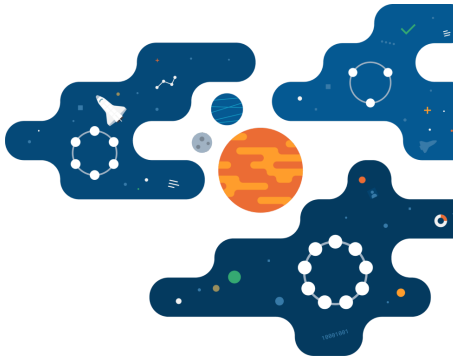
WHITEPAPER



Data Modeling in Apache Cassandra™

Five Steps to an Awesome Data Model

For global-scale applications, Apache Cassandra is the favorite choice among architects and developers.

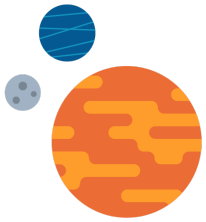
It offers many advantages including zero-downtime high availability, zero lock-in with easy data portability, and global scale with extreme performance. Today, Cassandra is among the most successful NoSQL databases. It is used in countless applications from online retail to internet portals to time-series databases to mobile application backends.

Having a well-designed data model is essential to meeting application performance and scalability goals. In this paper, aimed at technical people experienced with relational databases, we discuss five useful steps to realizing a high-quality data model for your Cassandra application.

DataStax

01

Data Modeling in Cassandra



Why data modeling is critical

Having the correct data model is important for any application, but it becomes especially critical for applications that need zero downtime, zero lock-in, and global scale. An application may work with thousands of records and a few hundred concurrent users, but what happens when record and user counts are in millions or billions?

Regardless of the database, if the data model isn't right, or doesn't suit the underlying database architecture, users can experience poor performance, downtime, and even data loss or data corruption. Fixing a poorly designed data model after an application is in production is an experience that nobody wants to go through. It's better to take some time upfront and use a proven methodology to design a data model that will be scalable, extensible, and maintainable over the application lifecycle.

Differences between Cassandra and relational databases

Most readers will be familiar with relational databases such as Oracle®, MySQL®, and PostgreSQL®. Before jumping into data modeling with Cassandra, it's helpful to explain a few differences between Cassandra and relational databases. High-level relational databases prioritize space over response time—one copy of data stored, but many joins that slow performance in order to pull a result set out of it. Cassandra prioritizes response time over space—a few copies of data and a real-time return of a result set.

- **Zero-downtime, high availability** – In a relational database, high availability has to be carefully configured and tested for each system, which delays the time it takes your application to get to production. The cloud-native architecture of Cassandra allows for your data to sit on premises, in any cloud, or multiple clouds with built-in data availability and out-of-the-box direction of traffic to available nodes.
- **Zero lock-in, highly portable data** – In a relational database, developers and DBAs rely on ETL to move data around which takes a significant amount of time when you have massive amounts of data. The time it takes to move data means it can be unavailable to your system, and often applications running on relational databases have scheduled maintenance windows to handle porting data. The cloud-native architecture of Cassandra powers highly portable data out of the box, allowing developers to move faster when building applications. With Cassandra, there's no need for scheduled maintenance windows, even during database upgrades.

- **Global scale, extreme performance worldwide** – In a relational database, to achieve distributed data, geo-local low latencies across multiple geographies is a challenge and when you layer on top of that the need to serve out extremely high throughput the relational databases fall over. Relational databases at their core were not designed for this level of customer interaction at a true global scale. In Cassandra, the performance of high writes at global scale is a major reason enterprises select Cassandra for delivering applications with lightning fast customer experiences globally.

- **In Cassandra, denormalization is expected** – With relational databases, designers are usually encouraged to store data in a normalized[1] form to save disk space and ensure data integrity. In Cassandra, storing the same data redundantly in multiple tables is not only expected, but it also tends to be a feature of a good data model.

- **In Cassandra, writes are (almost) free** – Owing to Cassandra's architecture, writes are shockingly fast compared to relational databases. Write latency may be in the hundreds of microseconds and large production clusters can support millions of writes-per-second.[2]

- **No joins** – Relational database users assume that they can reference fields from multiple tables in a single query, joining tables on the fly which can slow performance of your application. With Cassandra, joins don't exist, so developers structure their data model to provide equivalent functionality.

- **Developers need to think about consistency** – Relational databases are ACID compliant (Atomicity, Consistency, Isolation, Durability), characteristics that help guarantee data validity with multiple reads and writes. Cassandra supports the notion of tunable consistency, balancing between Consistency, Availability, and Partition Tolerance (CAP)[3]. Cassandra gives developers the flexibility to manage trade-offs between data consistency, availability, and application performance when formulating queries.

- **Indexing** – In a relational database, queries can be optimized by simply creating an index on a field. While secondary indexes exist in Cassandra, they are not a "silver bullet" as they are in a relational database management system (RDBMS). In Cassandra, tables are usually designed to support specific queries, and secondary indexes are useful only in specific circumstances.

