Data Model

Table of Contents

Context & Scope

ECCO Sko A/S, a Danish company specializing in footwear and leather accessories, was established in 1963 by Karl Toosbuy in Bredebro, Denmark. ECCO shoes and leather products are available in over 100 countries, with a strong presence in markets across Asia, Eastern and Central Europe, Canada, South America, and the United States. They distribute their merchandise through a network of over 2,250 exclusive brand stores, as well as independent retailers and online sales channels.

Ecco is currently in the process of transitioning their table data from Azure tables to Astra. Due to limitations in partition size within Azure tables, the decision was made to adopt Astra as their new database solution. This document focuses on reviewing a specific part of the shipping data model within this migration context.

General Recommendations

Data Modelling Guidelines

Data modeling is a process used to analyze, understand and organize data for efficient product use. Unlike relational database systems where the model can be designed independent of the queries, in Cassandra, we must understand the query usage of the data before we can address the data model.

Defining a good data model is the hardest part of using Cassandra.

Cassandra is a distributed storage system

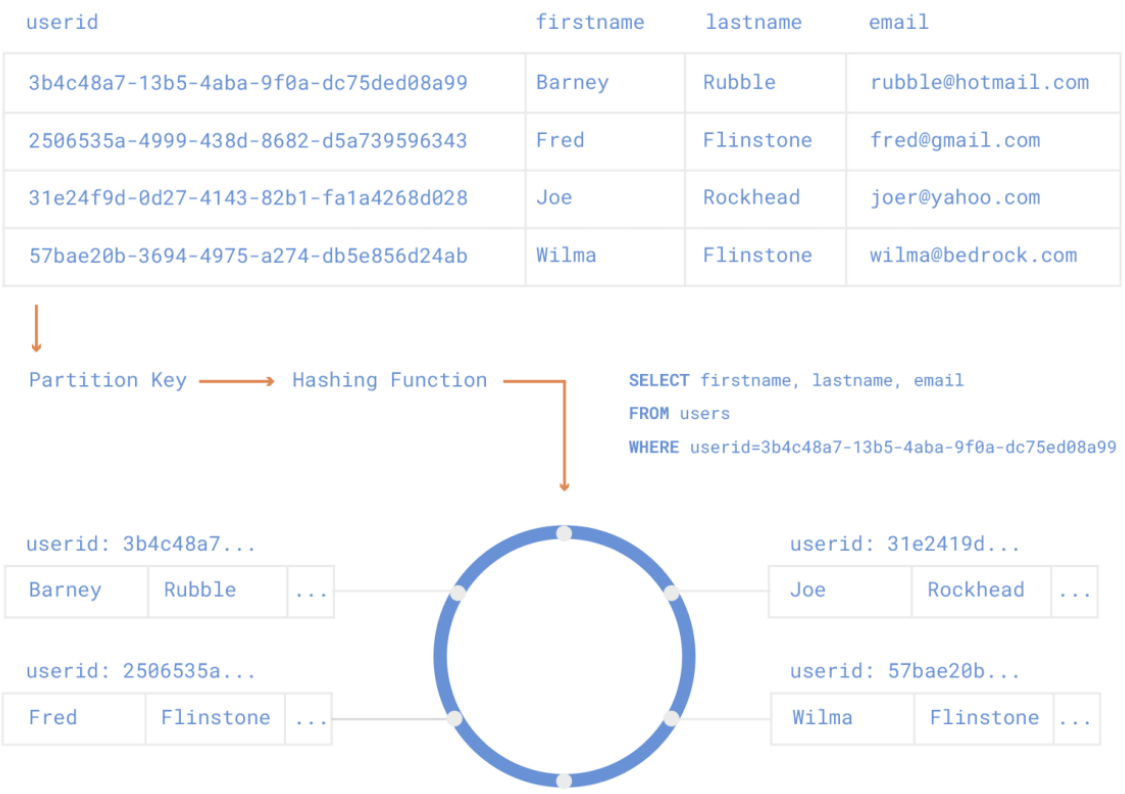
Understanding how Cassandra stores data is essential in developing a good data model. DataStax recommends reading Apache Cassandra™ Architecture white paper[1] to get a deeper understanding of Cassandra's internal architecture.

