements <u>consistently replicated bag of mappings</u>
  - KV mapping state of ∀ replica is stored in corresponding tablet.
  - Writes must initiate the Paxos protocol at the *<u>participant leader</u>*.
    - Other participants are slaves.
  - Reads access state directly from the tablet at any replica.
  - Set of replicas is collectively a *<u>Paxos group</u>*.

Spanserver Software Stack



- A leader spanserver
  - Uses a *<u>lock table</u>* to implement concurrency control.
  - Implem. a <u>transaction manager</u> to support distributed txns.
    - Distributed, eager replication (see lecture on Replication)
  - If txn involves only one Paxos group, it can bypass TM.
    - Lock tables provide transactionality
  - If txn involves more than one Paxos group
    - Groups' leaders coordinate to perform 2PC
    - One of the participant groups is chosen as *<u>coordinator leader</u>.*
    - Slaves in that group are called *<u>coordinator slaves</u>*.

Directories and Placement



- Bucketing abstraction called a <u>*directory*</u>
  - Set of contiguous keys that share common prefix (~50MB).
  - Directories allow apps to control locality of their data
    - Choosing the keys carefully.
    - Keys in directory have the same prefix (see Data model)
- A directory is the unit of *data placement*.
  - Data in directory has the same replication configuration.
  - Placement of dirs can be specified by an application.
    - Placement-specification language allows admin to specify
      - the number and types of replicas, and
      - the geographic placement of those replicas.

Directories and Placement

- A <u>Paxos group</u> is a set of directories.
  - Directories in PG are often accessed together.
  - This is how we obtain locality of data.
  - Movement between Paxos groups is in directories
    - To shed load from Paxos group; to put dirs frequently accessed together into the same group; or to move a directory into a group that is closer to its ancessors.
    - Movedir moves the data in the background
      - Only last part is moved in txn (to update metadata).
- <u>Spanner tablet</u> is different from BigT tablet
  - Can include different ranges (directories) of KV pairs.
  - Colocate multiple dirs that are freq accessed together.

Spanner Data Model

- What Spanner exposes to applications?
- Semi-relational tables & syncronous replication
  - Lead by the popularity of Megastore (300 apps but low performance)
    - Megastore apps: Gmail, Picasa, Calendar, Android Market, and AppEngine
  - Simpler data model & support for sync replication across DC
- SQL-like query language
  - The need to include a SQL-like query language supported by popularity of Dremel (an interactive data-analysis tool)
- General-purpose transactions.
  - Lead by lack of cross-row transactions in BigT.
    - 2PC too expensive? Performance or availability problems?
  - Better that apps programmers deal with performance problems.
    - Use snapshot read, careful organization of data, overuse of txns, ...
  - Running 2PC over Paxos mitigates the availability problems.

Spanner Data Model

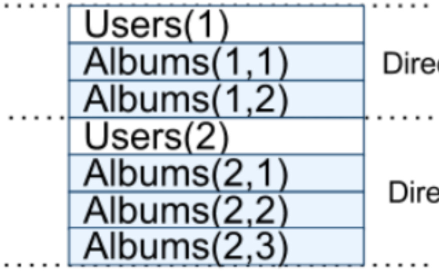
- Spanner's data model is semi-relational
  - Every row is named with ordered set of primary-key columns.
  - A relation is a mapping from PK columns to non-key clmns.
  - This is where Spanner still looks like a key-value store.

Spanner Data Model

- Example schema:
  - Photo metadata on per-user, per-album basis.
    - Schema language is similar to Megastore's.
  - Every database must be partitioned by clients into one or more hierarchies of tables.
    - INTERLEAVE IN
    - ON DELETE CASCADE
  - This allows clients to describe the locality relationships that exist between multiple tables.
    - Necessary for good performance in a sharded, distributed database.

