Apache Cassandra : Data Modeling

Data Modeling

Data Modeling

- **Introduction**
- **RDBMS**
- **Conceptual**
- **Logical**
- **Physical**
- **Schema**
- **Queries**
- **Refining**
- **Tools**

Introduction

Introduction

Apache Cassandra stores data in tables, with each table consisting of rows and columns. CQL (Cassandra Query Language) is used to query the data stored in tables. Apache Cassandra data model is based around and optimized for querying. Cassandra does not support relational data modeling intended for relational databases.

What is Data Modeling?

Data modeling is the process of identifying entities and their relationships. In relational databases, data is placed in normalized tables with foreign keys used to reference related data in other tables. Queries that the application will make are driven by the structure of the tables and related data are queried as table joins.

In Cassandra, data modeling is query-driven. The data access patterns and application queries determine the structure and organization of data which then used to design the database tables.

Data is modeled around specific queries. Queries are best designed to access a single table, which implies that all entities involved in a query must be in the same table to make data access (reads) very fast. Data is modeled to best suit a query or a set of queries. A table could have one or more entities as best suits a query. As entities do typically have relationships among them and queries could involve entities with relationships among them, a single entity may be included in multiple tables.

Query-driven modeling

Unlike a relational database model in which queries make use of table joins to get data from multiple tables, joins are not supported in Cassandra so all required fields (columns) must be grouped together in a single table. Since each query is backed by a table, data is duplicated across multiple tables in a process known as denormalization. Data duplication and a high write throughput are used to achieve a high read performance.

Goals

The choice of the primary key and partition key is important to distribute data evenly across the cluster. Keeping the number of partitions read for a query to a minimum is also important because different partitions could be located on different nodes and the coordinator would need to send a request to each node adding to the request overhead and latency. Even if the different partitions involved in a query are on the same node, fewer partitions make for a more efficient query.

Partitions

Apache Cassandra is a distributed database that stores data across a cluster of nodes. A partition key is used to partition data among the nodes. Cassandra partitions data over the storage nodes using a variant of consistent hashing for data distribution. Hashing is a technique used to map data with which given a key, a hash function generates a hash value (or simply a hash) that is stored in a hash table. A partition key is generated from the first field of a primary key. Data partitioned into hash tables using partition keys provides for rapid lookup. Fewer the partitions used for a query faster is the response time for the query.

As an example of partitioning, consider table `t` in which `id` is the only field in the primary key.

```
CREATE TABLE t (
   id int,
   k int,
   v text,
   PRIMARY KEY (id)
);
```

The partition key is generated from the primary key `id` for data distribution across the nodes in a cluster.

Consider a variation of table `t` that has two fields constituting the primary key to make a composite or compound primary key.

```
CREATE TABLE t (
   id int,
   c text,
   k int,
   v text,
   PRIMARY KEY (id,c)
);
```

For the table `t` with a composite primary key the first field `id` is used to generate the partition key and the second field `c` is the clustering key used for sorting within a partition. Using clustering keys to sort data makes retrieval of adjacent data more efficient.

In general, the first field or component of a primary key is hashed to generate the partition key and the remaining fields or components are the clustering keys that are used to sort data within a partition. Partitioning data improves the efficiency of reads and writes. The other fields that are not primary key fields may be indexed separately to further improve query performance.

The partition key could be generated from multiple fields if they are grouped as the first component of a primary key. As another variation of the table `t`, consider a table with the first component of the

primary key made of two fields grouped using parentheses.

```
CREATE TABLE t (
   id1 int,
   id2 int,
   c1 text,
   c2 text
   k int,
   v text,
   PRIMARY KEY ((id1,id2),c1,c2)
);
```

For the preceding table `t` the first component of the primary key constituting fields `id1` and `id2` is used to generate the partition key and the rest of the fields `c1` and `c2` are the clustering keys used for sorting within a partition.

Comparing with Relational Data Model

Relational databases store data in tables that have relations with other tables using foreign keys. A relational database's approach to data modeling is table-centric. Queries must use table joins to get data from multiple tables that have a relation between them. Apache Cassandra does not have the concept of foreign keys or relational integrity. Apache Cassandra's data model is based around designing efficient queries; queries that don't involve multiple tables. Relational databases normalize data to avoid duplication. Apache Cassandra in contrast de-normalizes data by duplicating data in multiple tables for a query-centric data model. If a Cassandra data model cannot fully integrate the complexity of relationships between the different entities for a particular query, client-side joins in application code may be used.

Examples of Data Modeling

As an example, a `magazine` data set consists of data for magazines with attributes such as magazine id, magazine name, publication frequency, publication date, and publisher. A basic query (Q1) for magazine data is to list all the magazine names including their publication frequency. As not all data attributes are needed for Q1 the data model would only consist of `id` (for partition key), magazine name and publication frequency as shown in Figure 1.