In a nutshell, Cassandra's clusters have multiple nodes. Data can be stored redundantly across the nodes according to a keyspace-level configurable replication factor.

Nodes in a Cassandra cluster are assigned hashes (tokens) which are used to identify where data is stored. All nodes know which tokens are assigned to which nodes. A consistent Hashing Function (algorithm) is used to map a table partition to a token in Cassandra.

To retrieve a row from Cassandra's cluster, it is necessary to provide a partition key. Cassandra will use the partition to calculate the equivalent token and then determine which node owns the token to retrieve the stored data. Queries providing partition keys are extremely fast because Cassandra can immediately determine the host holding the required data.

[1] https://www.datastax.com/resources/whitepaper/apache-cassandratm-architecture



Spread data evenly around the cluster

Spreading data as evenly as possible across cluster nodes is recommended for Cassandra to work optimally. As such partitions of data need to be defined such that they will be distributed all around the cluster, rather than be grouped on a handful of nodes.

Minimize the number of partitions to read

Partitions are groups of rows sharing the same partition key. When you read data from Cassandra, you want to read as few partitions as possible. It is always more expensive to read from multiple partitions than from a single one.

Avoid using Truncate over the tables

Truncate takes a long time to run and effectively requires agreement from all nodes, be mindful of this in your performance guarantees. Truncate will act as if it has CONSISTENCY set to ALL no matter what CONSISTENCY level is set.

Structure tables around the application query

The way to minimize partition reads is to model your data to fit your queries. A single table should be able to answer a single query efficiently. Avoid modeling around data relationships and objects. This can be done using the following steps.

Step 1. Determine what information the application needs to return to the user

In the relational database world, it is perfectly normal to start with the data model, thinking about the data items that need to be stored and how they relate to one another. With Cassandra, just the opposite is recommended.

The best practice is to start with the application workflow also known as “query-first design”.

We do not know how data will be stored yet and this is absolutely fine. The first step is to know what types of queries the database will need to support. This will be done using the application workflow.

Step 2. Determine the queries based on the information the application needs

As a rule of thumb a single action in the application should translate to a single query to Cassandra. In this phase, developers can think through the sequence of tasks required, mock-up what each screen will look like, and decide what data will be required at each stage. It is as well possible to build an Entity-relationship model[2] in this step.

Try to determine exactly what queries you need to support. This can include a lot of considerations that you may not think of at first. For example, you may need to think about:

- Grouping by an attribute
- Ordering by an attribute
- Filtering based on some set of conditions
- Enforcing uniqueness in the result set

Step 3. Build the tables

Next step is to think about how the table should be structured to support queries required by the application. We will decide how Cassandra will store data.

---

[2] https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model

Cassandra tables can be grouped into two distinct categories: tables with single-row partitions and multi-rows partitions.

A table has a single-row partition when the primary key is also the partition key. These tables should be named based on the entity for clarity (like customer or shop).

Multi-row partition tables have primary keys composed of partition keys and clustering columns. Remember that Cassandra doesn't support joins, so developers need to structure tables to support queries that relate to multiple data items. It's a good idea to give tables meaningful names so that people examining the schema can understand the purpose of different tables (like meta_by_user_id).

Plan for data growth

Your data model might make sense at the time being but in the future, when your data will double in size it might not. It is a good idea to anticipate the future and the coming growth to avoid introducing bottlenecks. Data model should be designed for the cluster to scale (in size).

List Collection Type

Lists have a number of critical limitations

Lists are ordered collections that can contain many of the same element types. Whilst they can be convenient to use as part of a table definition they have a number of limitations.

- Server-side race condition when simultaneously adding and removing elements.
- Setting and removing an element by position and removing occurrences of particular elements incur an internal read-before-write.
- Prepend or append operations are non-idempotent.
- There is an additional overhead to store the index of the elements for which the UUID is used (16 bytes per element).

A list lacks the ability to be simultaneously updated. Consider the case where an element is removed from the head on one server. In this case, if an element is also added to the head on another server at the same time, there will be a conflict.