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```



TrueTime

Method	Returns
$TT.now()$	$TTinterval$: $[earliest, latest]$
$TT.after(t)$	true if t has definitely passed
$TT.before(t)$	true if t has definitely not arrived

- TrueTime represents time as a *TTinterval*
  - Interval with bounded time uncertainty!
    - Endpoints of a *TTinterval* are of type *TTstamp.*
  - Define the instantaneous error bound as ε.
    - Half of the *TTinterval* width; the average error bound as $\bar{\varepsilon}$.
  - **<u>Guaranteed</u>:**
    - $tt = TT.now() \Rightarrow tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest$
  - Time references: GPS and atomic clocks.
    - Synchronisation among clocks every 30s
    - ε is usually a sawtooth function of time: it varies from 1ms to 7ms; $\bar{\varepsilon}$ is about 4ms (sawtooth bounds).
    - Drift rate is set at 200 μs/s (micros).

Concurrency Control

- TrueTime is used to guarantee the correctness properties in concurrency control.
- Those properties are used to implement features:
  1) externally consistent transactions,
  2) lock-free read-only transactions, and
  3) non-blocking reads in the past.
- We will distinguish writes as seen by
  1) Paxos, from
  2) Spanner client writes.

Timestamp Management

Operation
Read-Write Transaction
Read-Only Transaction
Snapshot Read, client-provided timestamp
Snapshot Read, client-provided bound

- Read/Write transaction
  - Uses Paxos and 2PC
- Read-only transaction
  has performance benefits of snapshot isolation
  - It must be predeclared as not having any writes.
  - Reads execute without locking, at a system-chosen timestamp, so that incoming writes are not blocked.
- Snapshot read is a read in the past
  - Executes without locking.
  - A client specifies a timestamp, or provide an upper bound on TS's staleness.
  - Read proceeds at any replica that is sufficiently up-to-date.

Paxos Leader Leases

- Paxos uses timed leases to make leadership long-lived (10s)
- Potential leader sends requests for timed lease votes.
  - When receiving a quorum of votes, leader has a lease.
  - Lease is extended on a successful write; and, leader requests lease extensions if near expiration.
- Leader's <u>lease interval</u>
  - Starting when it discovers it has a quorum of lease votes, and
  - Ending when it no longer has a quorum of lease votes

Paxos Leader Leases

- Spanner depends on disjointness invariant:
  - For each Paxos group, each Paxos leader's lease interval is disjoint from every other leader's.
- Paxos implementation allows leader to abdicate
  - Leader releases slaves from its lease
- Spanner constrains when abdication is permissible.
  - Leader must wait until $TT.after(s_{max})$ = true.

Assigning TS to RW Transactions

- Transactional reads and writes use two-phase locking.
  - TS can be assigned after all locks acquired, but before any locks have been released.
  - Spanner assigns TS to txn that Paxos assigns to the Paxos write for the txn commit.
- Spanner depends on the monotonicity invariant:
  - Within each Paxos group, Spanner assigns TS to Paxos writes in monotonically increasing order, even across leaders.
  - This invariant is enforced across leaders by making use of the *disjointness invariant*:
    - Leader must only assign TS within the interval of leader’s lease.

Assigning TS to RW Transactions

- <u>External-consistency invariant</u>:
  - If the start of T_2 occurs after the commit of T_1, then the commit TS of T_2 must be greater than the commit TS of T_1 .
    - $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \Rightarrow s_1 < s_2$, s_1=TS(T1), s_2=TS(T2), e_i event of T_i
- Commit request at the coordinator leader (abbr. CL)
  - Arrival of commit request for a write T_i is the event e_i^{server}.
  - **start** rule: CL for a write T_i assigns a commit TS s_i no less than the value of TT.now().latest, computed after e_i^{server}
    - "start" rule ensures: $t_{abs}(e_i^{server}) < s_i$.
  - **commit wait** rule: CL ensures that clients cannot see any data committed by T_i until TT.after(s_i) is true.
    - "commit wait" rule ensures: $s_i < t_{abs}(e_i^{commit})$.