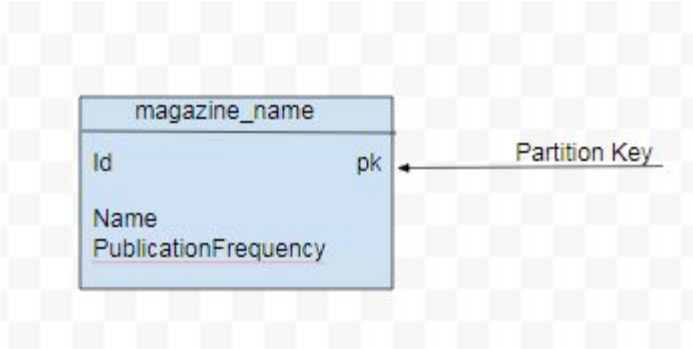

Figure 1. Data Model for Q1

Another query (Q2) is to list all the magazine names by publisher. For Q2 the data model would consist of an additional attribute `publisher` for the partition key. The `id` would become the clustering key for sorting within a partition. Data model for Q2 is illustrated in Figure 2.

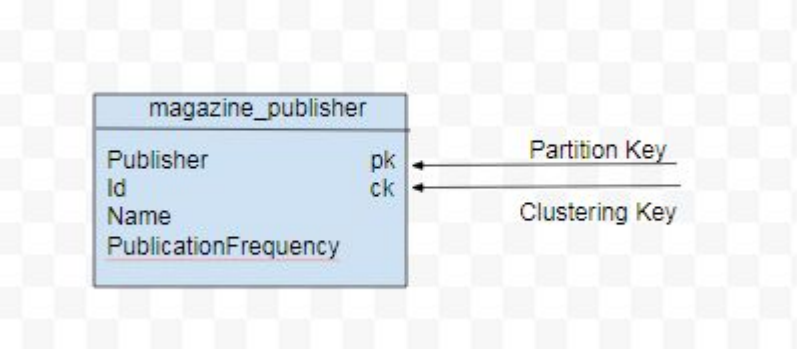

Figure 2. Data Model for Q2

Designing Schema

After the conceptual data model has been created a schema may be designed for a query. For Q1 the following schema may be used.

```
CREATE TABLE magazine_name (id int PRIMARY KEY, name text, publicationFrequency text)
```

For Q2 the schema definition would include a clustering key for sorting.

```
CREATE TABLE magazine_publisher (publisher text,id int,name text, publicationFrequency
text,
PRIMARY KEY (publisher, id)) WITH CLUSTERING ORDER BY (id DESC)
```

Data Model Analysis

The data model is a conceptual model that must be analyzed and optimized based on storage, capacity, redundancy and consistency. A data model may need to be modified as a result of the analysis.

Considerations or limitations that are used in data model analysis include:

- Partition Size
- Data Redundancy
- Disk space
- Lightweight Transactions (LWT)

The two measures of partition size are the number of values in a partition and partition size on disk. Though requirements for these measures may vary based on the application a general guideline is to keep number of values per partition to below 100,000 and disk space per partition to below 100MB.

Data redundancies as duplicate data in tables and multiple partition replicates are to be expected in the design of a data model , but nevertheless should be kept in consideration as a parameter to keep to the minimum. LWT transactions (compare-and-set, conditional update) could affect performance and queries using LWT should be kept to the minimum.

Using Materialized Views

WARNING	Materialized views (MVs) are experimental as of the 4.0 release.

Materialized views (MVs) could be used to implement multiple queries for a single table. A materialized view is a table built from data from another table, the base table, with new primary key and new properties. Changes to the base table data automatically add and update data in a MV. Different queries may be implemented using a materialized view as an MV's primary key differs from the base table. Queries are optimized by the primary key definition.

Conceptual data modeling

Conceptual data modeling

First, let's create a simple domain model that is easy to understand in the relational world, and then see how you might map it from a relational to a distributed hashtable model in Cassandra.

Let's use an example that is complex enough to show the various data structures and design patterns, but not something that will bog you down with details. Also, a domain that's familiar to everyone will allow you to concentrate on how to work with Cassandra, not on what the application domain is all about.

For example, let's use a domain that is easily understood and that everyone can relate to: making hotel reservations.

The conceptual domain includes hotels, guests that stay in the hotels, a collection of rooms for each hotel, the rates and availability of those rooms, and a record of reservations booked for guests. Hotels typically also maintain a collection of "points of interest," which are parks, museums, shopping galleries, monuments, or other places near the hotel that guests might want to visit during their stay. Both hotels and points of interest need to maintain geolocation data so that they can be found on maps for mashups, and to calculate distances.

The conceptual domain is depicted below using the entity–relationship model popularized by Peter Chen. This simple diagram represents the entities in the domain with rectangles, and attributes of those entities with ovals. Attributes that represent unique identifiers for items are underlined. Relationships between entities are represented as diamonds, and the connectors between the relationship and each entity show the multiplicity of the connection.

[image] | *data-modeling_hotel_erd.png*

Obviously, in the real world, there would be many more considerations and much more complexity. For example, hotel rates are notoriously dynamic, and calculating them involves a wide array of factors. Here you're defining something complex enough to be interesting and touch on the important points, but simple enough to maintain the focus on learning Cassandra.

