# Summary 5 and 6

*Gabriela Gutiérrez Valverde - 2019024089*

*Book: Spanner: Becoming a SQL System*

## Introduction

Spanner is a globally-distributed data management system that backs hundreds of mission-critical services at Google. It has evolved into a relational database system, with strongly-typed schema system and a SQL query processor. Spanner is a full featured SQL system, with query execution tightly integrated with the other architectural features of Spanner (such as strong consistency and global replication) widely used as an OLTP database management system for structured data at Google. Replicas of the data are served from datacenters around the world to provide low latency to scattered clients. Provides transactional consistency, strongly consistent replicas and high availability. It implements a dialect of SQL, called Standard SQL, based on standard ANSI SQL, fully using standard features such as ARRAY and row type. Unique features:

- Distributed query execution
- Range extraction
- Query restarts
- Common SQL dialect
- Blockwise-columnar store

## Background

Spanner is a sharded, geo-replicated relational database system. Shards are distributed across multiple servers, then replicated to multiple, geographically separated datacenters. It may contain multiple tables, and the schema may specify parent-child relationships. Transactions use a replicated write-ahead redo log and the *Paxos* algorithm is used to get replicas to agree on the contents of each log entry. All transactions that involve data in a particular group write to a logical Paxos write-ahead log, each log entry is committed by successfully replicating it to a quorum of replicas. *Leader* is elected and can commit multiple log entries in parallel.

- only replicate log records
- Non-leader replicas may be temporarily behind the leader

For blind write and read-modify-write transactions, strict two-phase locking ensures serializability. Reads can be done in lock-free snapshot transactions. Stale reads choose a timestamp in the past. Strong reads see the effects of all previously committed transactions, they may have to wait for the nearby replica to become fully caught up.

**Coprocessor framework:** determines which Paxos group (or groups) owns the data being addressed, and finds the nearest replica of that group that is sufficiently up-to-date for the specified concurrency mode.

**Colosus:** append-only distributed filesystem, based on log-structured merge trees. Spanner query compiler first transforms an input query into a relational algebra tree, then uses schema and data properties to rewrite the initial algebraic tree into an efficient execution plan via transformation rules, like:

- pushing predicates toward scans, picking indexes, or removing subqueries

For logical operators:

- **CrossApply(input, map):** evaluates ‘map’ for every tuple from ‘input’ and concatenates the results of ‘map’ evaluations.
- **OuterApply(input, map):** emits a row of output even if ‘map’ produced an empty relation for a particular row from ‘input’.

## QUERY DISTRIBUTION

### Distributed query compilation

Traditional approach of building a relational algebra operator tree and optimizing it using equivalent rewrites. **Distributed Union:** used to ship a subquery to each shard and to concatenate the results. Provides a building block for more complex distributed operators. A global scan of the table is replaced by an explicit distributed operation that does local scans of table shards, ordering the shards according to the table key order if necessary:

```
Scan(T) -> DistributedUnion[shard  T](Scan(shard))
```

Push the maximum amount of computation down to the servers responsible for the data shards. Partitionability must be satisfied for any relational operation:

```
F(Scan(T)) = OrderedUnionAll[shard  T](F(Scan(shard)))
```

**Table interleaving:** mechanism of co-locating rows from multiple tables sharing a prefix of primary keys. Partial local Top or local aggregation are pushed to table shards below Distributed Union while results from shards are further merged or grouped on the machine executing the Distributed Union:

```
Op(DistributedUnion[shard  T](F(Scan(shard)))) = OpFinal(DistributedUnion[shard  T]
(OpLocal(F(Scan(shard))))
```

### Distributed Execution

Distributed Union minimizes latency by using the Spanner coprocessor framework to route a subquery request. Before executing the subquery, performs a runtime analysis of the sharding key filter expression to extract a set of sharding key ranges. Guaranteed to fully cover all table rows on which the subquery may yield results. Reduces latency by dispatching subqueries to each shard or group in parallel.

### Distributed joins

Apply Join, Cross Join, or Nested Loop Join in a distributed environment involves a cross-machine call made per each row coming from the left input of the join operator. Distributed Apply operator by extending Distributed Union and implementing Apply style join in a batched manner. Steps:

1. Evaluate the sharding key filter expression for each row.
2. Merge the sharding key ranges.
3. Compute the minimal set of shards to send the batch.
4. Construct a minimal batch for each shard.