---

[1] Database normalization is the process of structuring a relational database to reduce data redundancy and improve data integrity – details are available at https://en.wikipedia.org/wiki/Database_normalization

[2] Cassandra delivers one million writes per second at Netflix – https://medium.com/netflix-techblog/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e

[3] How Apache Cassandra™ Balances Consistency, Availability, and Performance – https://www.datastax.com/2019/05/how-apache-cassandra-balances-consistency-availability-and-performance

How Cassandra stores data

Understanding how Cassandra stores data is essential to developing a good data model. Readers wishing to get a better understanding of Cassandra's internal architecture can read the DataStax Apache Cassandra™ Architecture whitepaper available at ***https://www.datastax.com/resources/whitepapers/apache-cassandra-architecture.***

Cassandra clusters have multiple nodes running in local data centers or public clouds. Data is typically stored redundantly across nodes according to a configurable replication factor so that the database continues to operate even when nodes are down or unreachable.

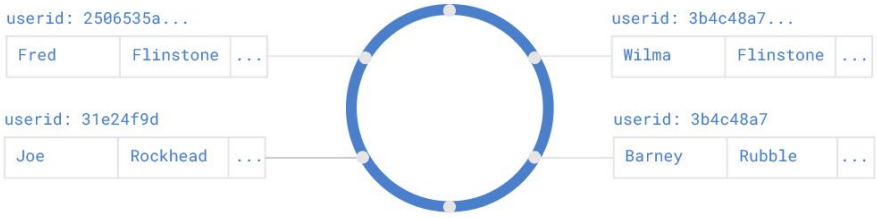
Tables in Cassandra are much like RDBMS tables. Physical records in the table are spread across the cluster at a location determined by a partition key. The partition key is hashed to a 64-bit token that identifies the Cassandra node where data and replicas are stored.[4] The Cassandra cluster is conceptually represented as a ring, as shown in Figure 1, where each cluster node is responsible for storing tokens in a range.

Queries that look up records based on the partition key are extremely fast because Cassandra can immediately determine the host holding required data using the partitioning function. Since clusters can potentially have hundreds or even thousands of nodes, Cassandra can handle many simultaneous queries because queries and data are distributed across cluster nodes.

userid	firstname	lastname	email
57bae20b-3694-4975-a274-db5e856d24ab	Wilma	Flinstone	fred@gmail.com
3b4c48a7-13b5-4aba-9f0a-dc75ded08a99	Barney	Rubble	rubble@hotmail.com
57bae20b-3694-4975-a274-db5e856d24ab	Wilma	Flinstone	wilma@bedrock.org
31e24f9d-0d27-4143-82b1-fa1a4268d028	Joe	Rockhead	joey@yahoo.com

Partition Key ——— Hashing Function

```
SELECT firstname, lastname, email
FROM users
WHERE userid=3b4c48a7-13b5-4aba-9f0a-dc75ed08a99
```



[4] This is a bit of an over-simplification. Recent versions of Cassandra support the notion of vnodes where each cluster node is responsible for multiple token ranges to better distribute data. https://docs.datastax.com/en/dse/5.1/dse-arch/datastax_enterprise/dbArch/archDataDistributeVnodesUsing.html. The ring is still a useful way to conceptualize how Cassandra stores data however

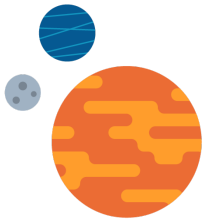
Partition keys can be single columns or can be composed of multiple columns. Cassandra also supports clustering columns (discussed shortly) that control how data records are grouped and organized within each partition. Records in Cassandra are stored as lists of key-value pairs where the column name is the key.

High-level goals for a Cassandra data model

- **Spread data evenly around the cluster** – For Cassandra to work optimally, data should be spread as evenly as possible across cluster nodes. Distributing data evenly depends on selecting a good partition key.
- **Minimize the number of partitions to read** – When Cassandra reads data, it's best to read from as few partitions as possible since each partition potentially resides on a different cluster node. If a query involves multiple partitions, the coordinator node responsible for the query needs to interact with many nodes, thereby reducing performance.
- **Anticipate how data will grow, and think about potential bottlenecks in advance** – A particular data model might make sense when you have a few hundred transactions per user, but what would happen to performance if there were millions of transactions per user? It's a good idea to always "think big" when building data models for Cassandra and avoid knowingly introducing bottlenecks.

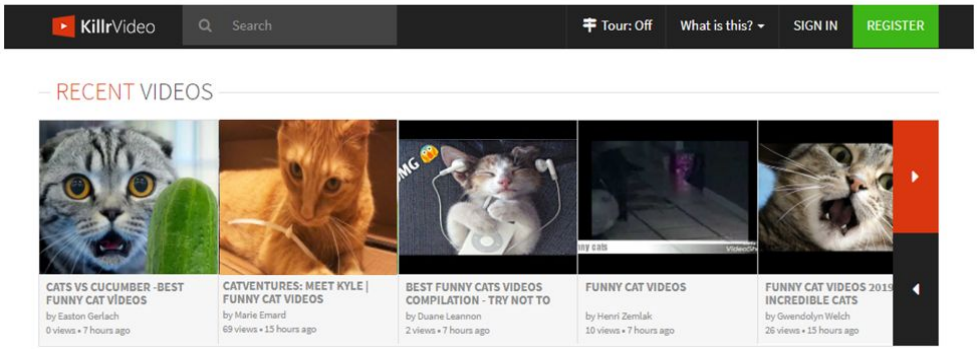
02

Five Steps to an Awesome Data Model



A sample application

To illustrate how to build Cassandra applications, DataStax offers a reference application called KillrVideo[5], a fictitious company operating a video service like YouTube™.