RDBMS design

RDBMS design

When you set out to build a new data-driven application that will use a relational database, you might start by modeling the domain as a set of properly normalized tables and use foreign keys to reference related data in other tables.

The figure below shows how you might represent the data storage for your application using a relational database model. The relational model includes a couple of “join” tables in order to realize the many-to-many relationships from the conceptual model of hotels-to-points of interest, rooms-to-amenities, rooms-to-availability, and guests-to-rooms (via a reservation).

[image] | *data-modeling_hotel_relational.png*

Design Differences Between RDBMS and Cassandra

Let’s take a minute to highlight some of the key differences in doing data modeling for Cassandra versus a relational database.

No joins

You cannot perform joins in Cassandra. If you have designed a data model and find that you need something like a join, you’ll have to either do the work on the client side, or create a denormalized second table that represents the join results for you. This latter option is preferred in Cassandra data modeling. Performing joins on the client should be a very rare case; you really want to duplicate (denormalize) the data instead.

No referential integrity

Although Cassandra supports features such as lightweight transactions and batches, Cassandra itself has no concept of referential integrity across tables. In a relational database, you could specify foreign keys in a table to reference the primary key of a record in another table. But Cassandra does not enforce this. It is still a common design requirement to store IDs related to other entities in your tables, but operations such as cascading deletes are not available.

Denormalization

In relational database design, you are often taught the importance of normalization. This is not an advantage when working with Cassandra because it performs best when the data model is denormalized. It is often the case that companies end up denormalizing data in relational databases as well. There are two common reasons for this. One is performance. Companies simply can’t get the

performance they need when they have to do so many joins on years' worth of data, so they denormalize along the lines of known queries. This ends up working, but goes against the grain of how relational databases are intended to be designed, and ultimately makes one question whether using a relational database is the best approach in these circumstances.

A second reason that relational databases get denormalized on purpose is a business document structure that requires retention. That is, you have an enclosing table that refers to a lot of external tables whose data could change over time, but you need to preserve the enclosing document as a snapshot in history. The common example here is with invoices. You already have customer and product tables, and you'd think that you could just make an invoice that refers to those tables. But this should never be done in practice. Customer or price information could change, and then you would lose the integrity of the invoice document as it was on the invoice date, which could violate audits, reports, or laws, and cause other problems.

In the relational world, denormalization violates Codd's normal forms, and you try to avoid it. But in Cassandra, denormalization is, well, perfectly normal. It's not required if your data model is simple. But don't be afraid of it.

Historically, denormalization in Cassandra has required designing and managing multiple tables using techniques described in this documentation. Beginning with the 3.0 release, Cassandra provides a feature known as `materialized views <materialized-views>` which allows you to create multiple denormalized views of data based on a base table design. Cassandra manages materialized views on the server, including the work of keeping the views in sync with the table.

Query-first design

Relational modeling, in simple terms, means that you start from the conceptual domain and then represent the nouns in the domain in tables. You then assign primary keys and foreign keys to model relationships. When you have a many-to-many relationship, you create the join tables that represent just those keys. The join tables don't exist in the real world, and are a necessary side effect of the way relational models work. After you have all your tables laid out, you can start writing queries that pull together disparate data using the relationships defined by the keys. The queries in the relational world are very much secondary. It is assumed that you can always get the data you want as long as you have your tables modeled properly. Even if you have to use several complex subqueries or join statements, this is usually true.

By contrast, in Cassandra you don't start with the data model; you start with the query model. Instead of modeling the data first and then writing queries, with Cassandra you model the queries and let the data be organized around them. Think of the most common query paths your application will use, and then create the tables that you need to support them.

Detractors have suggested that designing the queries first is overly constraining on application design, not to mention database modeling. But it is perfectly reasonable to expect that you should think hard about the queries in your application, just as you would, presumably, think hard about your relational domain. You may get it wrong, and then you'll have problems in either world. Or your query needs

might change over time, and then you'll have to work to update your data set. But this is no different from defining the wrong tables, or needing additional tables, in an RDBMS.

Designing for optimal storage

In a relational database, it is frequently transparent to the user how tables are stored on disk, and it is rare to hear of recommendations about data modeling based on how the RDBMS might store tables on disk. However, that is an important consideration in Cassandra. Because Cassandra tables are each stored in separate files on disk, it's important to keep related columns defined together in the same table.

A key goal that you will see as you begin creating data models in Cassandra is to minimize the number of partitions that must be searched in order to satisfy a given query. Because the partition is a unit of storage that does not get divided across nodes, a query that searches a single partition will typically yield the best performance.

Sorting is a design decision

In an RDBMS, you can easily change the order in which records are returned to you by using `ORDER BY` in your query. The default sort order is not configurable; by default, records are returned in the order in which they are written. If you want to change the order, you just modify your query, and you can sort by any list of columns.

In Cassandra, however, sorting is treated differently; it is a design decision. The sort order available on queries is fixed, and is determined entirely by the selection of clustering columns you supply in the `CREATE TABLE` command. The CQL `SELECT` statement does support `ORDER BY` semantics, but only in the order specified by the clustering columns.