In case of a failure for a prepend or append operation, the state of the list is unknown. Specifically, it is unknown if the operation was able to alter the list when the failure occurred. Retrying the operation may result in duplicate elements; ignoring the failure without a retry may result in loss of data.

They can be replaced when element ordering or duplication is unnecessary.

If there is no requirement to keep elements in a specific order or have elements with duplicate values, then the list type should be replaced.

For lists with a very low number of elements (10 or less), change them to a clustering column in the table. In this case, the list becomes a column of the same type as the elements it holds, and is defined as a clustering key in the table.

For lists with a small number of elements (under 100), they can be replaced with a set type in the table. In this case, the list is changed to a set that stores the same element type as the list. The preferred alternative is to remove the list from the table and create a dedicated table for it.

For lists with a large number of elements (100 or more), they will need to be removed from their respective tables. A dedicated table will need to be created for each one. This solution is preferred for this case where there are a large number of elements.

***Recommendation**: We recommend replacing all list collection type instances in the schema definition.*

Map and Set Types

Non-frozen Maps and Sets carry additional overhead

Maps and Sets are Conflict-free replicated data types[3] (CRDTs). This means that simultaneous operations on the collections on different servers will converge to the same value on both servers. For instance, consider the case where a map contains elements A and B. Removing A on one node server and B on another server will result in both servers thinking the map contains no elements.

Collection types such as maps and sets can be either defined as frozen or non-frozen. By default, a collection is non-frozen unless the frozen keyword is specified.

frozen: The whole content of the collection is serialized and stored as one value. This type solves some of the drawbacks of non-frozen collections but comes at the expense of updates to individual elements of a frozen collection. In addition, when two or more clients are writing to the same table unless Lightweight Transactions are used, there is no guarantee of the write ordering for each replica of a frozen collection.

[3] https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

non-frozen: The collection is stored as a set of individual elements in separate cells. This is the default state for a collection. There are a number of drawbacks to consider when using non-frozen collections.

- Additional overhead for keeping metadata (write timestamp, TTL, etc) of individual elements.
- When an INSERT or full UPDATE occurs, a tombstone is created even if there was no data that previously existed. For example, replacing the value of a column with another value results in a tombstone marker to prevent possible overlap with previous data.
- When reading a column with a collection type, its whole content is returned. Large numbers of elements can result in performance problems when accessing data.
- Where individual elements were updated after insertion, performance can degrade. This is because column updates could be spread between multiple SSTables. To read back the values, all the SSTables that contain the column update need to be read to reconstruct the final column value.

Use frozen maps when their data is immutable

Unless there is a necessity to update individual elements in a map, or if it is storing immutable data, use the frozen variant. This is so Cassandra will automatically serialize content into a single binary value. It will remove the overhead of using a collection type at the expense of the ability to perform a partial update of the value.

***Recommendation**: We recommend using the frozen variant of the map collection types where data is immutable.*

User Defined Types

Cassandra allows the creation of User Defined Types (UDTs). This type allows you to group related information together and use it as a single entity. From a data model analysis point of view, you can apply the same rules as for collections:

- Use frozen UDTs where possible.
- For non-frozen UDTs keep the number of columns to a minimum.

One of the problems with UDTs arises from a schema evolution standpoint. While it is possible to add columns to UDTs, it is impossible to remove them.

UDTs should only be used sparingly and when absolutely necessary. Otherwise, it is preferable to structure the data as regular columns in the table. Alternatively, store the UDT as a text blob and perform serialization and deserialization of UDT data inside the application.

UDTs can be nested inside other UDTs or as elements in the collections. Caution needs to be taken when doing this. If too many elements exist in a collection or there are too many nested UDTs, then the size of the mutation will increase. If the maximum mutation size limit is reached, operations involving the table will fail.

Recommendation: *We recommend against using non-frozen UDTs for performance reasons. Latencies and heap pressure will be much lower when using a JSON representation of the objects stored as a text column.*

CQL Types

Cassandra CQL language defines a number of different data types[4]. It is recommended to use the right type for each column in order to optimize storage space on disk and data integrity. Serialization and compression will be much more efficient with fixed-length data types compared to types that can grow unbounded.

