CS4224/CS5424 Lecture 11
Google Spanner

Background

- Bigtable [ODSI 2006]
  - NoSQL data store
- Megastore [SIGMOD 2008]
  - Limited transactional NoSQL DBMS
  - Used in: Gmail, Picasa, Calendar, Android Market, AppEngine
- Spanner [OSDI 2012]
  - Transactional NoSQL data store
- F1 [SIGMOD 2012, VLDB 2013]
  - Loosely coupled RDBMS
  - Used in: AdWords
- Spanner [SIGMOD 2017]
  - Tightly coupled RDBMS
  - Part of Google Cloud Platform (Cloud Spanner)

Lessons from Bigtable & Megastore

Even though many projects happily use ***Bigtable****, we have also consistently received complaints from users that* ***Bigtable*** *can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication.*

Lessons from Bigtable & Megastore (cont.)

Spanner *exposes the following set of data features to applications: a data model based on schematized semi-relational tables, a query language, and general purpose transactions. The move towards supporting these features was driven by many factors.*

Lessons from Bigtable & Megastore (cont.)

The need to support schematized semi-relational tables and synchronous replication is supported by the popularity of **Megastore***. At least 300 applications within Google use* **Megastore** *(despite its relatively low performance) because its data model is simpler to manage than* **Bigtable***'s, and because of its support for synchronous replication across datacenters. (***Bigtable** *only supports eventually-consistent replication across datacenters.)*

Lessons from Bigtable & Megastore (cont.)

The need to support a SQL like query language in **Spanner** *was also clear, given the popularity of Dremel as an interactive data analysis tool. Finally, the lack of cross-row transactions in* **Bigtable** *led to frequent complaints; ...*

Lessons from Bigtable & Megastore (cont.)

Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Running two-phase commit over Paxos mitigates the availability problems.

Lessons from Bigtable & Megastore (cont.)

We also have a lot of experience with eventual consistency systems at Google. In all such systems, developers spend a significant fraction of their time building extremely complex and error-prone mechanisms to cope with eventual consistency and handle data that may be out of date. We think this is an unacceptable burden to place on developers and that consistency problems should be solved at the database level. Full transactional consistency is one of the most important properties of **F1***.*