Defining application queries

Defining application queries

Let's try the query-first approach to start designing the data model for a hotel application. The user interface design for the application is often a great artifact to use to begin identifying queries. Let's assume that you've talked with the project stakeholders and your UX designers have produced user interface designs or wireframes for the key use cases. You'll likely have a list of shopping queries like the following:

- Q1. Find hotels near a given point of interest.
- Q2. Find information about a given hotel, such as its name and location.
- Q3. Find points of interest near a given hotel.
- Q4. Find an available room in a given date range.
- Q5. Find the rate and amenities for a room.

It is often helpful to be able to refer to queries by a shorthand number rather that explaining them in full. The queries listed here are numbered Q1, Q2, and so on, which is how they are referenced in diagrams throughout the example.

Now if the application is to be a success, you'll certainly want customers to be able to book reservations at hotels. This includes steps such as selecting an available room and entering their guest information. So clearly you will also need some queries that address the reservation and guest entities from the conceptual data model. Even here, however, you'll want to think not only from the customer perspective in terms of how the data is written, but also in terms of how the data will be queried by downstream use cases.

You natural tendency as might be to focus first on designing the tables to store reservation and guest records, and only then start thinking about the queries that would access them. You may have felt a similar tension already when discussing the shopping queries before, thinking "but where did the hotel and point of interest data come from?" Don't worry, you will see soon enough. Here are some queries that describe how users will access reservations:

- Q6. Lookup a reservation by confirmation number.
- Q7. Lookup a reservation by hotel, date, and guest name.
- Q8. Lookup all reservations by guest name.
- Q9. View guest details.

All of the queries are shown in the context of the workflow of the application in the figure below. Each box on the diagram represents a step in the application workflow, with arrows indicating the flows between steps and the associated query. If you've modeled the application well, each step of the workflow accomplishes a task that "unlocks" subsequent steps. For example, the "View hotels near POI" task helps the application learn about several hotels, including their unique keys. The key for a selected hotel may be used as part of Q2, in order to obtain detailed description of the hotel. The act of

booking a room creates a reservation record that may be accessed by the guest and hotel staff at a later time through various additional queries.

[image] | *cassandra:developing/data-modeling/data-modeling_hotel_queries.png*

Logical data modeling

Logical data modeling

Now that you have defined your queries, you're ready to begin designing Cassandra tables. First, create a logical model containing a table for each query, capturing entities and relationships from the conceptual model.

To name each table, you'll identify the primary entity type for which you are querying and use that to start the entity name. If you are querying by attributes of other related entities, append those to the table name, separated with `by`. For example, `hotels_by_poi`.

Next, you identify the primary key for the table, adding partition key columns based on the required query attributes, and clustering columns in order to guarantee uniqueness and support desired sort ordering.

The design of the primary key is extremely important, as it will determine how much data will be stored in each partition and how that data is organized on disk, which in turn will affect how quickly Cassandra processes reads.

Complete each table by adding any additional attributes identified by the query. If any of these additional attributes are the same for every instance of the partition key, mark the column as static.

Now that was a pretty quick description of a fairly involved process, so it will be worthwhile to work through a detailed example. First, let's introduce a notation that you can use to represent logical models.

Several individuals within the Cassandra community have proposed notations for capturing data models in diagrammatic form. This document uses a notation popularized by Artem Chebotko which provides a simple, informative way to visualize the relationships between queries and tables in your designs. This figure shows the Chebotko notation for a logical data model.

[image] | *cassandra:developing/data-modeling/data-modeling_chebotko_logical.png*

Each table is shown with its title and a list of columns. Primary key columns are identified via symbols such as **K** for partition key columns and **C**↑ or **C**↓ to represent clustering columns. Lines are shown entering tables or between tables to indicate the queries that each table is designed to support.

Hotel Logical Data Model

The figure below shows a Chebotko logical data model for the queries involving hotels, points of interest, rooms, and amenities. One thing you'll notice immediately is that the Cassandra design doesn't include dedicated tables for rooms or amenities, as you had in the relational design. This is because the workflow didn't identify any queries requiring this direct access.

[image] | *cassandra:developing/data-modeling/data-modeling_hotel_logical.png*

Let's explore the details of each of these tables.

The first query Q1 is to find hotels near a point of interest, so you'll call this table `hotels_by_poi`. Searching by a named point of interest is a clue that the point of interest should be a part of the primary key. Let's reference the point of interest by name, because according to the workflow that is how users will start their search.

You'll note that you certainly could have more than one hotel near a given point of interest, so you'll need another component in the primary key in order to make sure you have a unique partition for each hotel. So you add the hotel key as a clustering column.

An important consideration in designing your table's primary key is making sure that it defines a unique data element. Otherwise you run the risk of accidentally overwriting data.

Now for the second query (Q2), you'll need a table to get information about a specific hotel. One approach would have been to put all of the attributes of a hotel in the `hotels_by_poi` table, but you added only those attributes that were required by the application workflow.