When discussing data modeling, it's helpful to focus on a concrete example. In the sections that follow, data modeling will be discussed in the context of the KillrVideo application.

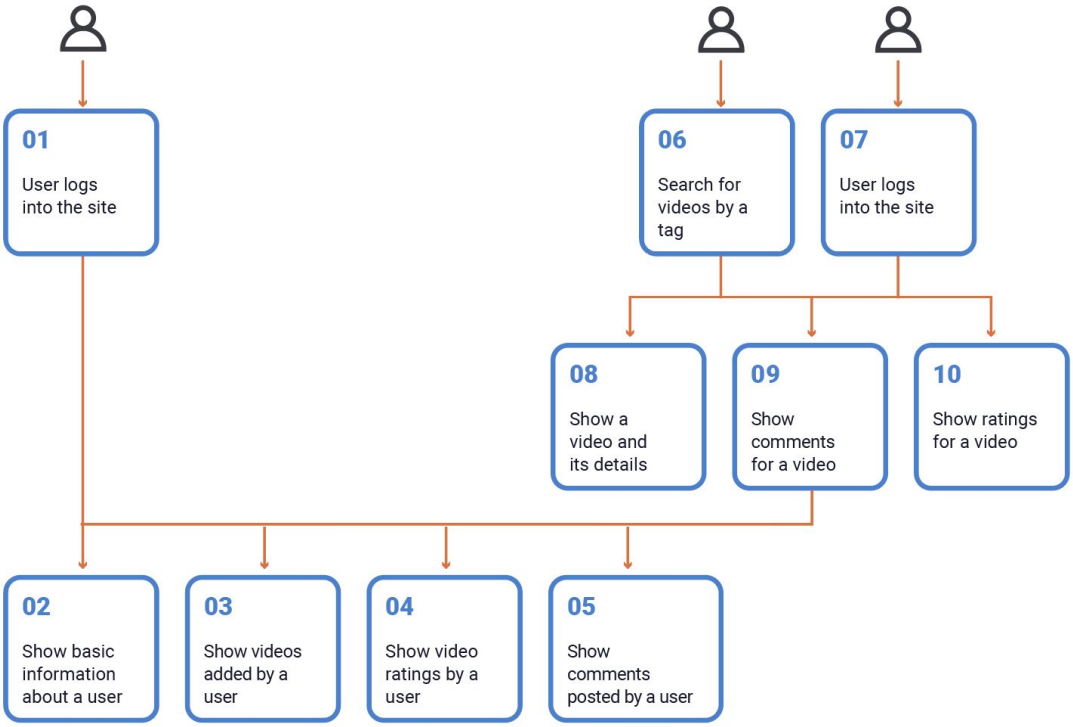
In KillrVideo, users perform activities such as creating profiles, submitting, tagging, and sharing videos, searching for and retrieving videos submitted by others, viewing them, rating them, and commenting on them. KillrVideo serves as a useful example because it needs to operate at global scale, supporting millions of users and transaction volumes that would be all but impossible to achieve using a relational database.

Step 1: Build the application workflow

When building applications using relational databases, developers often 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 - start with the user experience. The best practice is to start with the application workflow; an approach referred to as "query-first design."

Before thinking about how data will be stored, designers need to know what types of queries the database will need to support. Figure 3 presents a simplified application workflow for KillrVideo.com.

[5] Details about KillrVideo including source code and design documents are available at http://killrvideo.com