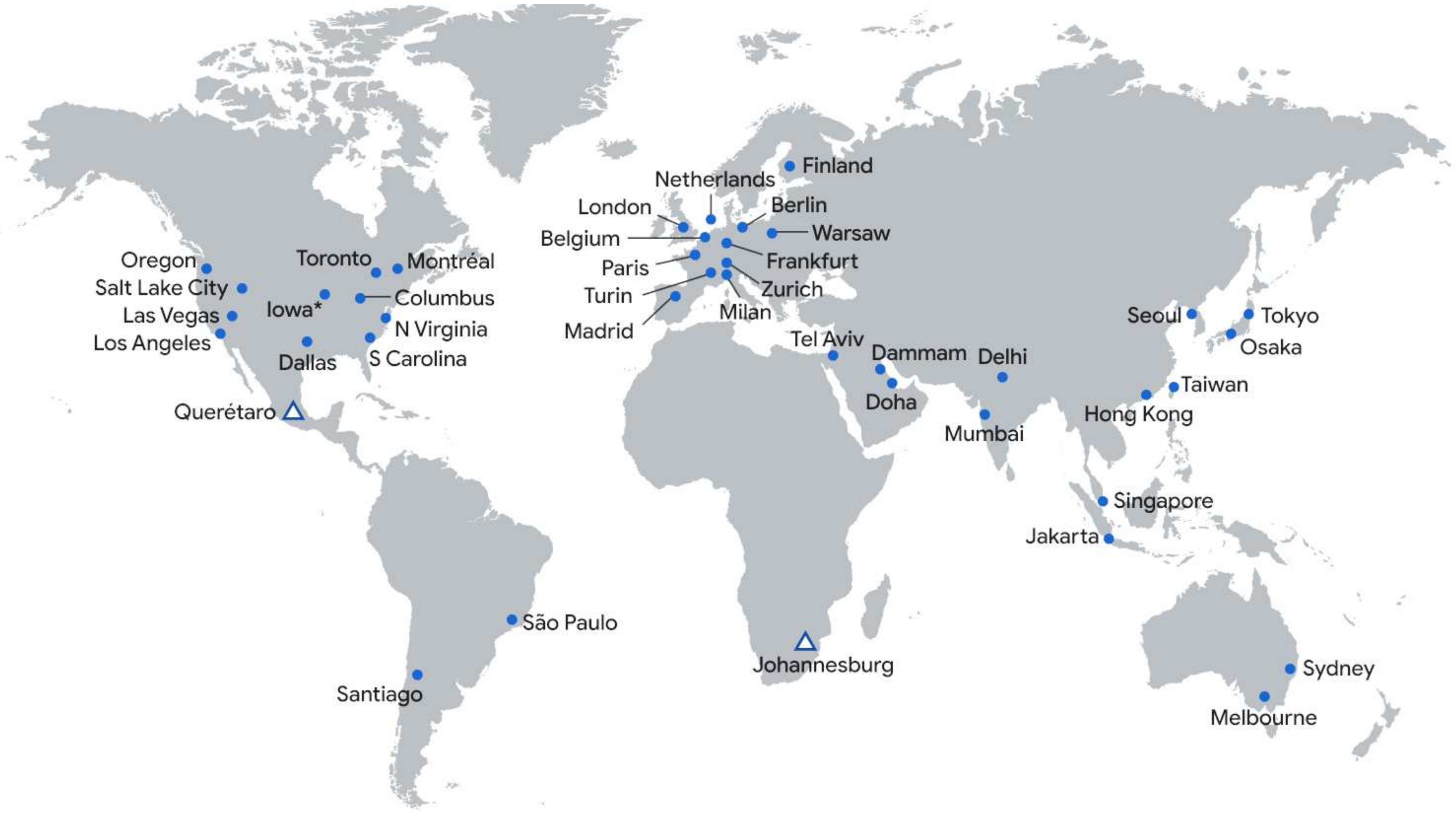
Spanner

- Scalable, globally distributed database
- Sharded, synchronously geo-replicated
- Externally consistent distributed transactions