From the workflow diagram, you know that the `hotels_by_poi` table is used to display a list of hotels with basic information on each hotel, and the application knows the unique identifiers of the hotels returned. When the user selects a hotel to view details, you can then use Q2, which is used to obtain details about the hotel. Because you already have the `hotel_id` from Q1, you use that as a reference to the hotel you're looking for. Therefore the second table is just called `hotels`.

Another option would have been to store a set of `poi_names` in the hotels table. This is an equally valid approach. You'll learn through experience which approach is best for your application.

Q3 is just a reverse of Q1—looking for points of interest near a hotel, rather than hotels near a point of interest. This time, however, you need to access the details of each point of interest, as represented by the `pois_by_hotel` table. As previously, you add the point of interest name as a clustering key to guarantee uniqueness.

At this point, let's now consider how to support query Q4 to help the user find available rooms at a selected hotel for the nights they are interested in staying. Note that this query involves both a start date and an end date. Because you're querying over a range instead of a single date, you know that you'll need to use the date as a clustering key. Use the `hotel_id` as a primary key to group room data for each hotel on a single partition, which should help searches be super fast. Let's call this the `available_rooms_by_hotel_date` table.

To support searching over a range, use `clustering columns <clustering-columns>` to store attributes that you need to access in a range query. Remember that the order of the clustering columns is important.

The design of the `available_rooms_by_hotel_date` table is an instance of the **wide partition** pattern. This pattern is sometimes called the **wide row** pattern when discussing databases that support similar models, but wide partition is a more accurate description from a Cassandra perspective. The essence of the pattern is to group multiple related rows in a partition in order to support fast access to multiple rows within the partition in a single query.

In order to round out the shopping portion of the data model, add the `amenities_by_room` table to support Q5. This will allow users to view the amenities of one of the rooms that is available for the desired stay dates.

Reservation Logical Data Model

Now let's switch gears to look at the reservation queries. The figure shows a logical data model for reservations. You'll notice that these tables represent a denormalized design; the same data appears in multiple tables, with differing keys.

[image] | *cassandra:developing/data-modeling/data-modeling_reservation_logical.png*

In order to satisfy Q6, the `reservations_by_guest` table can be used to look up the reservation by guest name. You could envision query Q7 being used on behalf of a guest on a self-serve website or a call center agent trying to assist the guest. Because the guest name might not be unique, you include the guest ID here as a clustering column as well.

Q8 and Q9 in particular help to remind you to create queries that support various stakeholders of the application, not just customers but staff as well, and perhaps even the analytics team, suppliers, and so on.

The hotel staff might wish to see a record of upcoming reservations by date in order to get insight into how the hotel is performing, such as what dates the hotel is sold out or undersold. Q8 supports the retrieval of reservations for a given hotel by date.

Finally, you create a `guests` table. This provides a single location that used to store guest information. In this case, you specify a separate unique identifier for guest records, as it is not uncommon for guests to have the same name. In many organizations, a customer database such as the `guests` table would be part of a separate customer management application, which is why other guest access patterns were omitted from the example.

Patterns and Anti-Patterns

As with other types of software design, there are some well-known patterns and anti-patterns for data modeling in Cassandra. You've already used one of the most common patterns in this hotel model—the wide partition pattern.

The **time series** pattern is an extension of the wide partition pattern. In this pattern, a series of measurements at specific time intervals are stored in a wide partition, where the measurement time is used as part of the partition key. This pattern is frequently used in domains including business analysis, sensor data management, and scientific experiments.

The time series pattern is also useful for data other than measurements. Consider the example of a banking application. You could store each customer's balance in a row, but that might lead to a lot of read and write contention as various customers check their balance or make transactions. You'd

probably be tempted to wrap a transaction around writes just to protect the balance from being updated in error. In contrast, a time series–style design would store each transaction as a timestamped row and leave the work of calculating the current balance to the application.

One design trap that many new users fall into is attempting to use Cassandra as a queue. Each item in the queue is stored with a timestamp in a wide partition. Items are appended to the end of the queue and read from the front, being deleted after they are read. This is a design that seems attractive, especially given its apparent similarity to the time series pattern. The problem with this approach is that the deleted items are now `tombstones <asynch-deletes>` that Cassandra must scan past in order to read from the front of the queue. Over time, a growing number of tombstones begins to degrade read performance.

The queue anti-pattern serves as a reminder that any design that relies on the deletion of data is potentially a poorly performing design.

Physical data modeling

Physical data modeling

Once you have a logical data model defined, creating the physical model is a relatively simple process.

You walk through each of the logical model tables, assigning types to each item. You can use any valid `CQL data type <data-types>`, including the basic types, collections, and user-defined types. You may identify additional user-defined types that can be created to simplify your design.

After you've assigned data types, you analyze the model by performing size calculations and testing out how the model works. You may make some adjustments based on your findings. Once again let's cover the data modeling process in more detail by working through an example.

Before getting started, let's look at a few additions to the Chebotko notation for physical data models. To draw physical models, you need to be able to add the typing information for each column. This figure shows the addition of a type for each column in a sample table.

[image] | *cassandra:developing/data-modeling/data-modeling_chebotko_physical.png*

The figure includes a designation of the keyspace containing each table and visual cues for columns represented using collections and user-defined types. Note the designation of static columns and secondary index columns. There is no restriction on assigning these as part of a logical model, but they are typically more of a physical data modeling concern.