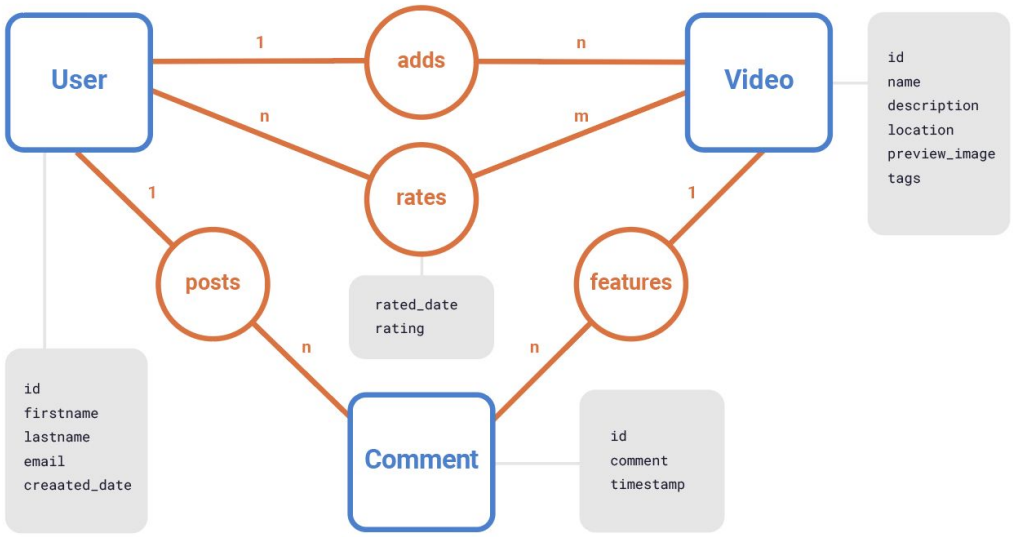


The sequence of workflow steps matters because it helps us determine what data is available and required for each query. For example, before we can show basic information about a user (step 2 above), a userid is required. The user first needs to log in to the site (step 1) supplying an email address and password in exchange for the required userid. A userid might also be obtained by searching for a video (steps 6 or 7), showing comments for a video (step 9), and looking up details about the user that commented. Similarly, before the application can display details about a video (step 8) the application needs a videoid obtained by selecting from a list of the latest videos (step 7) or by searching videos by tag (step 6).

Step 2: Model the queries required by the application

Even at the design stage, 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.

Figure 4 shows a simplified entity relationship diagram (ERD) for the KillrVideo application. The application needs to be able to keep track of entities such as users, videos, and comments. Users can perform activities such as adding videos, rating videos, and posting comments. Users can comment on multiple videos, and each video can have multiple user comments associated, but there is only one owner of each video.



It's a good idea to iterate between the application workflow and ERD, updating both as new data items and relationships required by the application are identified. Once developers have a clear idea of the application workflow and the key data objects required, it's possible to start identifying the queries that the application needs to support. A diagram showing key queries and how they relate to data domains is shown in Figure 5.

Step 3: Create the tables

The next step is to think about how tables should be structured to support queries required by the application. At this stage, we are getting more concrete about how Cassandra will store data.

In Cassandra, tables can be grouped into two distinct categories:

- **Tables with single-row partitions** – These types of tables have primary keys that are also partition keys. They are used to store entities and are usually normalized. Users, comments, and videos are examples of entities in our application. These tables should be named based on the entity for clarity (i.e., users or videos).
- **Tables with multi-row partitions** – These types of tables have primary keys that are composed of partition and clustering keys. They are used to store relationships and related entities. 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 (i.e., comments_by_user or videos_by_tag).

A Cassandra Query Language (CQL) schema for the single-row partition videos table is shown below. The videos table includes columns, such as the user that uploaded the video, a description, details about where the video was taken, a timestamp, and a preview_image. A video can have multiple associated tags, so we use a set data type in Cassandra known as a collection (discussed shortly) to store a variable number of tags per record.

```
CREATE TABLE videos (
    videoid uuid,
    userid uuid,
    name text,
    description text,
    location text,
    location_type text,
    preview_image_location text,
    tags set<text>,
    added_date timestamp,
    PRIMARY KEY (videoid)
);
```

A table with multi-row partitions supports the query that looks up the latest videos (step 7 in the workflow in Figure 3) is shown below.

```
CREATE TABLE IF NOT EXISTS latest_videos (
    yyyymmdd text,
    added_date timestamp,
    videoid uuid,
    userid uuid,
    name text,
    preview_image_location text,
    PRIMARY KEY (yyyymmdd, added_date, videoid)
) WITH CLUSTERING ORDER BY (added_date DESC, videoid ASC);
```

The latest_videos table illustrates what is meant by "query-first design." The application will need to query the most recently uploaded videos every time a user visits the KillrVideo homepage, so this query needs to be very efficient. On a global video sharing service, the latest dozen or so videos are almost guaranteed to have been uploaded on the same day (or in some cases the day before). The table stores a timestamp (indicating the precise upload time) but rather than using the timestamp as a partition key, the developer stores a "yyyymmdd" value derived from the timestamp uniquely representing the day that the video was uploaded. By making yyyymmdd the partition key, videos uploaded on the same day will be stored in the same partition, making lookups fast and efficient.[6]