Data Partitioning & Storage

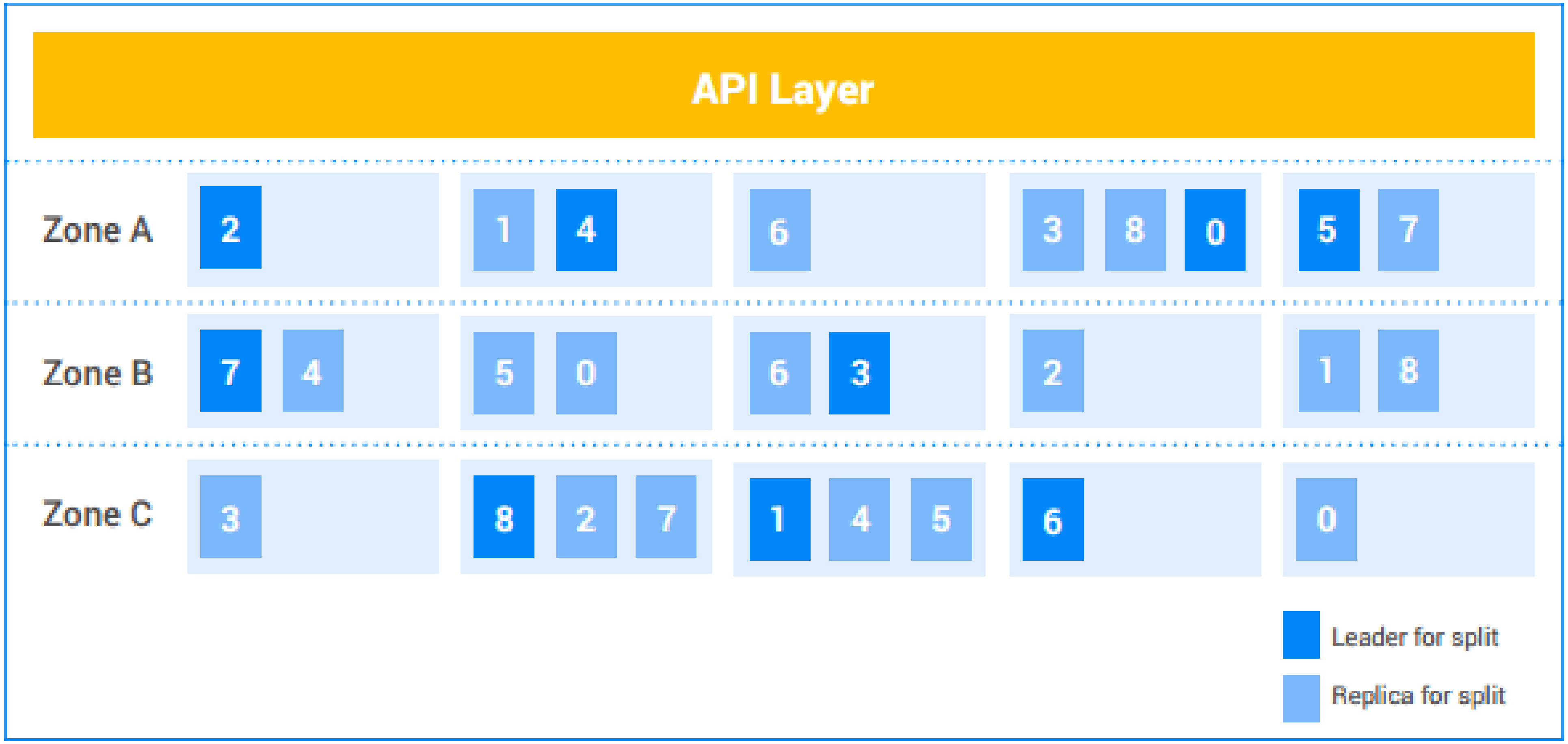
- Range-partitioned on table key
- Each contiguous key range is called a **split**
- Each split could be further partitioned vertically
- **LSM**-based storage
- Each data version is timestamped with **commit timestamp**