Hotel Physical Data Model

Now let's get to work on the physical model. First, you need keyspaces to contain the tables. To keep the design relatively simple, create a `hotel` keyspace to contain tables for hotel and availability data, and a `reservation` keyspace to contain tables for reservation and guest data. In a real system, you might divide the tables across even more keyspaces in order to separate concerns.

For the `hotels` table, use Cassandra's `text` type to represent the hotel's `id`. For the address, create an `address` user defined type. Use the `text` type to represent the phone number, as there is considerable variance in the formatting of numbers between countries.

While it would make sense to use the `uuid` type for attributes such as the `hotel_id`, this document uses mostly `text` attributes as identifiers, to keep the samples simple and readable. For example, a common convention in the hospitality industry is to reference properties by short codes like "AZ123" or "NY229". This example uses these values for `hotel_ids`, while acknowledging they are not necessarily globally unique.

You'll find that it's often helpful to use unique IDs to uniquely reference elements, and to use these `uuids` as references in tables representing other entities. This helps to minimize coupling between different entity types. This may prove especially effective if you are using a microservice architectural

style for your application, in which there are separate services responsible for each entity type.

As you work to create physical representations of various tables in the logical hotel data model, you use the same approach. The resulting design is shown in this figure:

[image] | *cassandra:developing/data-modeling/data-modeling_hotel_physical.png*

Note that the `address` type is also included in the design. It is designated with an asterisk to denote that it is a user-defined type, and has no primary key columns identified. This type is used in the `hotels` and `hotels_by_poi` tables.

User-defined types are frequently used to help reduce duplication of non-primary key columns, as was done with the `address` user-defined type. This can reduce complexity in the design.

Remember that the scope of a UDT is the keyspace in which it is defined. To use `address` in the `reservation` keyspace defined below design, you'll have to declare it again. This is just one of the many trade-offs you have to make in data model design.

Reservation Physical Data Model

Now, let's examine reservation tables in the design. Remember that the logical model contained three denormalized tables to support queries for reservations by confirmation number, guest, and hotel and date. For the first iteration of your physical data model design, assume you're going to manage this denormalization manually. Note that this design could be revised to use Cassandra's (experimental) materialized view feature.

[image] | *cassandra:developing/data-modeling/data-modeling_reservation_physical.png*

Note that the `address` type is reproduced in this keyspace and `guest_id` is modeled as a `uuid` type in all of the tables.

Evaluating and refining data models

Evaluating and refining data models

Once you've created a physical model, there are some steps you'll want to take to evaluate and refine table designs to help ensure optimal performance.

Calculating Partition Size

The first thing that you want to look for is whether your tables will have partitions that will be overly large, or to put it another way, too wide. Partition size is measured by the number of cells (values) that are stored in the partition. Cassandra's hard limit is 2 billion cells per partition, but you'll likely run into performance issues before reaching that limit.

In order to calculate the size of partitions, use the following formula:

$$N_v = N_r (N_c - N_{pk} - N_s) + N_s$$

The number of values (or cells) in the partition (N_v) is equal to the number of static columns (N_s) plus the product of the number of rows (N_r) and the number of of values per row. The number of values per row is defined as the number of columns (N_c) minus the number of primary key columns (N_{pk}) and static columns (N_s).

The number of columns tends to be relatively static, although it is possible to alter tables at runtime. For this reason, a primary driver of partition size is the number of rows in the partition. This is a key factor that you must consider in determining whether a partition has the potential to get too large. Two billion values sounds like a lot, but in a sensor system where tens or hundreds of values are measured every millisecond, the number of values starts to add up pretty fast.

Let's take a look at one of the tables to analyze the partition size. Because it has a wide partition design with one partition per hotel, look at the `available_rooms_by_hotel_date` table. The table has four columns total ($N_c = 4$), including three primary key columns ($N_{pk} = 3$) and no static columns ($N_s = 0$). Plugging these values into the formula, the result is:

$$N_v = N_r (4 - 3 - 0) + 0 = 1N_r$$

Therefore the number of values for this table is equal to the number of rows. You still need to determine a number of rows. To do this, make estimates based on the application design. The table is storing a record for each room, in each of hotel, for every night. Let's assume the system will be used to store two years of inventory at a time, and there are 5,000 hotels in the system, with an average of 100 rooms in each hotel.

Since there is a partition for each hotel, the estimated number of rows per partition is as follows:

```
\[N_r = 100 rooms/hotel \times 730 days = 73,000 rows\]
```

This relatively small number of rows per partition is not going to get you in too much trouble, but if you start storing more dates of inventory, or don't manage the size of the inventory well using TTL, you could start having issues. You still might want to look at breaking up this large partition, which you'll see how to do shortly.

When performing sizing calculations, it is tempting to assume the nominal or average case for variables such as the number of rows. Consider calculating the worst case as well, as these sorts of predictions have a way of coming true in successful systems.