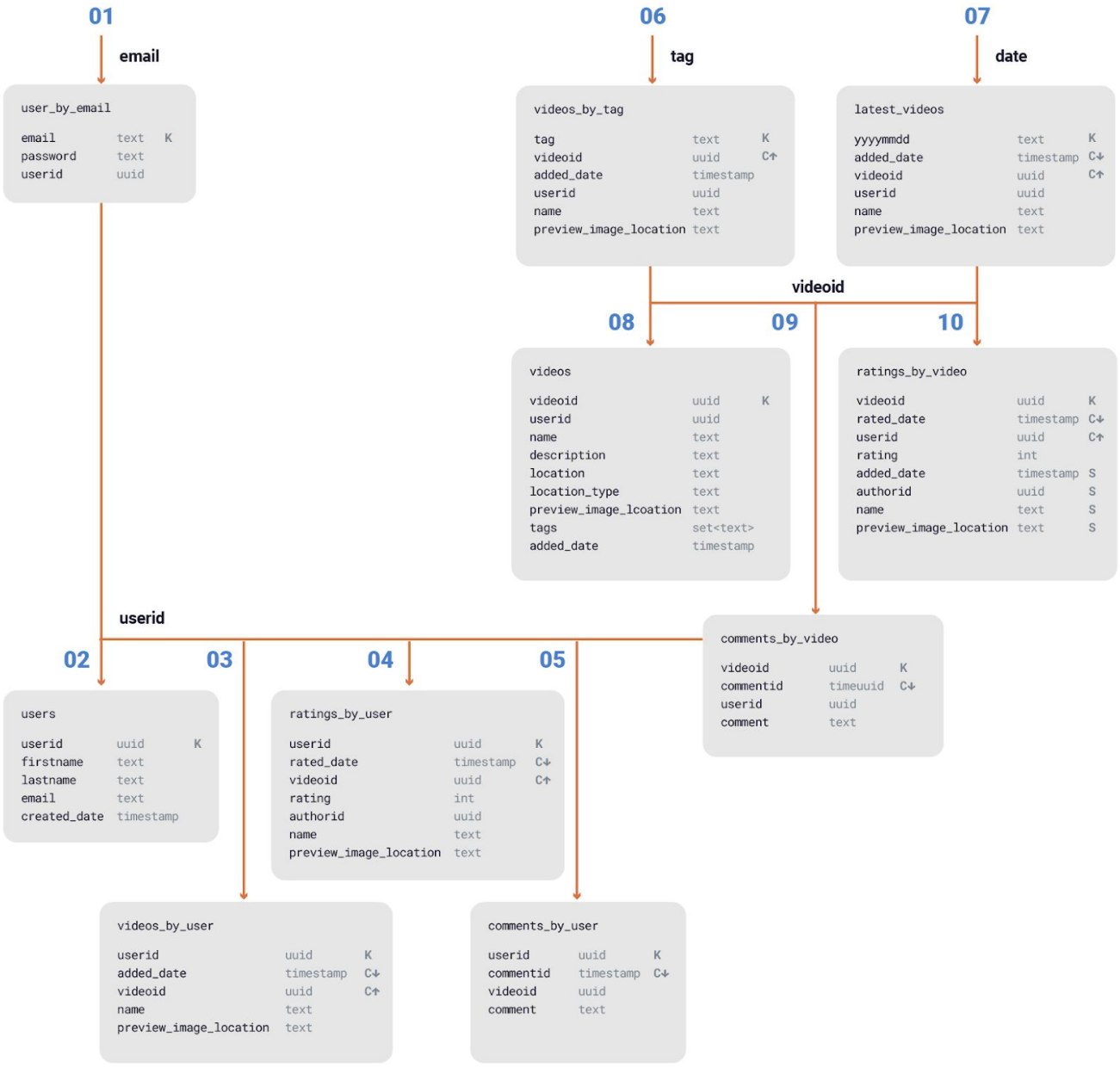
Additional clustering columns (added_date and videoid) specify how records are grouped and ordered within each partition. By designing the table in this fashion, queries will touch only one partition for the current day and possibly another partition for the day before. There will be no need to sort, filter, or otherwise post-process query results. This level of optimization and efficiency helps explain how Cassandra can support applications with enormous numbers of queries over very large data sets.

Chebotko Diagrams

A good tool for mapping the data model that supports an application is known as a Chebotko diagram[7] shown in Figure 6. While entity-relationship diagrams are useful in building a conceptual data model, a Chebotko diagram helps develop the logical and physical data models required to support the application.

[6] This is a design choice. If millions of uploads per day were anticipated, a better choice of key might be "yyyymmddhh" to store fewer records per partition. This is a good example of the trade-offs that architects need to consider when building a data model.

[7] Chebotko diagrams are named after Dr. Artem Chebotko, Solution Architect at DataStax.



The Chebotko diagram captures the database schema, showing table names, partition key columns (K), clustering key columns (C) and their ordering (ascending or descending), static columns (S), and regular columns with data types. The tables are organized based on the application workflow to support specific workflow steps and application queries. Let's take a look at a couple of tables more closely.

During the sign-in process, a user provides an email address and password for authentication purposes. The user_by_email table is optimized to quickly retrieve records by email and return the encrypted password for validation and the userid required for downstream queries.

Next, we may need to retrieve all comments posted by the user. The comments_by_user table supports such a query where we retrieve an ordered list of comments made by a logged-in user. Since comments will be retrieved "by user," the userid is defined as the partition key to ensure that comments made by the same user will always reside in the same partition. The commentid column is defined as a clustering key. Since commentid is of type timeuuid (a universally unique identifier generated based on a timestamp and node MAC address), it ensures both uniqueness and ordering of rows in a partition. User comments will be stored and retrieved in descending order from the same partition showing the most recent comments first. Since the application will need to display when a comment was made, the toTimestamp() function can be used to convert the commentid stored as a timeuuid value to a timestamp as shown.

```
SELECT commentid, toTimestamp(commentid) AS timestamp, videoid, comment
FROM comments_by_user
WHERE userid = 2506535a-4999-438d-8682-d5a739596343
LIMIT 3;
```

```
commentid | timestamp                       | videoid   | comment
-----------+---------------------------------+-----------+-------------------------------------------------------
8e8ea5c2… | 2019-08-02 21:57:44.418000+0000 | af9ff655… | I laughed I cried; it was better than Cats the musical
0e3956ac… | 2019-08-02 21:46:59.614000+0000 | a0a60a16… |                                                Cute cat!
94c0330a… | 2019-08-02 21:36:26.319000+0000 | 7dfeb453… |                            Can't wait for the sequel!
```

Denormalization is Expected

People experienced with relational databases may be surprised by how data elements are stored redundantly across tables. For example, in Figure 6, a pointer to the video preview image (preview_image_location) appears in six different tables. There are limits to how many redundant copies we want to maintain, but remember that in Cassandra, denormalization is expected and it is a good trade-off to achieve performance and scalability goals:

- Modern storage is inexpensive. For small data items, it's less of a concern to store data redundantly than it once was.
- Adding videos and generating preview images is a relatively infrequent activity in the application. There will be thousands of queries that require the presence of a preview image for every query that inserts or updates an image preview. It makes sense to optimize for the most frequent queries.