For example, UUID on disk will use 16 bytes whereas text will use 2 to 4 bytes per character.

Client-side data processing and manipulation are reduced if the correct data type is used. Removing the need to manipulate data will avoid inadvertently compromising its integrity.

For example, truncating the precision of a number during a conversion.

Recommendation*: We recommend using the correct data type for each column to optimize storage and data integrity. We recommend always using the UUID CQL type for UUID data.*

Bucketing

Bucketing is a strategy that lets us control how much data is stored in each partition as well as spread writes out to the entire cluster.

[4] https://cassandra.apache.org/doc/latest/cassandra/cql/types.html

- **Time Bucket**

The idea is to break our partitions into smaller ones based on the time window. The ideal size is going to keep partitions under 100MB. We can create a time bucket for a day/week and add that column to the partition. It is important to make sure to use the time bucket in the queries.

- **Multi partition with Time and Bucket Numbers**

Multi partition with day and bucket numbers which means that you can parallel the load by using a bucket number(1..5) for each day and you will use that bucket number to read the data for each day and bucket number in range 1..5. You can find more information in this post.

Time to Live

The default retention time can be set at the table level, and it determines how long data is kept in the database before it is automatically deleted. The exact value of the retention time will depend on the specific requirements of your use case .

Columns and tables support an optional expiration period called TTL (time-to-live); TTL is not supported on counter columns. The TTL is defined in seconds. Data expires once it exceeds the TTL period and is then marked with a tombstone. Expired data continues to be available for read requests during the grace period, see gc_grace_seconds. Normal compaction and repair processes automatically remove the tombstone data. You can find more information here.

In AstraDB, you can set the retention time using the Time-To-Live (TTL) feature, which allows you to specify a number of seconds that data should be kept in the database before it is automatically deleted. For example, if you set the TTL value to 3600 seconds, data will be automatically deleted from the database after one hour.

It's important to note that once data has been deleted from AstraDB, it cannot be recovered, so it's important to choose an appropriate retention time that balances your need for data retention with your storage constraints. Additionally, you should also be mindful of the impact of data deletion on the performance of your database and plan accordingly.

While inserting the data for that datapoint, you can choose different TTLs for each datapoint:

```
Ex: INSERT INTO cycling.calendar ( race_id, race_name, race_start_date, race_end_date) VALUES (
 200, 'placeholder', '2015-05-27', '2015-05-27')
USING TTL 200;
```

Insert,Update and Upsert

Because Cassandra uses an append model, there is no fundamental difference between the insert and update operations. If you insert a row that has the same primary key as an existing row, the row is replaced. If you update a row and the primary key does not exist, Cassandra creates it.

For this reason, it is often said that Cassandra supports upsert, meaning that inserts and updates are treated the same.

Consistency Level

Supported consistency levels: Reads: Any supported consistency level is permitted.
Single-region writes: *LOCAL_QUORUM* and *LOCAL_SERIAL*
Multi-region writes: *EACH_QUORUM* and *SERIAL*

Serial is for LWT (Light Weight Transactions)/CAS(Compare and Set) operations so we recommend using *LOCAL_QUORUM* for single_region or *EACH_QUORUM* for multi-region writes.

By adjusting the desired consistency level, you can trade off consistency for performance, and vice versa. For example, if you choose a consistency level of ONE, you may experience faster write and read times, but you also risk the possibility of reading stale data. If you choose a consistency level of ALL, you'll have stronger consistency guarantees, but the write and read times may be slower.

It's important to choose the appropriate consistency level based on the specific requirements of your use case, such as the volume and velocity of data, the frequency of updates, the importance of data accuracy, and the required performance.

Compaction Strategy

Astra has UnifiedCompactionStrategy which is a more efficient compaction strategy that combines ideas from STCS (SizeTieredCompactionStrategy), LCS (LeveledCompactionStrategy), and TWCS (TimeWindowCompactionStrategy) along with token range sharding. This all-inclusive compaction strategy works well for all use cases.

Bulk Data Load

DataStax Bulk Loader can be used to load bulk data as csv or json files.