Calculating Size on Disk

In addition to calculating the size of a partition, it is also an excellent idea to estimate the amount of disk space that will be required for each table you plan to store in the cluster. In order to determine the size, use the following formula to determine the size S_t of a partition:

```
\[S_t = \displaystyle\sum_i sizeOf\big (c_{k_i}\big) + \displaystyle\sum_j
sizeOf\big(c_{s_j}\big) + N_r\times \bigg(\displaystyle\sum_k sizeOf\big(c_{r_k}\big) +
\displaystyle\sum_l sizeOf\big(c_{c_l}\big)\bigg) +\]
```

```
\[N_v\times sizeOf\big(t_{avg}\big)\]
```

This is a bit more complex than the previous formula, but let's break it down a bit at a time. Let's take a look at the notation first:

- In this formula, c_k refers to partition key columns, c_s to static columns, c_r to regular columns, and c_c to clustering columns.
- The term t_{avg} refers to the average number of bytes of metadata stored per cell, such as timestamps. It is typical to use an estimate of 8 bytes for this value.
- You'll recognize the number of rows N_r and number of values N_v from previous calculations.
- The **sizeOf()** function refers to the size in bytes of the CQL data type of each referenced column.

The first term asks you to sum the size of the partition key columns. For this example, the `available_rooms_by_hotel_date` table has a single partition key column, the `hotel_id`, which is of type `text`. Assuming that hotel identifiers are simple 5-character codes, you have a 5-byte value, so the sum of the partition key column sizes is 5 bytes.

The second term asks you to sum the size of the static columns. This table has no static columns, so the size is 0 bytes.

The third term is the most involved, and for good reason—it is calculating the size of the cells in the partition. Sum the size of the clustering columns and regular columns. The two clustering columns are the `date`, which is 4 bytes, and the `room_number`, which is a 2-byte short integer, giving a sum of 6 bytes. There is only a single regular column, the boolean `is_available`, which is 1 byte in size. Summing the regular column size (1 byte) plus the clustering column size (6 bytes) gives a total of 7 bytes. To finish up the term, multiply this value by the number of rows (73,000), giving a result of 511,000 bytes (0.51 MB).

The fourth term is simply counting the metadata that that Cassandra stores for each cell. In the storage format used by Cassandra 3.0 and later, the amount of metadata for a given cell varies based on the type of data being stored, and whether or not custom timestamp or TTL values are specified for individual cells. For this table, reuse the number of values from the previous calculation (73,000) and multiply by 8, which gives 0.58 MB.

Adding these terms together, you get a final estimate:

```
\[Partition size = 16 bytes + 0 bytes + 0.51 MB + 0.58 MB = 1.1 MB\]
```

This formula is an approximation of the actual size of a partition on disk, but is accurate enough to be quite useful. Remembering that the partition must be able to fit on a single node, it looks like the table design will not put a lot of strain on disk storage.

Cassandra's storage engine was re-implemented for the 3.0 release, including a new format for SSTable files. The previous format stored a separate copy of the clustering columns as part of the record for each cell. The newer format eliminates this duplication, which reduces the size of stored data and simplifies the formula for computing that size.

Keep in mind also that this estimate only counts a single replica of data. You will need to multiply the value obtained here by the number of partitions and the number of replicas specified by the keyspace's replication strategy in order to determine the total required total capacity for each table. This will come in handy when you plan your cluster.

Breaking Up Large Partitions

As discussed previously, the goal is to design tables that can provide the data you need with queries that touch a single partition, or failing that, the minimum possible number of partitions. However, as shown in the examples, it is quite possible to design wide partition-style tables that approach Cassandra's built-in limits. Performing sizing analysis on tables may reveal partitions that are potentially too large, either in number of values, size on disk, or both.

The technique for splitting a large partition is straightforward: add an additional column to the partition key. In most cases, moving one of the existing columns into the partition key will be sufficient. Another option is to introduce an additional column to the table to act as a sharding key, but this requires additional application logic.

Continuing to examine the available rooms example, if you add the `date` column to the partition key for the `available_rooms_by_hotel_date` table, each partition would then represent the availability of rooms at a specific hotel on a specific date. This will certainly yield partitions that are significantly smaller, perhaps too small, as the data for consecutive days will likely be on separate nodes.

Another technique known as **bucketing** is often used to break the data into moderate-size partitions. For example, you could bucketize the `available_rooms_by_hotel_date` table by adding a `month` column to the partition key, perhaps represented as an integer. The comparison with the original design is shown in the figure below. While the `month` column is partially duplicative of the `date`, it provides a nice way of grouping related data in a partition that will not get too large.

[image] | *data-modeling_hotel_bucketing.png*

If you really felt strongly about preserving a wide partition design, you could instead add the `room_id` to the partition key, so that each partition would represent the availability of the room across all dates. Because there was no query identified that involves searching availability of a specific room, the first or second design approach is most suitable to the application needs.

