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.

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.

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.

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:

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