> *Bases de Datos 2, Resumen 5 y 6*
> *Miguel Ku Liang - 2019061913*

Introduction

Google's Spanner started as a key-value store offering multi-row transactions, external consistency and transparent failover across datacenters. It has evolved into a relational database system. The desire to make it behave like a traditional databse has forced the system to evolve:

- The architecture of the storage stack has driven fundamental changes in our query compilation and execution
- The demands of the query processor have driven fundamental changes in the way we store and manage data

Spanner is used as an OLTP database management system for structured data at Google and is available in beta as Cloud Spanner on the Google Cloud Platform.

The Spanner query processor implements a dialect of SQL called **Standard SQL**. It is based on standard ANSI SQL, fully using standard features like ARRAY and row type called STRUCT to support nested data as a first class citizen.

Spanner's query processor is built to serve a mix of transactional and analytical workloads and to support low-latency and long-running queries.

Background

Spanner is a sharded, geo-replicated relational database system. Spanner's transactions use a replicated write-ahead redo log, and the Paxos consensus algorithm is used to get replicas to agree on the contents of each log entry. Each shard of the database is assigned exactly one Paxos group. A group may be assigned multiple shards. All transactions that involve data in a group write to a logical Paxos write-ahead log. Paxos can make progress as long as a majority of replicas are up, achieving high availability despite failures.

Concurrency control uses a combination of pessimistic locking and timestamps. For blind write and read-modify-write transactions, strict two-phase locking ensures serializability within a given Paxos group, while two-phase commits ensure serializability across the database.

Spanner provides a special RPC framework called the **coprocessor framework**, to hide the complexity of locating data. Read and write requests are addressed to a key or key range, not to a particular server. The coprocessor framework determines which Paxos group owns the data being addressed and finds the nearest replica of that group that is sufficiently up-to-date for the specified concurrency mode.

A replica stores data in an append-only distributed filesystem called **Colossus**. The storage is based on log-structured merge trees which support high write throughput and fit well with the append-only nature of the filesystem. The original file format was a sorted map called **SSTables**.

Query Distribution

Distributed query compilation

The Spanner SQL query compiler uses the traditional approach of building a relational algebra operator tree and optimizing it using equivalent rewrites. Query distribution is represented using operators in the query algebra tree. One distribution operator is **Distributed Union**. It is used to ship a subquery to each shard of the underlying persistent or temporary data and to concatenate the results. Distributed Union is a fundamental operation in Spanner, because the sharding of a table may change during query execution and after a query restart.

Distributed Execution

Distributed Union minimizes latency by using the Spanner coprocessor framework to route a subquery request addressed to a shard to one of the nearest replicas that can serve the request. Shard pruning is used to avoid querying irrelevant shards. Shard pruning leverages the range keys of the shards and depends on pushing down conditions on sharding keys of tables to the underlying scans.

Distributed joins

The **batched apply join**'s primary use case is to join a secondary index and its independently distributed base table. It is also used for executing Inner/Left/Semi-joins with predicates involving the keys of the remote table. Spanner implements a Distributed Apply operator by extending Distributed Union and implementing Apply style join in a batched manner.

Distributed Apply allows Spanner to minimize the number of cross-machine calls for key-based joins and parallelize the execution. It allows turning full table scans into a series of minimal range scans.

Distributed Apply can execute its subquery in parallel on multiple shards. On each shard, seek range extraction takes care of scanning a minimal amount of data from the shard to answer the query.

Query distribution APIs

Spanner exposes two kinds of APIs for issuing queries and consuming results. The **single-consumer API** is used when a single client process consumes the results of a query. The **parallel-consumer API** is used for consuming query results in parallel from multiple processes running on multiple machines. This API is designed for data processing pipelines and map-reduce type systems that use multiple machines to join Spanner data with data from other systems or to perform data transformation outside of Spanner.

Query Range Extraction

Problem statement

Query range extraction refers to the process of analyzing a query and determining what portions of tables are referenced by the query. Spanner employs several flavors of range extraction:

- Distribution range extraction: knowing what table shards are referenced by the query is essential for routing the query to the servers hosting those shards
- Seek range extraction: once the query arrives at a Spanner server, we figure out what fragments of the relevant shard to read from the underlying storage stack.

- Lock range extraction: the extracted key ranges determine what fragments of the table are to be locked or checked for potential pending modifications.

Compile-time rewriting

The implementation of range extraction in Spanner relies on two main techniques. At compile time, we 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 post-filtering conditions.

Compile-time rewriting performs a number of expression normalization steps:

- NOT is pushed to the leaf predicates. This transformation is linear in the size of the expression.
- the leaves of the predicate tree that reference key columns are normalized by isolating the key reference.
- small integer intervals are discretized.
- complex conditions that contain subqueries or expensive library functions are eliminated for the purposes of range extraction.

Filter tree

The filter tree is a runtime data structure that is simultaneously used for extracting the key ranges via bottom-up intersection or union of intervals, and for post-filtering the rows emitted by the correlated self-joins. The tree memoizes the results of predicates whose values have not changed and prunes the interval computation.