- While it's a small burden for the developer to synchronize data across six tables, remember that writes in Cassandra are extremely fast, so there is a minimal performance penalty. If we can speed queries with denormalization, some data redundancy is a small price to pay. Cassandra has a batch feature (discussed shortly) that is useful when coordinating multiple writes to denormalized tables.

Step 4: Get the primary key right

Tables in Cassandra have a primary key. The primary key is made up of a partition key, followed by one or more optional clustering columns that control how rows are laid out in a Cassandra partition. Getting the primary key right for each table is one of the most crucial aspects of designing a good data model.

In the latest_videos table, yyyymmdd is the partition key, and it is followed by two clustering columns, added_date and videoid, ordered in a fashion that supports retrieving the latest videos.

```
    ..
    PRIMARY KEY (yyyymmdd, added_date, videoid)
) WITH CLUSTERING ORDER BY (added_date DESC, videoid ASC);
```

A partition key in Cassandra can be made up of multiple columns. For example, if we were to have a query that finds all videos based on a tag and location, we would need to use a composite partition key consisting of both the tag and location as shown.

```
..
    PRIMARY KEY ((tag, location), videoid)
..
```

Creating Unique Keys

For most applications, rows stored in a table need unique identifiers. Good examples are customer IDs, order IDs, and transaction IDs. If keys aren't unique, it's easy to accidentally over-write rows—disastrous for most applications. Relational databases often use simple auto-incrementing integers to assign unique keys to records, but this approach isn't practical in a distributed system like Cassandra.

To address this problem of unique keys, Cassandra supports universally unique identifiers[8] (UUIDs) as a native data type. UUIDs are 128-bit numbers that are guaranteed to be unique within the scope of an application. Cassandra supports two types of UUIDs. TIMEUUIDs (UUID version 1) factor details like the MAC address of the node generating the UUID and embed a 60-bit timestamp to help guarantee uniqueness. Cassandra also supports random UUIDs (UUID version 4). UUIDs are represented in hexadecimal and look something like the following:

```
bddf5f6b-f1a2-4624-8005-07f8cd5456de
```

Using a UUID to identify data items that need to be unique is the best practice in Cassandra. For time-series data (a series of readings from a sensor for example) a timestamp (another native data type in Cassandra) might also be used and combined with a sensor_id to generate a unique composite key, but since a timestamp can always be extracted from a TIMEUUID, using a TIMEUUID is also a good option.

Be Careful with Custom Naming Schemes

Some developers might prefer to devise their own naming schemes to make keys easier to understand. For example, if a table stores retail locations in the United States, a designer might use a five-character key comprised of a two-character state identifier and a three-digit store number within the state. Using "AZ003" to represent the third store in Arizona is much easier to understand than "bddf5f6b-f1a2-4624-8005-07f8cd5456de".

As tempting as this is, be careful constructing unique business-friendly identifiers. What happens if the business opens stores internationally? Is "AL001" in Alabama, Alberta, or Albania? What happens if the business starts opening kiosks, and larger states suddenly have more than 1,000 outlets? What happens if the business acquires another retailer with a different naming convention? Business-friendly identifiers are a fact of life in many applications, but it's worth thinking about whether the need for human-readable identifiers is worth the effort required to maintain them.

Step 5: Use data types effectively

So far, we've discussed tables and columns without paying much attention to the best choice of data type for each column. Cassandra supports a wide variety of data types that will be familiar to most developers—BigInt, Blob, Boolean, Decimal, Double, Float, Inet (IP addresses), Int, Text, VarChar, UUID, TIMEUUID, etc. Consider the fields in the video table:

```
CREATE TABLE videos (
    videoid uuid,
    userid uuid,
    name text,
    description text,
    location text,
    location_type text,
    preview_image_location text,
    tags set<text>,
    added_date timestamp,
    PRIMARY KEY (videoid)
);
```

[8] CUUIDs are a standard way of generating unique identifiers – details at https://en.wikipedia.org/wiki/Universally_unique_identifier

There may be millions of unique videos and users, so both are identified as a universally unique identifier (UUID) a good practice for unique keys in Cassandra.

The name, description, location, and location_type are all UTF-8 strings, so they are described as type text (essentially the same as a varchar).

The preview_image_location might be a blob, but in a global application, it's more efficient to store a pointer to an object store so that the application can leverage a content delivery network (CDN) to cache images.

The added_date is a date and time that doesn't need to be unique, so we use the built-in timestamp type.

In the video tables, there can be multiple tags or keywords associated with each video, so we use a "set," one of Cassandra's useful collection types.

