Resumen 5 y Resumen 6

Estudiante: Celina Madrigal Murillo
Carné: 2020059364

Spanner: Becoming a SQL System

INTRODUCTION

Google's Spanner started out as a key-value store offering multirow transactions, external consistency, and transparent failover across datacenters. Over the past 7 years it has evolved into a relational database system. In that time we have added a strongly-typed schema system and a SQL query processor, among other features. Initially, some of these database features were "bolted on" – the first version of our query system used high-level APIs almost like an external application, and its design did not leverage many of the unique features of the Spanner storage architecture. However, as we have developed the system, the desire to make it behave more like a traditional database has forced the system to evolve. In particular:

- The architecture of the distributed 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.

These changes have allowed us to preserve the massive scalability of Spanner, while offering customers a powerful platform for database applications.

BACKGROUND

Spanner is a sharded, geo-replicated relational database system. A database is horizontally row-range sharded. Within a data center, shards are distributed across multiple servers. Shards are then replicated to multiple, geographically separated datacenters. A database may contain multiple tables, and the schema may specify parent-child relationships between tables. The child table is co-located with the parent table. It can be interleaved in the parent, so that all child rows whose primary key is prefixed with key parts 'K' are stored physically next to the parent row whose key is 'K'. Shard boundaries are specified as ranges of key prefixes and preserve row co-location of interleaved child tables. 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.

3. QUERY DISTRIBUTION

3.1 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 explicit operators in the query algebra tree. One of such distribution operators 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. This operator provides a building block for more complex distributed operators such as distributed joins between independently sharded tables.

3.2 Distributed Execution

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.

3.3 Distributed joins

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 predicates involving the keys of the remote table. Apply-style joins that are correlated on table keys work very well in a single-machine query processing environment where the storage layer supports seeks. Distributed Apply allows Spanner to minimize the number of cross-machine calls for key-based joins and parallelize the execution.

3.4 Query distribution APIs

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 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

4.1 Problem statement

Query range extraction refers to the process of analyzing a query and determining what portions of tables are referenced by the query. The referenced row ranges are expressed as intervals of primary key values. 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: In this case, the extracted key ranges determine what fragments of the table are to be locked, or checked for potential pending modifications.

4.2 Compile-time rewriting

Compile-time rewriting performs a number of expression normalization steps, including the following:

- NOT is pushed to the leaf predicates. This transformation is linear in the size of the expression. We do not translate filter conditions into a DNF or CNF because of the possibility of exponential blowup.
- The leaves of the predicate tree that reference key columns are normalized by isolating the key references.
- Small integer intervals are discretized.
- Complex conditions that contain subqueries or expensive library functions, arithmetic, etc. are eliminated for the purposes of range extraction.

4.3 Filter tree

The filter tree is a 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. The filter tree is shared across all correlated scans produced by the compile-time rewriting.

5. QUERY RESTARTS

5.1 Usage scenarios and benefits

Hiding transient failures. Spanner fully hides transient failures during query execution. This is unlike most other distributed query processors that hide some transient failures but not necessarily all.
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. Traditionally, 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.
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 it is important to have execution environment that ensures forward progress in case of transient failures.
Recurrent rolling upgrades. This is an internal benefit which has been extremely important for the agility of Spanner development and the ability to deploy bug fixes quickly. **Simpler Spanner internal error handling.** As Spanner uses restartable RPCs not only for client-server calls but also for internal server-server calls, it simplified the architecture in regards to failure handling.

5.2 Contract and requirements

To support restarts 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 opaque 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. 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. That is, 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 nondeterministic execution, which 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.These changes need to be addressed to preserve restartability. The following aspects of Spanner must be compatible across these versions:

- ***Restart token wire format.*** Serialization format of restart token must be versioned and must be able to be parsed by the previous and the next version of the server.

