Instituto Tecnológico de Costa Rica

Bases de Datos - IC4302

Resumen 5 y 6

Estudiante:

Andrea María Li Hernández - 2021028783

Profesor:

Gerardo Nereo Campos Araya

Fecha de entrega: 25 de octubre del 2022
Segundo Semestre, 2022

Spanner: Becoming a SQL System

1. Introduction

Spanner is a relational database system from Google with a strongly typed schema system and a SQL query processor. It was previously a key-value store that offered multi-row transactions and transparent failover across datacenters. The Spanner query processor implements a dialect of SQL, called Standard SQL, the processor is built to serve a mix of transactional and analytical workloads, and to support both low-latency and long-running queries. In this summary you'll find how query execution has evolved and forced Spanner to evolve.

2. Background

This section consists of a brief review of the architecture of the Spanner system. Spanner is a shared, geo-replicated relational database system. It is horizontally row-range sharded. Within a data center, shards are distributed across multiple servers. Shards are then replicas to multiple, geographically separated datacenters.

- **Multi-Paxos**: Spanner's transactions use a replicated write-ahead log and the Paxos consensus algorithm to implement a form of Multi-Paxos, in which a single long-lived leader is elected and can commit multiple log entries in parallel, achieving high throughput. Hence achieving high availability despite server, network and data center failures.
- **Concurrency control**: Spanner uses a combination of pessimistic locking and timestamps. Using strict two-phase commits within a Paxos group and two-phase commits they ensure serializability across the database.
- **Coprocessor framework**: It is an RPC framework that 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.
- **Colossus**: This is a append-only distributed filesystem where a given replica stores data. The storage is based on log-structured merge trees.
- **The Spanner query compiler**: It first transforms an input query into a relational algebra tree. It uses transformation rules for an efficient execution as pushing predicates toward scans and picking indexes. Internally, it uses correlated join operatoes, for algebraic transformations and physical operators.

3. Query distribution

This section describes Spanner's distributed query processor, which serves online and long running queries by executing code in parallel on multiple machines.

3.1 Distributed query compilation

- The Spanner SQL query compiler builds a relational algebra operator tree and optimizes it using equivalent rewrites.
- Query distribution is represented using explicit operators in the algebra tree.
- Distributed Union: It is used to ship a subquery to each shard of the underlying persistent or temporary data, and to concatenate the results. It is inserted immediately above every Spanner table in the relational algebra.

3.2 Distributed Execution

- At runtime, Distributed Union minimizes latency by using the Spanner coprocessor framework to route a subquery request addressed to shard to one of the nearest replicas that can serve the request.
- **Shard pruning**: It is used to avoid querying irrelevant shards, it leverages the range keys of the shards and depends on pushing down conditions on sharding keys of tables to the underlying scans.
- During execution, **Distributed Union** may detect that the target shards are hosted locally on the server and can avoid making the remote call by executing its subquery locally.
- Shard affinity is typical for small databases that can fit into a single server or for shards sharing the key prefix.
    - This is an important latency optimization since most queries operate on rows from multiple tables sharing the same sharding key.

3.3. Distributed joins

- **Batched apply join**: Its 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 predicated involving the keys of the remote table.
- **Distributed Apply operator**: Spanner implements this by extending Distributed Union and implementing Apply style join in a batched manner.
- It uses two joins, one that applies batches of rows from the input to remote subquery and another that applies rows from each batch to the original join’s subquery locally on a shard.
- It allows Spanner to minimize the number of cross-machine calls for key-based joins and parallelize the execution. It also allows turning full table scans into a series of minimal range scans.
- It can execute its subquery in parallel on multiple shards.

3.4 Query distribution APIs

Spanner uses two kinds of APIs for issuing queries and consuming result:

- **Single-costumer API**: It is used when a single client process consumes the result of a query.
- **Parallel-consumer API**: It is used for consuming query results in parallel from multiple processes usually 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 transformations outside of Spanner.

4. Query range extraction

This refers to the process of analyzing a query and determining what portions of tables are referenced by the query.

4.1 Problem statement

Some of the flavors of range extraction that Spanner employs are Distribution range extraction, Seek range extraction and Lock range extraction.

4.2 Compile-time rewriting

Their implementation of range extraction relies on two main techniques:

- **At compile time**: They normalize and rewrite a filtered scan expression into a tree of correlated self-joins that extract the ranges for successive key columns. Also, rewriting performs several expression normalization steps.
- **At runtime**: They 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.

4.3 Filter tree

- It is a runtime data structure developed to simultaneously extract the key ranges via bottom-up intersection / union of intervals, and for post-filtering the rows emitted by the correlated self-joins.
- The tree is shared across all correlated scans produced by the compile-time rewriting.
- The tree memorizes the results of predicates whose values have not changed and prunes the interval computation.

5. Query restarts

In this section, we'll briefly explain how restarts are used, their benefits and the outline of the technical challenges in their implementation.

