BigTable Refinements

This lesson will explore different refinements that BigTable implemented.

We'll cover the following

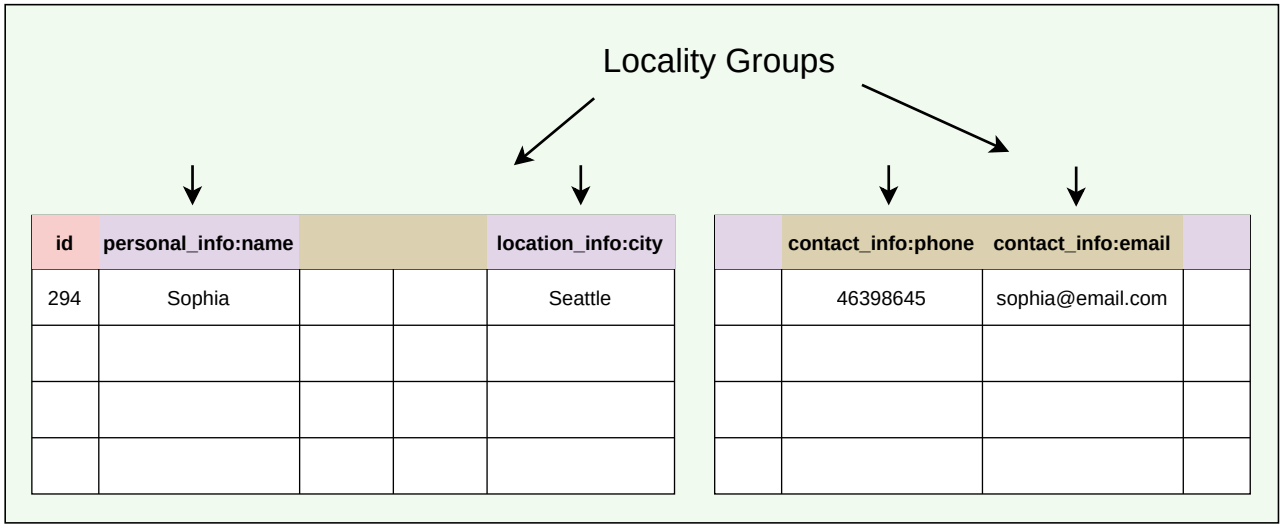
BigTable implemented certain refinements to achieve high performance, availability, and reliability. Here are their details:

Locality groups#

Clients can club together multiple column families into a locality group. BigTable generates separate SSTables for each locality group. This has two benefits:

- Grouping columns that are frequently accessed together in a locality group enhances the read performance.
- Clients can explicitly declare any locality group to be in memory for faster access. This way, smaller locality groups that are frequently accessed can be kept in memory.

- Scans over one locality group are $O(\text{bytes\_in\_locality\_group})$ and not $O(\text{bytes\_in\_table})$.



Grouping together columns to form locality groups

Compression#

Clients can choose to compress the SSTable for a locality group to save space. BigTable allows its clients to choose compression techniques based on their application requirements. The compression ratio gets even better when multiple versions of the same data are stored. Compression is applied to each SSTable block separately.

Caching#

To improve read performance, Tablet servers employ two levels of caching:

- **Scan Cache** — It caches (key, value) pairs returned by the SSTable and is useful for applications that read the same data multiple times.
- **Block Cache** — It caches SSTable blocks read from GFS and is useful for the applications that tend to read the data which is close to the data they recently read (e.g., sequential or random reads of different

columns in the same locality group within a frequently accessed row).

Bloom filters#

Any read operation has to read from all SSTables that make up a Tablet. If these SSTables are not in memory, the read operation may end up doing many disk accesses. To reduce the number of disk accesses BigTable uses Bloom Filters.

Bloom Filters are created for SSTables (particularly for the locality groups). They help to reduce the number of disk accesses by predicting if an SSTable may contain data corresponding to a particular (row, column) pair. Bloom filters take a small amount of memory but can improve the read performance drastically.

Unified commit Log#

Instead of maintaining separate commit log files for each Tablet, BigTable maintains one log file for a Tablet server. This gives better write performance. Since each write has to go to the commit log, writing to a large number of log files would be slow as it could cause a large number of disk seeks.

One disadvantage of having a single log file is that it complicates the Tablet recovery process. When a Tablet server dies, the Tablets that it served will be moved to other Tablet servers. To recover the state for a Tablet, the new Tablet server needs to reapply the mutations for that Tablet from the commit log written by the original Tablet server. However, the mutations for these Tablets were co-mingled in the same physical log file. One approach would be for each new Tablet server to read this full commit log file and apply just the entries needed for the Tablets it needs to recover. However, under such a scheme, if 100 machines were each assigned a single Tablet from a failed Tablet server,

then the log file would be read 100 times. BigTable avoids duplicating log reads by first sorting the commit log entries in order of the keys `<table, row name, log sequence number>`. In the sorted output, all mutations for a particular Tablet are contiguous and can therefore be read efficiently

To further improve the performance, each Tablet server maintains two log writing threads — each writing to its own and separate log file. Only one of the threads is active at a time. If one of the threads is performing poorly (say, due to network congestion), the writing switches to the other thread. Log entries have sequence numbers to allow the recovery process.

Speeding up Tablet recovery#

As we saw above, one of the complicated and time-consuming tasks while loading Tablets is to ensure that the Tablet server loads all entries from the commit log. When the master moves a Tablet from one Tablet server to another, the source Tablet server performs compactions to ensure that the destination Tablet server does not have to read the commit log. This is done in three steps:

- In the first step, the source server performs a minor compaction. This compaction reduces the amount of data in the commit log.
- After this, the source Tablet server stops serving the Tablet.
- Finally, the source server performs another (usually very fast) minor compaction to apply any new log entries that have arrived while the first minor compaction was being performed. After this second minor compaction is complete, the Tablet can be loaded on another Tablet server without requiring any recovery of log entries.

← Back

Next →

Fault Tolerance and Compaction BigTable Characteristics