Defining database schema

Defining database schema

Once you have finished evaluating and refining the physical model, you're ready to implement the schema in CQL. Here is the schema for the `hotel` keyspace, using CQL's comment feature to document the query pattern supported by each table:

```
CREATE KEYSPACE hotel WITH replication =
  {'class': 'SimpleStrategy', 'replication_factor' : 3};

CREATE TYPE hotel.address (
  street text,
  city text,
  state_or_province text,
  postal_code text,
  country text );

CREATE TABLE hotel.hotels_by_poi (
  poi_name text,
  hotel_id text,
  name text,
  phone text,
  address frozen<address>,
  PRIMARY KEY ((poi_name), hotel_id) )
  WITH comment = 'Q1. Find hotels near given poi'
  AND CLUSTERING ORDER BY (hotel_id ASC) ;

CREATE TABLE hotel.hotels (
  id text PRIMARY KEY,
  name text,
  phone text,
  address frozen<address>,
  pois set<text> )
  WITH comment = 'Q2. Find information about a hotel';

CREATE TABLE hotel.pois_by_hotel (
  poi_name text,
  hotel_id text,
  description text,
  PRIMARY KEY ((hotel_id), poi_name) )
  WITH comment = Q3. Find pois near a hotel';

CREATE TABLE hotel.available_rooms_by_hotel_date (
  hotel_id text,
  date date,
  room_number smallint,
  is_available boolean,
```

```
  PRIMARY KEY ((hotel_id), date, room_number) )
  WITH comment = 'Q4. Find available rooms by hotel date';

CREATE TABLE hotel.amenities_by_room (
  hotel_id text,
  room_number smallint,
  amenity_name text,
  description text,
  PRIMARY KEY ((hotel_id, room_number), amenity_name) )
  WITH comment = 'Q5. Find amenities for a room';
```

Notice that the elements of the partition key are surrounded with parentheses, even though the partition key consists of the single column `poi_name`. This is a best practice that makes the selection of partition key more explicit to others reading your CQL.

Similarly, here is the schema for the `reservation` keyspace:

```
CREATE KEYSPACE reservation WITH replication = {'class':
  'SimpleStrategy', 'replication_factor' : 3};

CREATE TYPE reservation.address (
  street text,
  city text,
  state_or_province text,
  postal_code text,
  country text );

CREATE TABLE reservation.reservations_by_confirmation (
  confirm_number text,
  hotel_id text,
  start_date date,
  end_date date,
  room_number smallint,
  guest_id uuid,
  PRIMARY KEY (confirm_number) )
  WITH comment = 'Q6. Find reservations by confirmation number';

CREATE TABLE reservation.reservations_by_hotel_date (
  hotel_id text,
  start_date date,
  end_date date,
  room_number smallint,
  confirm_number text,
  guest_id uuid,
  PRIMARY KEY ((hotel_id, start_date), room_number) )
  WITH comment = 'Q7. Find reservations by hotel and date';
```

```
CREATE TABLE reservation.reservations_by_guest (
  guest_last_name text,
  hotel_id text,
  start_date date,
  end_date date,
  room_number smallint,
  confirm_number text,
  guest_id uuid,
  PRIMARY KEY ((guest_last_name), hotel_id) )
  WITH comment = 'Q8. Find reservations by guest name';

CREATE TABLE reservation.guests (
  guest_id uuid PRIMARY KEY,
  first_name text,
  last_name text,
  title text,
  emails set,
  phone_numbers list,
  addresses map<text,
  frozen<address>,
  confirm_number text )
  WITH comment = 'Q9. Find guest by ID';
```

You now have a complete Cassandra schema for storing data for a hotel application.

Cassandra data modeling tools

Cassandra data modeling tools

There are several tools available to help you design and manage your Cassandra schema and build queries.

- Hackolade is a data modeling tool that supports schema design for Cassandra and many other NoSQL databases. Hackolade supports the unique concepts of CQL such as partition keys and clustering columns, as well as data types including collections and UDTs. It also provides the ability to create Chebotko diagrams.
- Kashlev Data Modeler is a Cassandra data modeling tool that automates the data modeling methodology described in this documentation, including identifying access patterns, conceptual, logical, and physical data modeling, and schema generation. It also includes model patterns that you can optionally leverage as a starting point for your designs.
- DataStax DevCenter is a tool for managing schema, executing queries and viewing results. While the tool is no longer actively supported, it is still popular with many developers and is available as a free download. DevCenter features syntax highlighting for CQL commands, types, and name literals. DevCenter provides command completion as you type out CQL commands and interprets the commands you type, highlighting any errors you make. The tool provides panes for managing multiple CQL scripts and connections to multiple clusters. The connections are used to run CQL commands against live clusters and view the results. The tool also has a query trace feature that is useful for gaining insight into the performance of your queries.
- IDE Plugins - There are CQL plugins available for several Integrated Development Environments (IDEs), such as IntelliJ IDEA and Apache NetBeans. These plugins typically provide features such as schema management and query execution.

Some IDEs and tools that claim to support Cassandra do not actually support CQL natively, but instead access Cassandra using a JDBC/ODBC driver and interact with Cassandra as if it were a relational database with SQL support. When selecting tools for working with Cassandra you'll want to make sure they support CQL and reinforce Cassandra best practices for data modeling as presented in this documentation.