Assigning TS to RW Transactions

- Proof of the external consistency invariant.
  - $s_1 = TS(T_1)$, $s_2 = TS(T_2)$
  - $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \Rightarrow s_1 < s_2$

$$s_1 < t_{abs}(e_1^{commit}) \qquad \text{(commit wait)}$$
$$t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \qquad \text{(assumption)}$$
$$t_{abs}(e_2^{start}) \leq t_{abs}(e_2^{server}) \qquad \text{(causality)}$$
$$t_{abs}(e_2^{server}) \leq s_2 \qquad \text{(start)}$$
$$s_1 < s_2 \qquad \text{(transitivity)}$$

Serving Reads at a Timestamp

- Is replica's state sufficiently up-to-date to read?
  - To determine this Spanner uses *monotononicity invariant*.
  - Every replica tracks a value at **t_{safe} = max TS up-to-date**.
- Replica can satisfy a read at a timestamp t if t <= t_{safe}.
  - Define $t_{safe} = \min(t_{safe}^{Paxos}, t_{safe}^{TM})$
- t_{safe} for Paxos
  - T_{safe}^{Paxos} = TS of highest-applied Paxos write
  - *Judgement:*

    TS-s increase monotonically + Writes applied in order

    $\Longrightarrow$

    Writes will no longer occur at or below T_{safe}^{Paxos}.

Serving Reads at a Timestamp

- t_{safe} for TM.
  - $T_{safe}^{TM} = \infty$
    - IF there are no prepared, but not commited txns (between 2P of 2PC)
    - This means any TS can be used (also current time)
  - $T_{safe}^{TM} = \min_i (s_{i,g}^{prepare})-1$
    - IF there are any prepared txns that are not commited.
    - State affected by prepared txns is indeterminate.
      - It is not known if txns will commit.
    - Every coordinator leader (for a group g) for a txn T_i assigns a prepare TS $s_{i,g}^{prepare}$ to its prepare record
      - Coordinator leader ensures: Commit TS $s_i \geq s_{i,g}^{prepare}$ for all g.
    - Therefore, for every replica in a group g, over all transactions T_i prepared at g, $T_{safe}^{TM} = \min_i (s_{i,g}^{prepare})-1$ over all transactions prepared at g.

Assigning TS to RO Transactions

- A read-only txn executes in two phases:
  - Assign a timestamp s_{read} to txn, and
  - Execute the txn's reads as snapshot reads at s_{read}.
    - Snapshot reads execute at any replicas sufficiently up-to-date.
- Simple assignment of s_{read} = TT.now().latest
  - Assign at <u>any time after a transaction starts</u>.
  - Preserves external consistency by an argument analogous to that presented for writes.
  - Txn may block at s_{read}, if t_{safe} has not advanced sufficiently.
  - To reduce the chances of blocking, Spanner should assign the oldest TS that preserves external consistency.

