CQL Data Modeling Overview

Wednesday, August 13, 2014 16:01

About CQL for Cassandra 1.2

Cassandra Query Language (CQL) is a SQL (Structured Query Language)-like language for querying Cassandra. The version of CQL described in this document and the default mode in Cassandra 1.2.x is based on the CQL specification 3.0. Cassandra's data model is a partitioned row store with tunable consistency. Rows are organized into tables; the first component of a table's primary key is the partition key; within a partition, rows are clustered by the remaining columns of the key. Other columns can be indexed separately from the primary key. Tables can be created, dropped, and altered at runtime without blocking updates and queries. Cassandra does not support joins or subqueries, except for batch analysis through Hadoop. Rather, Cassandra emphasizes denormalization through features like collections. CQL is the default and primary interface into the Cassandra DBMS. CQL provides a new API to Cassandra that is simpler than the Thrift API for new applications. The Thrift API, the Cassandra Command Line Interface (CLI), and legacy versions of CQL expose the internal storage structure of Cassandra. CQL adds an abstraction layer that hides implementation details and provides native syntaxes for CQL collections and other common encodings.

Compound keys and clustering

A compound primary key includes the partition key, which determines on which node data is stored, and one or more additional columns that determine clustering (The storage engine process that creates an index and keeps data in order based on the index). Cassandra uses the first column name in the primary key definition as the partition key. The remaining column, or columns that are not partition keys in the primary key definition are the clustering columns. The data for each partition is clustered by the remaining column or columns of the primary key definition. On a physical node, when rows for a partition key are stored in order based on the clustering columns, retrieval of rows is very efficient. Insertion, update, and deletion operations on rows sharing the same partition key for a table are performed atomically and in isolation.

Collection Columns

CQL 3 introduces these collection types:

- **Set**: A set stores a group of elements that are returned in sorted order when queried. A column of type set consists of unordered u nique values. Using the set data type, you can solve the multiple email problem in an intuitive way that does not require a read before adding a new email address.
  Details Usage: http://www.datastax.com/documentation/cql/3.0/cql/cql_using/use_set_t.html
- **List**: When the order of elements matters, which may not be the natural order dictated by the type of the elements, use a list. Also , use a list when you need to store same value multiple times. List values are returned according to their index value in the list, whereas set values are returned in alphabetical order, a ssuming the values are text. Using the list type you can add a list of preferred places for each user in a users table, and then query the database for the top x places for a user.
  Details Usage: http://www.datastax.com/documentation/cql/3.0/cql/cql_using/use_list_t.html
- **Map**: As its name implies, a map maps one thing to another. A map is a name and a pair of typed values. Using the map type, you can store timestamp-related information in user profiles. Each element of the map is internally stored as one Cassandra column that you can modify, replace, delete, and query. Each element can have an individual time-to-live and expire when the TTL ends.
  Details Usage: http://www.datastax.com/documentation/cql/3.0/cql/cql_using/use_map_t.html

Expiring columns

Data in a column can have an optional expiration date called TTL (time to live). Whenever a column is inserted, the client request can specify an optional TTL value, defined in seconds, for the data in the column. TTL columns are marked as having the data deleted (with a tombstone) after the requested amount of time has expired. After columns are marked with a tombstone, they are automatically removed during the normal compaction (defined by the gc_gracegc_grace_seconds) and repair processes. If you want to change the TTL of an expiring column, you have to re-insert the column with a new TTL. In Cassandra, the insertion of a column is actually an insertion or update operation, depending on whether or not a previous version of the column exists. This means that to update the TTL for a column with an unknown value, you have to read the column and then re-insert it with the new TTL value. TTL columns have a precision of one second, as calculated on the server. Therefore, a very small TTL probably does not make much sense. Moreover, the clocks on the servers should be synchronized; otherwise reduced precision could be observed because the expiration time is computed on the primary host that receives the initial insertion but is then interpreted by other hosts on the cluster. An expiring column has an additional overhead of 8 bytes in memory and on disk (to record the TTL and expiration time) compared to standard columns.

Counter columns

A counter is a special kind of column used to store a number that incrementally counts the occurrences of a particular event or process. For example, you might use a counter column to count the number of times a page is viewed. Counter column tables must use Counter data type. Counters may only be stored in dedicated tables. After a counter is defined, the client application then updates the counter column value by incrementing (or decrementing) it. A client update to a counter column passes the name of the counter and the increment (or decrement) value; no timestamp is required. Internally, the structure of a counter column is a bit more complex. Cassandra tracks the distributed state of the counter as well as a server-generated timestamp upon deletion of a counter column. For this reason, it is important that all nodes in your cluster have their clocks synchronized using a source such as network time protocol (NTP). Unlike normal columns, a write to a counter requires a read in the background to ensure that distributed counter values remain consistent across replicas. Typically, you use a consistency level of ONE with counters because during a write operation, the implicit read does not impact write latency.

Indexing

An index provides a means to access data in Cassandra using attributes other than the partition key. The benefit is fast, efficient lookup of data matching a given condition. The index indexes column values in a separate, hidden table from the one that contains the values being indexed. Cassandra has a number of techniques for guarding against the undesirable scenario where a data might be incorrectly retrieved during a query involving indexes on the basis of stale values in the index.

- **When to use an index**
  Cassandra's built-in indexes are best on a table having many rows that contain the indexed value. The more unique values that exist in a particular column, the more overhead you will have, on average, to query and maintain the index. For example, suppose you had a playlists table with a billion songs and wanted to look up songs by the artist. Many songs wil l share the same column value for artist. The artist column is a good candidate for an index.
- **When not to use an index**
  a. On high-cardinality columns because you then query a huge volume of records for a small number of results.
  b. In tables that use a counter column.
  c. On a frequently updated or deleted column. Cassandra stores tombstones in the index until the tombstone limit reaches 100K cells. After exceeding the tombstone limit, the query that uses the indexed value will fail.
  d. To look for a row in a large partition unless narrowly queried.