Collections in Cassandra

When modeling the database, some developers might be tempted to store tags associated with videos in a separate table. When the list of anticipated tags is small however, using a collection data type that stores tags inside the database record can be more efficient. This simplifies the database design and reduces the number of tables required.

The five collection data types in Cassandra are:

- **Set** – a group collection of unique values of the same data type.
- **List** – an ordered collection of non-unique values of the same data type.
- **Map** – a set of key-value pairs, where keys are unique, and both keys and values have associated data types.
- **Tuple** – a fixed length list of non-unique values of different data types.
- **Nested collection** – a collection (i.e., set, list, map, or tuple) that is nested inside of another collection.

When defining a collection, the user needs to provide a data type for its elements. A simplified version of our videos table is provided below for illustration.

```
CREATE TABLE videos (videoid uuid PRIMARY KEY, name text, tags set<text>);
```

A sample row in this table may look as shown:

```
videoid                              | name           | tags
-------------------------------------+----------------+-----------------------------------
5b6962dd-3f90-4c93-8f61-eabfa4a803e2 | My Funny Cat Video | {'buster', 'ball of wool', 'bath'}
```

CQL provides convenient syntax to insert, update, or delete items in collections. For example, a user can update the record for “My Funny Cat Video” and add a tag “wet cat” as shown:

```
UPDATE videos SET tags = tags + {'wet cat'}

WHERE videoid = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

User-Defined Data Types

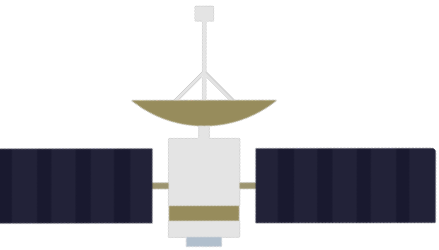
Another data type in Cassandra that provides flexibility is a user-defined type (UDT). UDTs can attach multiple data fields—each named and typed—to a single column.

Let’s assume that the designers of KillrVideo decide to store an optional mailing address for each user. Rather than add multiple address-related fields, an address type can be created and leveraged across multiple Cassandra tables.

```
CREATE TYPE address (
    unit text,
    street_number text,
    street_name text,
    city text,
    prov_state text,
    post_code text
);
```

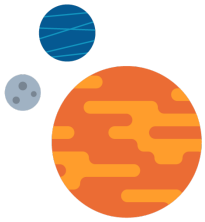
The user-defined address type can now be included in the users table as shown. The frozen keyword is required to use a UDT inside of a collection. It forces Cassandra to treat the address as a single value. Individual elements of a frozen address cannot be updated individually; rather, the entire address must be overwritten.

```
CREATE TABLE users (
    userid uuid PRIMARY KEY,
    firstname text,
    lastname text,
    address address,
    previous_addresses list<frozen<address>>s
);
```



03

Things to Keep in Mind



Thinking in relational terms can cause problems

On the surface, Cassandra's Query Language (CQL) looks much like SQL but failing to consider how Cassandra stores data can cause serious problems. Consider an example where we decide to partition our user community into groups. We want to be able to retrieve information about all users that belong in a group as shown.

```
SELECT * FROM groups WHERE groupname = ? ;
```

A good table structure to support such a query in Cassandra would use the groupname as a partition key so that members of the same group reside on the same partition for fast lookup.

```
CREATE TABLE groups (
    groupname text,
    group_description text static,
    userid uuid,
    firstname text,
    lastname text,
    PRIMARY KEY (groupname, userid)
);
```

Someone with a relational database background might be tempted to normalize the database schema by creating tables for users, groups, and user-group relationships as shown.

```
CREATE TABLE users (
    userid uuid,
    firstname text,
    lastname text,
    PRIMARY KEY (userid)
);
CREATE TABLE groups (
    groupname text,
    group_description text,
    PRIMARY KEY (groupname)
);
CREATE TABLE user_groups (
    groupname text,
    userid uuid,
    PRIMARY KEY (groupname, userid)
);
```

Now suppose there were 1,000 users in each group, and we wanted to perform the same query retrieving information for all users in a group. In this normalized case, the application would need to read 1,002 partitions—first querying the groups table to retrieve the group_description, then querying the user_groups table to retrieve all userids in the group. After this, the users table needs to be queried for each of the 1,000 userids, each on a separate partition.

Secondary indexes

Consider a table holding user accounts where a unique username identifies an account. The table structure below is ideal for looking up users by username (the partition key).