Distributed database system F1

Distributed DBMS F1

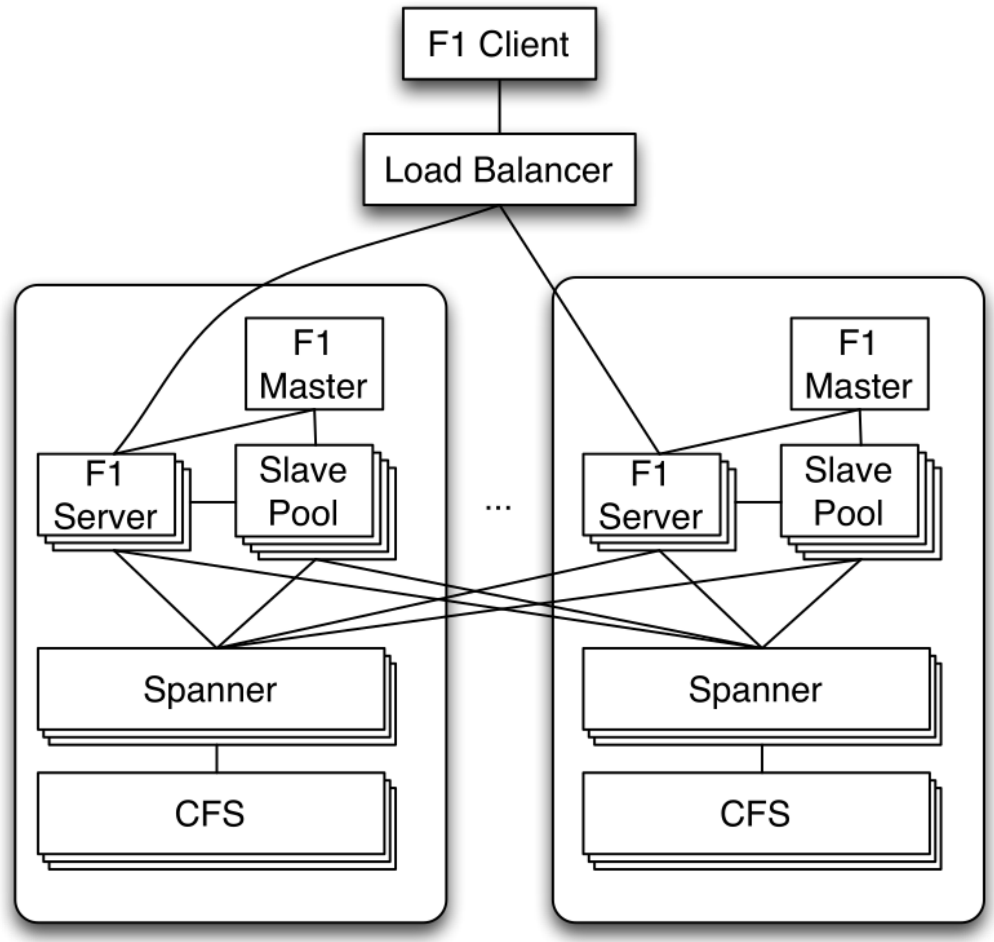
- F1 is a distributed relational database system
  - Supports the AdWords business
  - Filial 1 hybrid: cross mating NoSQL and RDBMS systems
    - Scalability of NoSQL systems like Bigtable
    - Consistency and usability of traditional SQL databases
- The key goals of F1's design are
  - Scalability, availabiity, consistentcy and usability
  - These design goals were considered mutually exclusive
- F1 inherits from Spanner
  - Extremely scalable data storage
  - Synchronous cross-datacenter replication
  - Strong consistency

Distributed DBMS F1

- Features of F1
  - Extremely scalable data storage (from Spanner, Bigtable)
  - Synchronous replication using 2PC and Paxos
    - Implies higher commit latency
  - Hierarchical schema model with structured data types
    - F1 schema makes data clustering explicit
    - Asynchronous schema changes
  - Distributed SQL query engine
    - Asynchronous reads, Optimistic transactions, Transactionally consistent secondary indexes
  - Automatic change tracking and publishing
- Experiences with AdWords
  - 100s of apps, 1000s users (in Ads), 100TB DB,
  - Availability 99.999%, latency same as in old MySQL