Regions & Zones



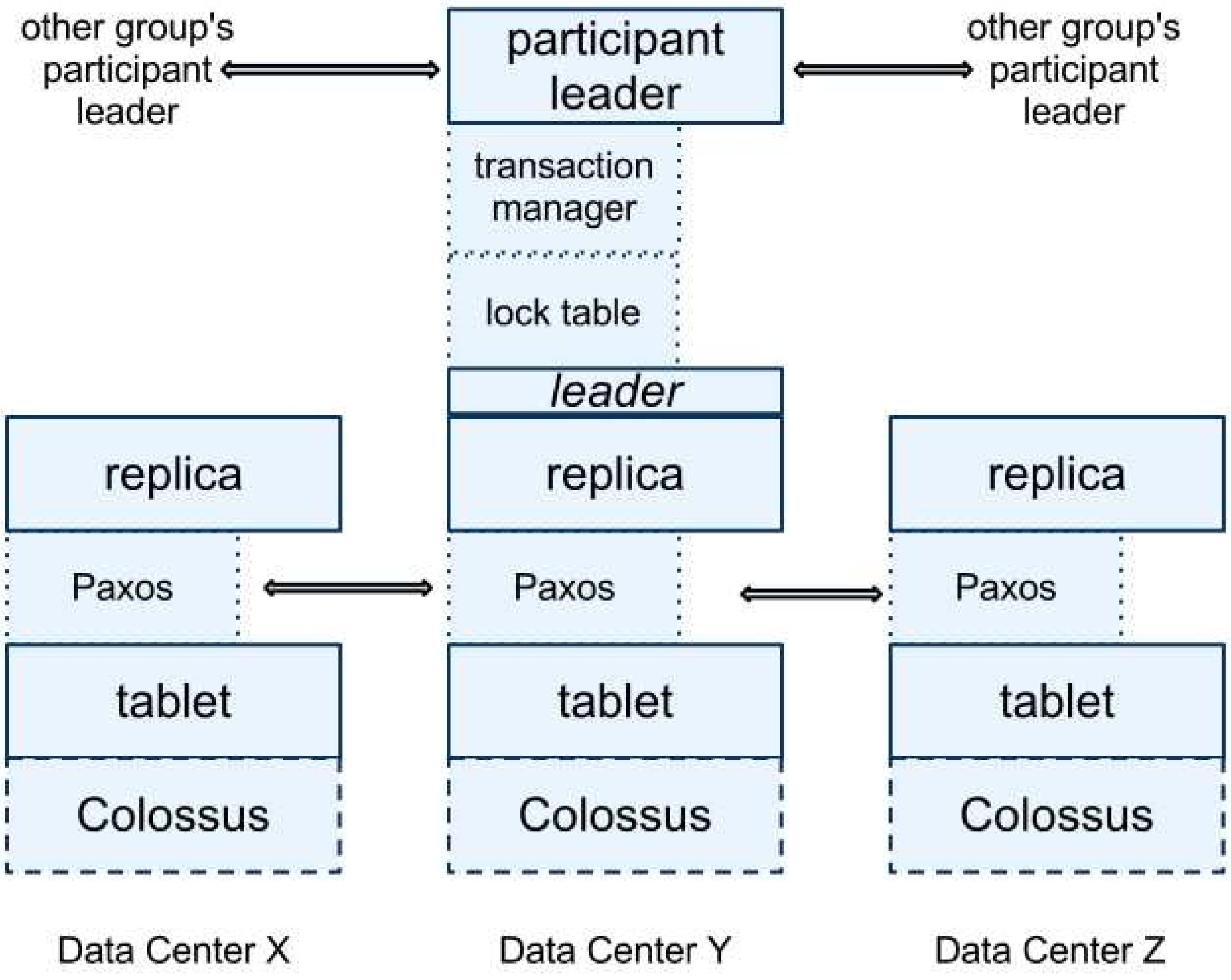
https://cloud.google.com/about/locations

Replication of Sharded Data



https://cloud.google.com/spanner/docs/whitepapers/life-of-reads-and-writes

Server Software Stack



Corbett, et al., 2013

Data Replication

- Synchronous replication using **Paxos consensus protocol**
- Each shard is assigned to one **Paxos group**
  - leader replica & slave replicas
  - Replicas are synchronized using replicated state machine
- Three types of replicas: read-write replicas, read-only replicas, & witness replicas
- **Single-region instances** use only read-write replicas
- **Multi-region instances** use a combination of all three types

Replica Types

Replica type	Can vote	Can become leader	Can serve reads
Read-write	yes	yes	yes
Read-only	no	no	yes
Witness	yes	no	no

Transaction Management

- Each **leader replica** implements a **transaction manager**
  - State of TM is replicated in its Paxos group
- Consider a transaction T that has updated n shards (i.e., n leader replicas)
- Commit of T involves n Paxos groups (one for each of the updated shards)
- For each Paxos group,
  - participant leader refers to leader replica
  - participant slaves refers to slave replicas
- If $n > 1$, one of the n participant groups is chosen as the coordinator for the commit of T using **2PC protocol**
  - coordinator leader refers to participant leader of chosen group
  - coordinator slaves refers to participant slaves of chosen group