- ***Query plan.*** A query that started on version N and is restarted on version N+1 should be executed by the same operator tree, even when compiling the same SQL statement would result in a different plan in version N+1.
- ***Operator behavior.*** Operators must be able to interpret their portions of restart token the same way as the previous and the next version of the server.

6. COMMON SQL DIALECT

Spanner's SQL engine shares a common SQL dialect, called "Standard SQL", with several other systems at Google including internal systems and external systems such as BigQuery. In collaboration with those teams, we have defined a common data model, type system, syntax, semantics, and function library that the systems share. For users within Google, this lowers the barrier of working across the systems. A developer or data analyst who writes SQL against a Spanner database can transfer their understanding of the language to Dremel without concern over subtle differences in syntax, NULL handling, etc.

7. BLOCKWISE-COLUMNAR STORAGE

Ressi data layout

As with SSTables, Ressi stores a database as an LSM tree, whoselayers 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. Ressi supports Spanner's INTERLEAVE IN PARENT directive by storing rows of child tables in the same (or nearby) blocks as their parent rows. A multi-level index provides fast binary search for random key access. Since Spanner is a time-versioned database, there may be many values of a particular row-key/column, with different timestamps. Ressi divides the values into an active file, which contains only the most recent values, and an inactive file, which may contain many older versions (or none). This allows queries for the most recent data to avoid loading old versions. Ressi's fundamental data structure is the vector, which is an ordinally indexed sequence of homogeneously typed values. Within a block, each column is represented by one or multiple vectors. Ressi can operate directly on vectors in compressed form.

Live migration from SSTables to Ressi

Migrating the storage format from SSTable to Ressi for a globally replicated, highly available database like Spanner requires both extraordinary care in conversion to ensure data integrity and minimal, reversible rollout to avoid user-visible latency spikes or outages. Not only must user data be preserved in its entirety, the entire migration process must be minimally invasive and perturb ongoing user workloads as little as possible. Spanner is capable of live data migrations via bulk data movement and transactions. Spanner's data movement mechanism copies data (and converts as needed) from the current group, which continues to serve live data from a full set of replicas, to the new group.

8. LESSONS LEARNED AND CHALLENGES

Aggressive focus on horizontal scalability enabled widespread deployment without investing too heavily in single machine performance. The original API of Spanner provided NoSQL methods for point lookups and range scans of individual and interleaved tables. While NoSQL methods provided a simple path to launching Spanner, and continue to be useful in simple retrieval scenarios, SQL has provided significant additional value in expressing more complex data access patterns and pushing computation to the data. Spanner supports very strong guarantees that establish a global order among transactions, called external consistency. Within a

given read-modify-write transaction, however, Spanner imposes a constraint that all updates must be committed as the last step of the transaction. Since the OSDI'12 paper, Spanner continued improving its True-Time epsilon, the measure of clock drift between servers. This has simplified certain aspects of query processing, in particular, timestamp picking for SQL queries. Spanner offers a wide array of physical data layout options. These include geographic placement, replication, using protocol buffers vs. interleaved tables, vertical partitioning of tables within shards, etc.

9. CONCLUSIONS

Several aspects of a DBMS architecture had to be rethought to accommodate the scale at which Spanner operates: Spanner was designed from the beginning to run exclusively as a service; there is a huge economy of scale in centralizing and automating the human load of database management. Spanner does not use static table partitioning. Shard boundaries may change dynamically due to load balancing or reconfiguration; a query may see different table partitioning layouts across restarts. The communication layer of Spanner uses remote calls that are addressed to logical data ranges, not to compute nodes or processors directly; calls are routed to the best replica. A variety of replication configurations are used to reduce read/write latencies and replication cost of dissimilar workloads. Table shards can become unavailable without affecting workloads on other data stored in the same table(s). Database metadata, which includes schema versions and data placement information, is managed as data due to its volume. Last but not least, Spanner's control plane, which monitors server health, SLOs, data placement, lock conflicts, availability of the storage layer, etc. has evolved into a complex database application of its own.