Basic architecture

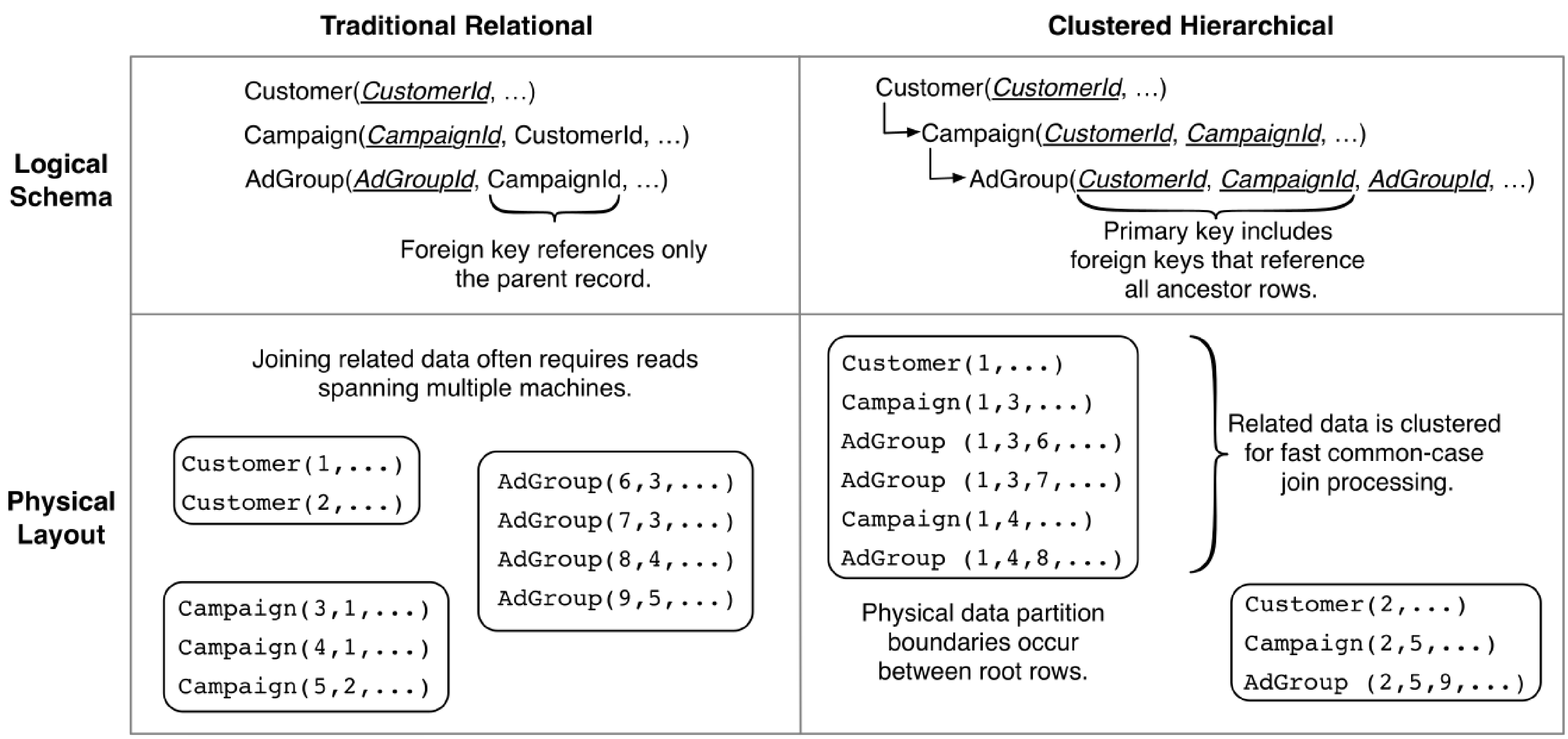
- Interfaces: F1 client library
- Avoid unnecessarily increasing request latency
  - Requests may transparently go to the remote Dcs
- F1, Spanner collocated in DC
  - F1 servers can also use Spanner servers in remote Dcs
- Sservers access data in CFS
  - Never from remote DC
- F1 srvrs are mostly stateless
  - Except for 2PC txn lock table
  - Can be added, removed quickly
- Shared slave pool
  - Runs parts of query plans (F1 processes)



Data model

- F1 data model is very similar to the Spanner data model.
  - Spanner later adopted F1's data model
  - Relational schemas similar to that of a traditional RDBMS
    - Extensions: explicit table hierarchy and columns with Protocol Buffer DTyps
- Logically, tables in F1 schema can be organized into hierarchy.
  - Physically, child tables clustered with rows from its parent table
  - Child table includes FK to parent table as a prefix of its primary key
    - Stored in single Spanner directory (single Spanner server) so we get fast localized queries.
    - Advantages: fast joins between tables in hierarchy, reduces the number of Spanner groups involved in txn, fast single root txns usually avoid 2PC.
  - Hirarchies are not necessary; we get flat relational model
- Protocol buffers (columns with structured data types; one Span. blob)
  - PBs include typed fields; Fields can also be nested Protocol Buffers
  - Many tables in an F1 schema consist of a single Protocol Buffer column.
    - At Google, PBs are ubiquitous for data storage and interchange between apps.

Data model



Traditional Relational
Clustered Hierarchical
Logical Schema
Customer(CustomerId, …)
Campaign(CampaignId, CustomerId, …)
AdGroup(AdGroupId, CampaignId, …)
Foreign key references only the parent record.
Customer(CustomerId, …)
Campaign(CustomerId, CampaignId, …)
AdGroup(CustomerId, CampaignId, AdGroupId, …)
Primary key includes foreign keys that reference all ancestor rows.
Physical Layout
Joining related data often requires reads spanning multiple machines.
Customer(1,...)
Customer(2,...)
AdGroup(6,3,...)
AdGroup(7,3,...)
AdGroup(8,4,...)
AdGroup(9,5,...)
Campaign(3,1,...)
Campaign(4,1,...)
Campaign(5,2,...)
Customer(1,...)
Campaign(1,3,...)
AdGroup (1,3,6,...)
AdGroup (1,3,7,...)
Campaign(1,4,...)
AdGroup (1,4,8,...)
Related data is clustered for fast common-case join processing.
Physical data partition boundaries occur between root rows.
Customer(2,...)
Campaign(2,5,...)
AdGroup (2,5,9,...)