Concurrency Control

- Uses Multiversion 2PL (MV2PL) for concurrency control
- Read-write Xacts (RW-xacts) are serialized using S2PL protocol
  - Each read is processed at appropriate leader replica
  - Writes are buffered in client & not visible to any Xact
  - At commit time, a new object version is created for each write
  - Deadlocks are prevented using Wound-wait policy
- To optimize for **blind writes**, uses a new lock mode writer share lock (WSL)
  - WSL is compatible only with WSL
- Read-only Xacts (RO-xacts) are executed without acquiring locks
  - RO-xact reads an appropriate DB snapshot

External Consistency (Strict Serializability)

- Strict serializability requires the serialization order to be consistent with the real-time ordering of transactions:

  if T_1 commits before the start of T_2, commitTS(T_1) $<$ commitTS(T_2)

- if T_1 commits before the start of T_2, clients should not see a database state that includes the effects of T_2 but does not include the effects of T_1

External Consistency (Strict Serializability)

- A schedule S is externally consistent (or strict serializable) if S is serializable and is equivalent to a serial schedule S' such that whenever a Xact T_i in S commits before the start of another Xact T_j in S, then T_i precedes T_j in S'

- **Example**: Consider the following schedule:

$$\begin{array}{llllll}
T_1: & W_1(x), & & & W_1(y), C_1 \\
T_2: & & R_2(x), C_2, & & \\
T_3: & & & W_3(y), C_3, &
\end{array}$$

TrueTime API

- Uses both satellite-connected GPS clocks & atomic clocks
- Achieves external consistency, consistent reads without locking, & consistent snapshots

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

- $t_{abs}(e)$ = absolute time of an event *e*
- e_{now} = invocation event of TT.now()
- $e_{now}.\text{earliest} \leq t_{abs}(e_{now}) \leq e_{now}.\text{latest}$

Read-Write Transactions

- Client issues each **read** request to leader replica
    - Acquires read lock & returns most recent version
    - Deadlocks are prevented using wound-wait algorithm
- Client buffers all **writes** for transaction T_i until T_i commits
- Client initiates **2PC** when it has completed all reads & buffered all writes for T_i
    - Client chooses **coordinator group** & sends commit message to each **participant leader**
    - Commit message includes identity of coordinator leader & any buffered writes

Read-Write Transactions (cont.)

- On receiving commit message, **non-coordinator participant leader** performs the following:
  - Acquires **write locks**
  - Chooses **prepare timestamp**
    - Prepare timestamp is greater than all timestamps previously chosen by non-coordinator participant leader
  - Writes a **prepare log record** (via Paxos)
  - Sends prepare timestamp to coordinator

Read-Write Transactions (cont.)

- On receiving commit message, **coordinator participant leader** (CPL) performs the following:
  - Acquires **write locks**
  - e_i^{start} = invocation event of TT.now()
  - On receiving all prepare timestamps, chooses commit timestamp cts_i for T_i
    - ★ $cts_i > \max\{e_i^{start}$.latest, all prepare timestamps, all timestamps previously assigned by CPL $\}$
    - ★ $\boldsymbol{cts_i > e_i^{start}}$**.latest** ensures that T_i's commit timestamp is larger than T_i's absolute start time
  - Writes a **commit log record** (via Paxos)
  - Commit-wait rule: Waits until TT.after(cts_i)
    - ★ e_i^{commit} = invocation event of TT.now()
    - ★ Commit-wait ensures that $\boldsymbol{cts_i < e_i^{commit}}$**.latest**; i.e., T_i's commit timestamp is smaller than T_i's absolute commit time
  - Sends commit timestamp to client & non-coordinator participant leaders

Read-Write Transactions (cont.)