```
CREATE TABLE user_accounts (
    username text PRIMARY KEY,
    email text,
    password text,
    country text
);
```

Now imagine that we need to generate a list of user accounts in a single country. Normally, a query like this would require a full table scan. This is a horrible idea in Cassandra because depending on the size of the table, a query like this may be very expensive and may consume all the resources on the cluster.

One way to solve this problem is to create another table designed to support the new query, with country designated as the partition key and username as the clustering key. However, such a solution is still problematic because country is a low-cardinality column. As our database scales, we may end up with millions of users in the same country, causing the partition for a country to become very large. In general, we do not recommend creating partitions with more than a few hundred thousand rows. A better practice would be to use a composite partition key with a country, state, city, and so forth to reduce the number of rows per partition.

Another solution, which works better for low-cardinality columns, is to use a secondary index:

```
CREATE INDEX user_account_country ON user_accounts(country);
```

The secondary index allows users to run queries like the following:

```
SELECT * FROM user_accounts WHERE country = 'UK';
```

This is convenient and looks much like a relational database. So why not always use a secondary index as opposed to creating query-specific tables?

The challenge is how secondary indexes are stored. Replicating a global index for every record on every Cassandra host is not feasible, so instead, Cassandra maintains a secondary index on each host indexing only the data on that host. To use a secondary index to search across all user_accounts in the UK, the query needs to check with every node in the Cassandra cluster, and then use the local index to retrieve only the records where country = "UK". This is expensive since user accounts stored in the index reside in different partitions, but at least the query only reads records for users known to reside in the UK—far less expensive than a full table scan.

Creating a secondary index on a high-cardinality column like email would be disastrous. Whereas there are only 195 countries, in a table with 100,000,000 rows, every email address would be unique. The index on each host would be enormous, and the amount of work required to support the query on each host would scale accordingly.

Secondary indexes have their place, but they should only be used when queries are expected to return tens or perhaps hundreds of rows at most, and when the indexed column has medium cardinality.

Materialized views

Consider a table holding user accounts where a unique username identifies an account. The table structure below is ideal for looking up users by username (the partition key).

```
CREATE TABLE user_accounts (
    username text PRIMARY KEY,
    email text,
    password text,
    country text
);
```

Now imagine that we also need to retrieve user accounts by email. It would be straightforward to solve this problem by creating another table designed to support the new query, with email designated as the partition key and username as the clustering key. There is also another good option called materialized view.

```
CREATE MATERIALIZED VIEW users_by_email AS
SELECT email, password, country
FROM user_accounts
WHERE username IS NOT NULL AND email IS NOT NULL
PRIMARY KEY (email, username);
```

A materialized view is essentially a CQL table that is defined using a base table. It is automatically maintained by Cassandra and queried like regular tables.

```
SELECT * FROM users_by_email WHERE email = ?;
```

Materialized views are great for convenience and have comparable performance as base tables, but each materialized view is restricted to a single base table. They work the best for high-cardinality columns.

Using batches effectively

Batch operations in Cassandra provide a way to group multiple insert, update, or delete queries together. Cassandra supports two types of batches: unlogged and logged batches. Logged batches are atomic—in other words, either all operations in the batch complete or none complete. Batches are useful in updating denormalized tables—updating a value spread across multiple tables and treating the update as a single transaction.

Developers new to Cassandra often make the mistake of assuming that batches will improve performance. For batches that update data across multiple partitions, the opposite is usually the case. A single coordinator node is assigned the responsibility of handling all the queries between the BEGIN BATCH and END BATCH statements. The coordinator most likely won't have local replicas of required data so it will need to reach out to multiple nodes in the Cassandra cluster, thereby becoming a bottleneck as it manages traffic for multiple queries across many nodes. The batch may fail if there are not enough nodes up and responding.

The exception is a batch where all operations write to a single partition. In this single partition case, the batch coordination and database update activities can all be handled efficiently on a single Cassandra node and the unlogged batch can outperform discrete queries.

Logged batches are 30% slower than unlogged batches, but logged batches provide an excellent way to enforce consistency when updating denormalized data. Suppose a user wishes to change the name of a previously uploaded video. The video name appears in multiple tables. This operation is performed rarely, and in this case, the need for atomicity outweighs any performance concerns, so grouping queries in a logged batch is a good design decision.

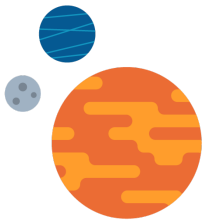
Learn More

For developers wanting to learn more about data modeling in Cassandra, DataStax offers a free course ***(DS220: Data Modeling)*** available at the DataStax Academy. This is one of the most popular courses.

The data modeling guidelines offered here apply to Apache Cassandra and DataStax Enterprise.

04

Summary



Getting the data model right is a critical first step in building a zero downtime, zero lock-in, global-scale application that is easy to manage and maintain.

A good way to arrive at a correct and robust data model is to work through a proven methodology for data modeling. Five key steps are to:

1. Map out the application workflow to identify all the functionality that the application needs to support.
2. Practice "query-first design" thinking about how to structure tables optimally to support the queries required by each workflow step.
3. Design the tables that will support the required queries using Chebotko diagrams.
4. Pay special attention to primary key, partition key, and clustering key design.
5. Use Cassandra data types effectively and think about where you can use unique Cassandra features such as collections and user-defined types to simplify the design.

Cassandra meets all customer requirements around scale, performance, and availability, making it a preferred choice among IT architects, developers, administrators, and decision-makers.

To learn more about data modeling with Cassandra and DataStax Enterprise, visit ***www.datastax.com*** or send an email to info@datastax.com.

© 2020 DataStax, All Rights Reserved. DataStax, Titan, and TitanDB are registered trademarks of DataStax, Inc. and its subsidiaries in the United States and/or other countries.

Apache, Apache Cassandra, and Cassandra are either registered trademarks or trademarks of the Apache Software Foundation or its subsidiaries in Canada, the United States, and/or other countries.