### Query distribution APIs

- **Single-consumer API:** sent initially to a root, upon the first execution of a query, its physical plan is analyzed to detect an outermost distribution operator containing a sharding key expression. Then converted into a special pattern that encodes how to obtain sharding keys as a function of query parameters. Location hint, is cached on the client. In subsequent executions, the client can cheaply compute where to send the query without any further analysis.
- **Parallel-consumer API:** at Spanner’s scale, the results of some queries are prohibitively large to be consumed on one machine. First, divide the work between the desired number of clients, returns a set of opaque query partition descriptors. Then, the query is executed on the individual partitions, normally using requests initiated in parallel from separate machines. Guarantees that the concatenation of results from all the partitions yields the same unordered set of rows as for the query submitted. Requires queries to be root partitionable.

## QUERY RANGE EXTRACTION

### Problem statement

Range extraction:

- **Distribution range extraction:** routing the query to the servers hosting those shards.
- **Seek range extraction:** what fragments of the relevant shard to read from the underlying storage stack.
- **Lock range extraction:** what fragments of the table are to be locked, or checked for potential pending modifications.

### Compile-time rewriting

At compile time, it normalize and rewrite a filtered scan expression into a tree of correlated self-joins that extract the ranges for successive key columns. At runtime, we use a special data structure called a filter tree for both computing the ranges via bottom-up interval arithmetic and for efficient evaluation of postfiltering conditions. Compile-time rewriting performs a number of expression normalization steps, including the following:

- NOT is pushed to the leaf predicates.
- The leaves of the predicate tree that reference key columns are normalized by isolating the key references.
- Small integer intervals are discretized.
- Complex conditions are eliminated for the purposes of range extraction.

### Filter tree

Runtime data structure we developed that is simultaneously used for extracting the key ranges via bottom-up intersection / union of intervals, and for post-filtering the rows emitted by the correlated self-joins. Memoizes the results of predicates whose values have not changed, and prunes the interval computation. Range extraction is done one key column at a time. Every leaf node in the filter tree is assigned the initial interval for each key column.

## QUERY RESTARTS

Automatically compensates for failures, resharding, and binary rollouts, affecting request latencies in a minimal way. Resumes execution of snapshot queries.

### Usage scenarios and benefits

- **Hiding transient failures:** a snapshot transaction will never return an error on which the client needs to retry. Non-transient errors such as “deadline exceeded” or “snapshot data has been garbage collected” must still be expected. Includes network disconnects, machine reboots, and process crashes, as well as distributed wait and data movement.
- **Simpler programming model: no retry loops:** Retry loops in database client code is a source of hard to troubleshoot bugs, since writing a retry loop with proper backoff is not trivial.
- **Streaming pagination through query results:** enables efficient use of long-running queries instead of paging queries.
- **Improved tail latency for online requests:** important to have execution environment that ensures forward progress in case of transient failures.
- **Recurrent rolling upgrades:** allows to deploy new server versions regularly, without significantly affecting request latency and system load.
- **Simpler Spanner internal error handling:** Whenever a server cannot execute a request, whether it is a server-wide reason such as memory or CPU overuse, or related to a problematic shard or a dependent service failure, it can just return an internal retry error code and rely on the restart mechanism.

### Contract and requirements

Restart tokens accompany all query results, sent in batches, one restart token per batch, prevents the rows already returned to the client to be returned again. Restart implementation has to overcome the following challenges:

- **Dynamic resharding:** the query processor does not have the guarantee that the server covering a range of keys will receive a given subquery once.
- **Non-determinism:** it makes it difficult to fast-forward query execution to a given state without keeping track of large intermediate results.
- **Restarts across server versions:** must be compatible across versions: *Restart token wire format*, *Query plan*, and *Operator behavior*.

## COMMON SQL DIALECT

Defined a common data model, type system, syntax, semantics, and function library that the systems share. Lowers the barrier of working across the systems. Support a UTF8-based *STRING* type rather than the traditional *CHAR* and *VARCHAR* types. **Compiler front-end:** performs parsing, name resolution, type checking, and semantic validation. Has consistent error messages produced for semantic errors across systems. **Library of scalar functions:** reduces the chances for divergence in corner cases and brings consistency to runtime errors such as overflow checking. **Shared testing framework:** compliance tests and coverage tests. **FilterScanNodeGenerator:** has edges to a logical expression generator that chooses from several node generators that can generate boolean expression Resolved AST nodes for the condition, and a logical scan generator that chooses from several node generators that can produce scan Resolved AST nodes for the filter input.

## BLOCKWISE-COLUMNAR STORAGE

- **Ressi data layout:** stores a database as an LSM tree, whose layers are periodically compacted. Organizes data into blocks in row-major order. Divides the values into an active file, which contains only the most recent values, and an inactive file, which may contain many older versions.
- **Live migration from SSTables to Ressi:** is capable of live data migrations via bulk data movement and transactions. Group-by-group conversion either to new format replicas, to mixed format replicas for testing and verification, or back to SSTables for rollback. Copies data to the new group.