5.1 Usage scenarios and benefits

Some usage scenarios are the following:

- **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. Spanner users are encouraged to set realistic request deadlines and do not need to write retry loops around snapshot transactions.
- **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.

5.2 Contract and requirements

To support restarts Spanner extended its RPC mechanism with an additional parameter, a restart token.

- **Restart tokens**: They accompany all query results, sent in batches, one restart token per batch. This prevents the rows already returned to the client to be returned again. The restart contract makes no repeatability guarantees.

The team implemented SQL query restarts inside the query processor by capturing the distributed state of the query plan being executed. Some of the challenges that the restart implementation must overcome are:

- Non-determinism: It is difficult to fast-forward query execution to a given state without keeping track of large intermediate results. All the sources of non-repeatability need to be accounted for and compensated to preserve the restart contract.

- **Restarts across server versions**: The possible changes made to the query processor need to be addressed to preserve restartability. The following aspects of Spanner must be compatible across these versions:Restart token wire format, Query plan and Operator behavior.

These challenges are worth addressing because transparent restarts improve user-perceived system stability and provide important flexibility in other aspects of Spanner's design.

6. Common SQL dialect

In this section, we'll describe the common SQL dialect called **"Standard SQL"**, which consists of a common data model, type system, syntax, semantics, and function library that the systems share. A developer or data analyst who writes SQL against Spanner database can transfer their understanding of the language and not concern about subtle differences in syntax, NULL handling, etc.

To ensure consistency between systems, there are several shared components referred as the GoogleSQL library.

- **Compiler front-end**: This component performs parsing, name resolution, type checking, and semantic validation. Sharing this component prevents many subtle divergences that could otherwise arrive in.
    - It outputs a data structure called a "Resolved Abstract Syntax Tree (AST)", which contains the type of information and linked schema objects.
- **Library of scalar functions**: The semantics of scalar functions are all defined in the language specification and user documentation. Direct sharing reduces the chances for divergence in corner cases and provides consistency to runtime errors such as overflow checking.
- **Shared testing framework and shared tests**: These tests primarily fall into two categories:
    - **Compliance test**: They are a suite of developer written queries each with a hard-coded result or supplied result checking procedure.
    - **Coverage tests**: They use a random query generation tool and a reference engine implementation to check query results.
        - The primary random query generation tool that they use targets the Resolved AST. It is a graph of node generators; each node generator corresponds to a kind of Resolved AST node. A problem with both tests is find blind spots, some bugs found by the random tests, for instance, signal gap in compliance tests that the team can fill.

7. Blockwise-Columnar storage

In this section, we'll overview and discuss some challenges for Ressi, the low-level storage format for Spanner, which is designed for handling SQL queries over large-scale, distributed databases comprising both OLTP and OLAP workloads. Previously, Spanner worked with SSTables, but since they left a lot of performance left behind, they migrated to Ressi.

Ressi data layout

- Ressi stores a database as an **LSM tree**, whose layers are periodically compacted. Within each layer, Ressi organizes data into blocks in row-major order but lays out the data within a block in column-major order.
- Since Spanner is a time-versioned database, there may be many values of a particular row-key/column, with different timestamps. So, Ressi divides the values into an active file.
    - This allows queries for the most recent data to avoid loading old versions.
    - Ressi's fundamental data structure is the vector.

Live migration from SSTable to Ressi

Migrating requires extraordinary care in conversion to ensure data integrity and minimal rollout to avoid user-visible latency spikes.

- Spanner is capable of live data migrations via bulk data movements and transactions.
- This mechanism copies data (and converts as needed) from the current group to a new one. Once all data has reached the new group it can take over responsibility for serving requests. The format change is pushed gradually to minimize the impact to live traffic.

8. Lessons learned and challenges

Here are some lessons learned and challenges from the evolution of Spanner:

- Production deployment with internal customers taught the team a great deal about the requirements of web scale database applications.
- Internal SQL adoption increased after the team switched to a common Google-wide SQL dialect.
- The improvement of True-Time epsilon, the measure of clock drift between servers, has simplified certain aspects of querying processing as timestamp picking for SQL queries.
- Due to the declarative nature of SQL, query optimization in Spanner is as relevant as in traditional database systems.
- It is a on going challenge to make Spanner perform well and be cost-effective across a broad spectrum of use cases over time.

9. Conclusions

- Several aspects of a DBMS architecture had to be rethought to have today's Spanner level of scalability and manageability.
- Spanner has an important relationship to the NoSQL movement. It needed transparent failover across clusters and the ability to reshard data easily upon expanding the system.
- Making ACID work at scale is extremely difficult and required substantial innovation in Spanner which is why a transactional NoSQL core can be used to build a scalable SQL DBMS.
- The team's recommendation is to start off with the relational model early on, once a scalable and available storage core is in place; well-knows relational abstractions speed up the development and reduces the costs of future migration.