Transactions

- Experiences with eventual consistency systems
  - Developers spend a lot of time building complex and error-prone mechanisms to cope with eventual consistency
  - Google: Consistency problems should be solved at database level.
- F1 txns consists of multiple reads, optionally followed by a single write
  - Three types of txns built on top of Spanner
  - 1) Snapshot, 2) Pessimistic and 3) Optimistic transactions
- Snapshot txns
  - RO txns with snapshot semantics; read as of a fixed Spanner TS
  - Read from a local Spanner replica; with specific TS or current TS
    - Later option: a txn may wait until concurrent txns commit
  - No access to remote DCs!
  - Default mode for SQL queries and for MapReduces

Transactions

- Pessimistic txns
  - These txns map directly on to Spanner txns
  - Centralized + Eager replication (2PC + Paxos)
  - Stateful communications protocol that requires holding locks
  - All requests get directed to the same F1 server
- Optimistic txns
  - Read phase (arbitrarily long + no locks!), then a short write phase
  - To detect row-level conflicts, rows are returned including latest TS
    - New commit TS is written into lock column on row update
    - Client library collects these TSs and passes them back to an F1 server with write that commits txn
    - F1 server creates a short-lived Spanner pessimistic transaction and re-reads the last modification TS for all read rows
    - If any re-read TSs differ from client's, F1 aborts the txn
    - Otherwise, writes are sent to Spanner to commit txn

Transactions

- Optimistic txns (cont.)
  - F1 clients use optimistic transactions by default
  - Clients are involved more intensively in txns
  - Advantages
    - Tolerating misbehaved clients (no locks; R do not conflict W; abandoned txns)
    - Long-lasting txns (pessimistic txns aborted while single-stepping)
    - Easier server-side retriability (self-contained txns; server has all data)
    - F1 server failover (client can send txn to other server)
    - Speculative writes (read data outside txn; no interference => spec. writes)
  - Disadvantages
    - Insertion phantoms (we have TSs of rows read at the beginning of txn)
    - Low throughput under high contention (many updates result in abort)
  - F1 users can mix optimistic and pessimistic transactions arbitrarily and still preserve ACID semantics.
    - Inventive use of txns to speed-up applications

Query processing

- Key properties of F1 SQL query processing system
  - Queries are executed either as (1) low-latency centrally executed queries or (2) distributed queries with high parallelism.
  - All data is remote and batching is used to mitigate network latency.
  - All data is arbitrarily partitioned; few useful ordering properties.
  - Queries use many hash-based repartitioning steps.
  - Query trees comprise operators that stream data to later operators (as soon as possible) to maximize pipelining.
  - Hierarchically clustered tables have optimized access methods.
  - Query data can be consumed in parallel.
  - PB-valued columns provide first-class support for structured DTs.
  - Spanner’s snapshot consistency model provides globally consistent results.

Centralized and Distributed Queries

- F1 SQL supports both centralized and distributed execution of queries.
  - Centralized execution is used for short OLTP-style queries and the entire query runs on one F1 server node.
  - Distributed execution is used for OLAP-style queries and spreads the query workload over worker tasks in the F1 slave pool
    - Distributed queries always use snapshot transactions.
  - The query optimizer uses heuristics to determine which execution mode is appropriate for a given query.

Distributed Query Example

- `AdGroup` = collection of ads with some shared configuration.
- `AdClick` = records the `Creative` that the user was shown and the `AdGroup` from which the `Creative` was chosen.
- `Creative` = actual ad text.
  - `Creatives` can be shared by multiple `AdGroups`.
- `AdGroupCreative` = link table between AdGroup and `Creatives`.

```
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
```

Distributed query example

A possible query plan for this query.