- On receiving commit timestamp, **non-coordinator participant leader** performs the following:
    - Writes a **commit log record** (via Paxos)
    - Releases locks after updates have been processed

Enforcing Strict Serializability

- Consider two Xacts T_1 & T_2
- Assume T_1 commits before the start of T_2
  - e_1^{commit}.latest $<$ e_2^{start}.latest
- By commit wait rule, $cts_1 < e_1^{commit}$.latest
- Since e_2^{start}.latest $< cts_2$, we have $cts_1 < cts_2$

Read-Only Transactions

- **Read-only transactions**
  - Reads are executed in a system-chosen timestamp without locking
  - RO-Xacts do not need to be committed
- **Snapshot reads**
  - Reads a past snapshot of database without locking
  - Specifies a specific read timestamp

Read-Only Transactions

- Each replica maintains a value called safe time t_{safe}
  - t_{safe} = maximum timestamp at which a replica is up-to-date
  - A replica can satisfy a read at a timestamp rts if $\boldsymbol{rts \leq t_{safe}}$
  - A read at a replica is blocked if $rts > t_{safe}$
- The execution of a RO-Xact T_i consists of two steps:
  - Step 1: Choose a read timestamp rts_i for T_i
    - ⋆ $\boldsymbol{rts_i}$ **= TT.now().latest**
    - ⋆ This ensures T_i's absolute start time $< rts_i$
  - Step 2: Perform the reads at appropriate replica(s)
    - ⋆ Read from sufficiently up-to-date replicas (i.e., $rts_i \leq t_{safe}$)
- Consider a RO-Xact T_2 that starts after a committed RW-Xact T_1
  - T_1's absolute commit time $<$ T_2's absolute start time
  - T_2's absolute start time $< rts_2$ (by step 1)
  - $cts_1 <$ T_1's absolute commit time (by T_1's commit wait)
  - $cts_1 < rts_2$ (thus, T_1's updates are in T_2's read snapshot)