Query Restarts

Spanner automatically compensates for failures, resharding and binary rollouts, affecting request latencies in a minimal way. The client library transparently resumes execution of snapshot queries if transient errors or other system events occur, even when partial results have been consumed by the user's code. In a read-modify-write transaction, a failure results in the loss of locks and requires the transaction to be aborted and retried. Remote subqueries executed as part of a client query by Spanner servers are restartable.

Usage scenarios and benefits

- Hiding transient failures: Spanner fully hides transient failures during query execution. This means that a snapshot transaction will never return an error on which the client needs to retry. Non-transient errors like "deadline exceeded" or "snapshot data has been garbage collected" must still be expected by clients. The list of transient failures Spanner hides includes network disconnects, machine reboots, process crashes, distributed wait and data movement. Spanner's failure handling implementation is optimized to minimize wasted work when recovering from transient failures.
- Simpler programming model: no retry loops: retry loops in database client code is a source of hard to troubleshoot bugs. Spanner users are encouraged to set realistic request deadlines and do not need to write retry loops around snapshot transactions and standalone read-only queries.
- Streaming pagination through query results: interactive applications use paging to retrieve results of a single query in portions that fit into memory and

constitute a reasonable chunk of work to do on the client side or on a middle tier. Batch processing jobs use paging to define portions of data to process in a single shot.
- Improved tail latency for online requests: Spanner's ability to hide transient failures and to redo minimal amount of work when restarting after a failure helps decrease tail latency for online requests.
- Forward progress for long-running queries: for long-running queries where the total running time of the query is comparable to mean time to failure is important to have execution environment that ensures forward progress in case of transient failures, or some queries may not finish in reasonable time, even with retry loops.
- Recurrent rolling upgrades: support for query restart complements other resumable operations like schema upgrades and online index building, allowing the Spanner team to deploy new server versions regularly without affecting request latency and system load.
- Simpler Spanner internal error handling: as Spanner uses restartable RPCs for client-server calls and for internal server-server calls, it simplified te architecture in regards to failure handling. Spanner server can return an internal retry error code and rely on the restart mechanism to not waste resources when the request is retried.

Contract and requirements

Spanner extended its RPC mechanism with an additional parameter, a restart token. **Restart tokens** accompany all query results, sent in batches, one restart token per batch. This restart token blob when added as a special parameter to the original request prevents the rows already returned to the client to be returned again. The restart contract makes no repeatability guarantees.

One way to hide transient failures is by buffering all results in persistent memory before delivering to the client and ensuring forward progress on transient failures by persisting intermediate results produced by each processing node. Does not work well for systems aiming at low latencies and requires the intermediate results to be persisted as reliably as the primary storage.

Instead, implement a fully streaming request processing where neither intermediate nor final results hit persistent storage for the purpose of restartability.

The restart implementation has to overcome the following challenges:

- Dynamic resharding: Spanner uses query restarts to continue execution when a server loses ownership of a shard or boundaries of a shard change. A request targeted to a key range needs to cope with ongoing splitting, merging and moving of the data.
- Non-determinism: many opportunities for improving query performance in a distributed environment present sources of non-deterministic execution that causes result rows to be returned in some non-repeatable order.
- Restarts across server versions: a new version of the server code may introduce subtle changes to the query processor that need to be addressed to preserve restartability. Restart token wire format, uery plan and operator behavior are aspects of Spanner that must be compatible across the versions.

Common SQL Dialect

Spanner's SQL engine shares a common SQL dialect called **Standard SQL**, with several other systems at Google including internal systems like F1 and Dremel, and external systems like BigQuery.

The effort to standardize the implementations covers gaps in the SQL standard where details are unclear or left to the implementation and covers the Google specific extensions. To ensure consistency between systems, several shared components were built as a collaboration between teams, referred to internally as the GoogleSQL library.

The first component is the **compiler front-end**. It performs parsing, name resolution, type checking and semantic validation. The second component is a **library of scalar functions**. Semantics of scalar functions are defined in the language specification and documentation, but direct sharing reduces the chances for divergence in corner cases and brings consistency to runtime errors such as overflow checking. The third component is a **shared testing framework and shared tests**. These tests fall into two categories, compliance and coverage tests. Compliance tests are a suite of developer written queries with a hard-coded result or supplied result checking procedure. Coverage tests use a random query generation tool and a reference engine implementation to check query results.

Blockwise-columnar storage

The SSTable data format inherited from Bigtable is optimized for schemaless NoSQL data consisting of large string. It is self-describing and highly redundant, and a traversal of individual columns within the same locality group is inefficient.

Ressi is the new low-level storage format for Spanner.

Ressi data layout

Ressi stores a database as an LSM tree. Each layer, Ressi organizes data into blocks in row-major order, but lays out the data within a block in column-major order.

Live migration from SSTables to Ressi

Migrating the storage format from SSTable to Ressi for a globally replicated, highly available database like Spanner requires extreme care in conversion to ensure data integrity and minimal, reversible rollout to avoid user-visible latency spikes or outages.

Conclusions

Spanner was designed from the beginning to run exclusively as a service. It is important to understand Spanner's relationship to the NoSQL movement. ACID transactions spanning arbitrary rows or keys is the next hardest barrier for scalable data management systems.