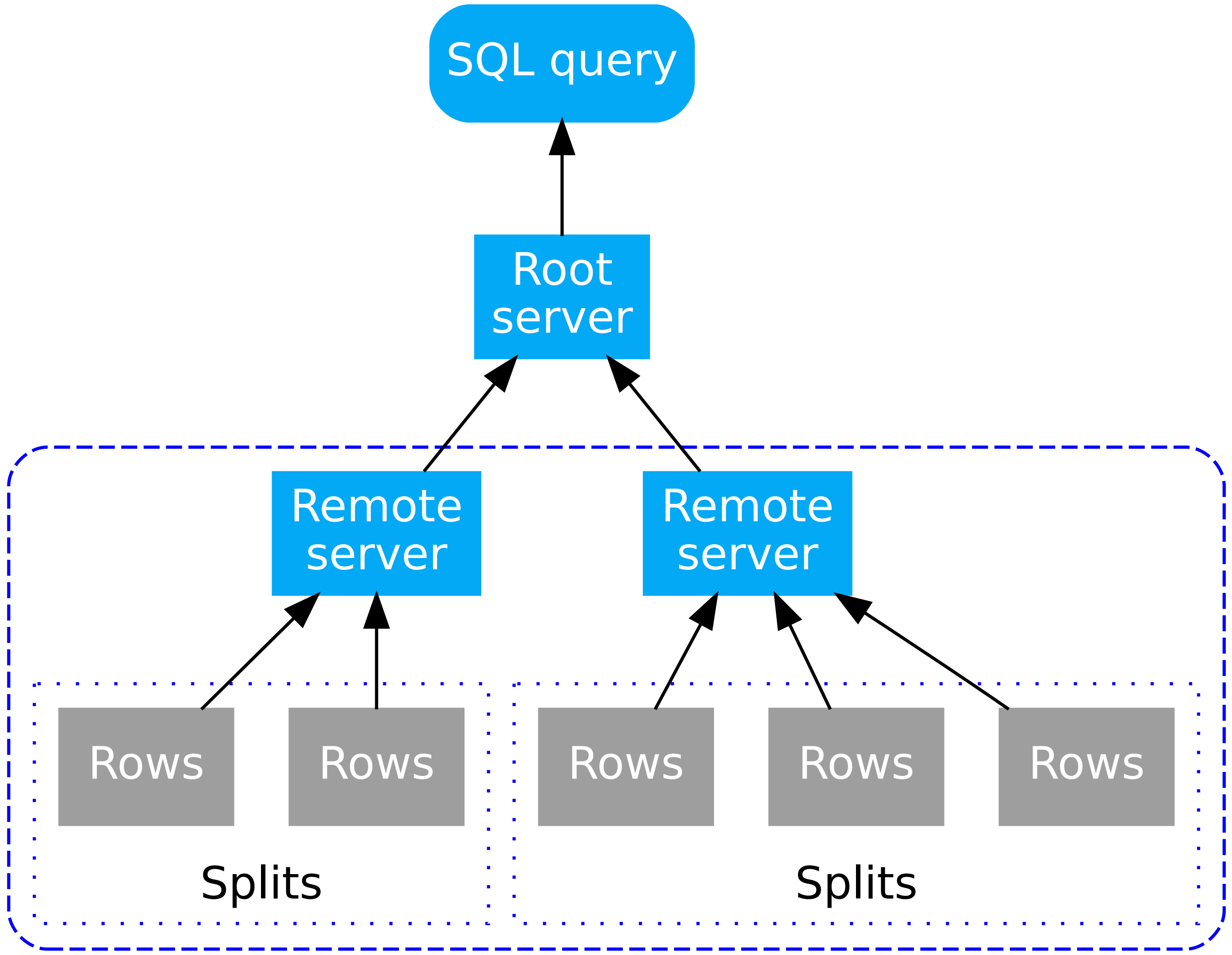
Safe Time t_{safe}

- $\boldsymbol{t_{safe} = \min(t_{safe}^{paxos}, t_{safe}^{tm})}$
- t_{safe}^{paxos} = safe time of Paxos state machine
  - t_{safe}^{paxos} = timestamp of last committed Paxos write
- t_{safe}^{tm} = safe time of transaction manager
  - Case 1: there are no prepared (but yet to be committed) Xacts
    - ★ $t_{safe}^{tm} = \infty$
  - Case 2: there's some prepared (but yet to be committed) Xact
    - ★ Let $pts_{i,g}$ denote the prepare timestamp chosen by participant leader (for Paxos group g) for prepared Xact T_i
    - ★ $\boldsymbol{t_{safe}^{tm} = \min_i(pts_{i,g}) - 1}$ over all Xacts prepared at g

Query Processing

- Query shipping
- Parallel query processing
- Partition pruning
- Local-Global optimization

Query Execution



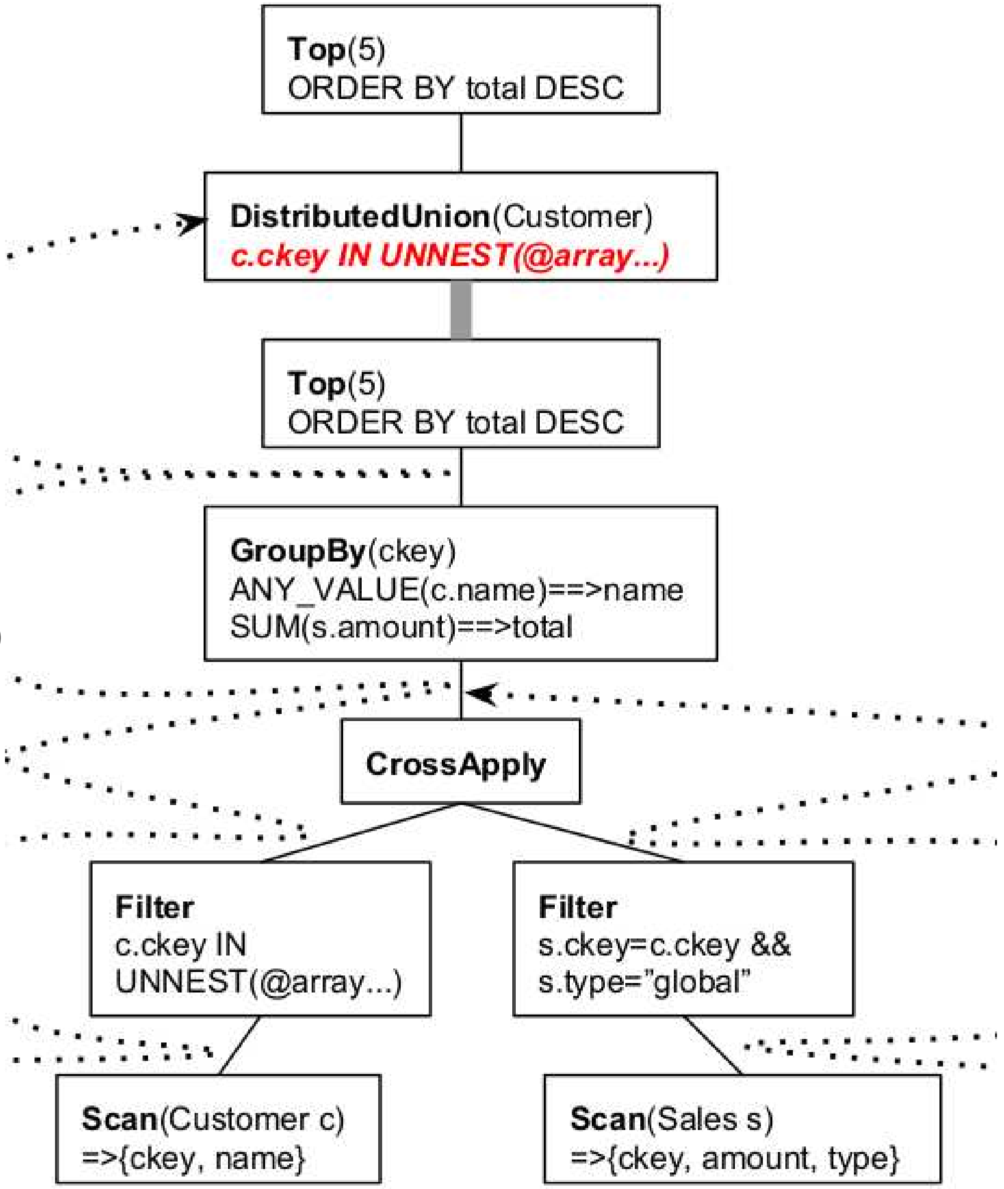
https://cloud.google.com/spanner/docs/query-execution-plans

Query Rewriting

- $\text{Scan(T)} \implies \text{DistributedUnion[shard} \subseteq \text{T](Scan(shard))}$
- $\mathbf{F}\text{(Scan(T))} = \text{DistributedUnion[shard} \subseteq \text{T](}\mathbf{F}\text{(Scan(shard)))}$
- $\text{Op(DistributedUnion[shard} \subseteq \text{T](}\mathbf{F}\text{(Scan(shard))))} =$
  $\text{OpFinal(DistributedUnion[shard} \subseteq \text{T](OpLocal(}\mathbf{F}\text{(Scan(shard)))))}$

Distributed Query Plans

```
SELECT ANY_VALUE(c.name) name,
      SUM(s.amount) total
FROM Customer c JOIN Sales s
      ON c.ckey = s.ckey
WHERE s.type = 'global'
AND c.ckey IN UNNEST(@customer_key_arr)
GROUP BY c.ckey
ORDER BY total DESC
LIMIT 5
```



Corbett, et al., 2013

References

- Cloud Spanner Documentation

  https://cloud.google.com/spanner/docs/

- James C. Corbett, et al., *Spanner: Google's Globally-Distributed Database*, TOCS 2013
- James C. Corbett, et al., *Spanner: Google's Globally-Distributed Database*, OSDI 2012

  https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

- David F. Bacon, et al., *Spanner: Becoming a SQL System*, SIGMOD 2017

  https://research.google.com/pubs/pub46103.html