# Unit 4 Syllabus

- **Data Modeling:** Conceptual Data modeling, RDBMS design, Logical Data Modeling, physical data Modeling, Evaluating and Refining, Defining database Schema.
- **Cassandra Architecture**: Da Data Centres and Racks, Gossip and Failure Detection, Snitches, Rings and Tokens, Virtual Nodes , Partitioners, Replication Strategies, Consistency Levels, Anti-Entropy, Repair, and Merkle Trees, Lightweight Transactions and Paxos, Memtables, SSTables, and Commit Logs, Caching

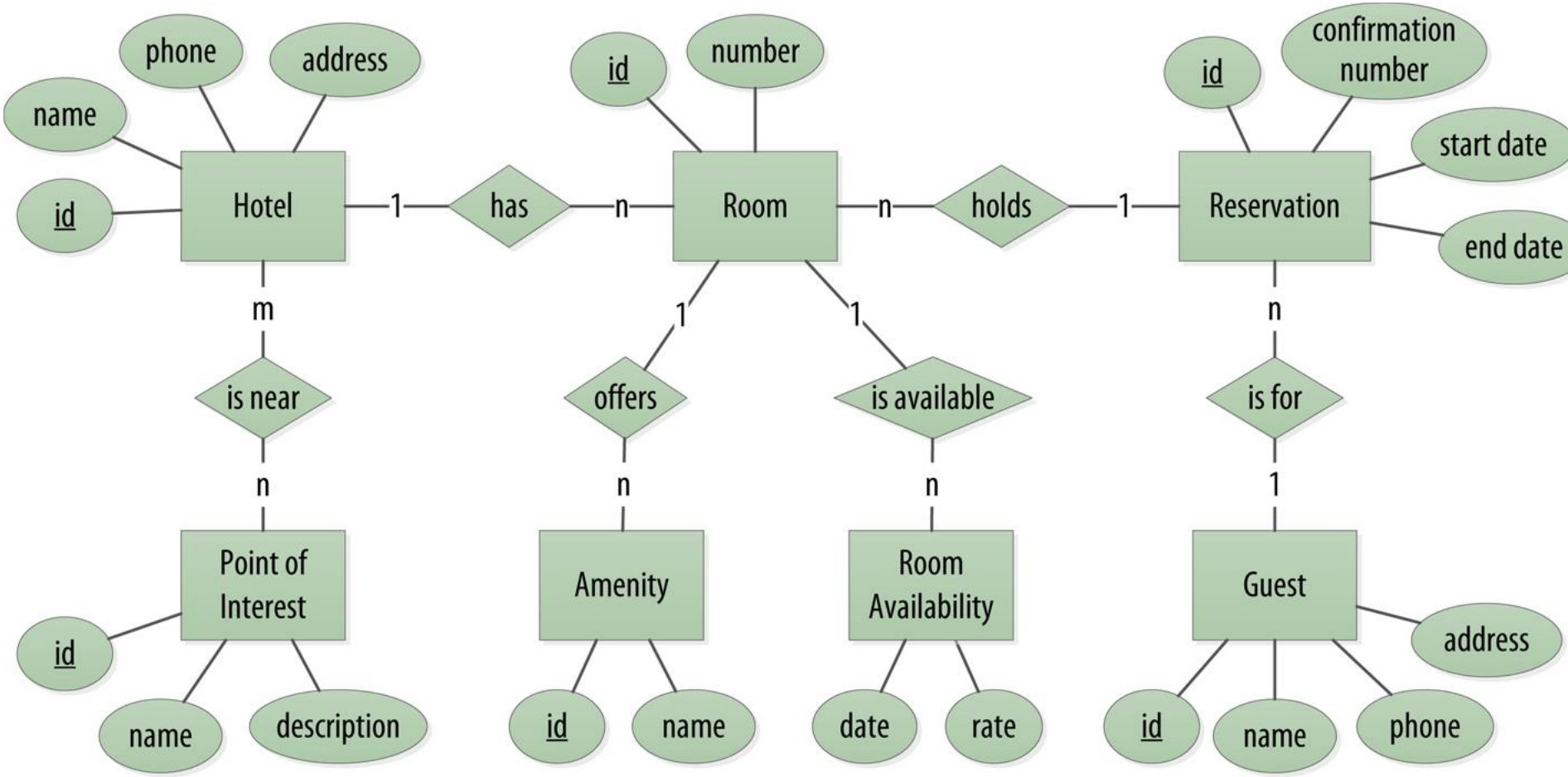
# Data Modeling

- Our 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

# Data Modeling

- The conceptual domain is shown in below figure 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

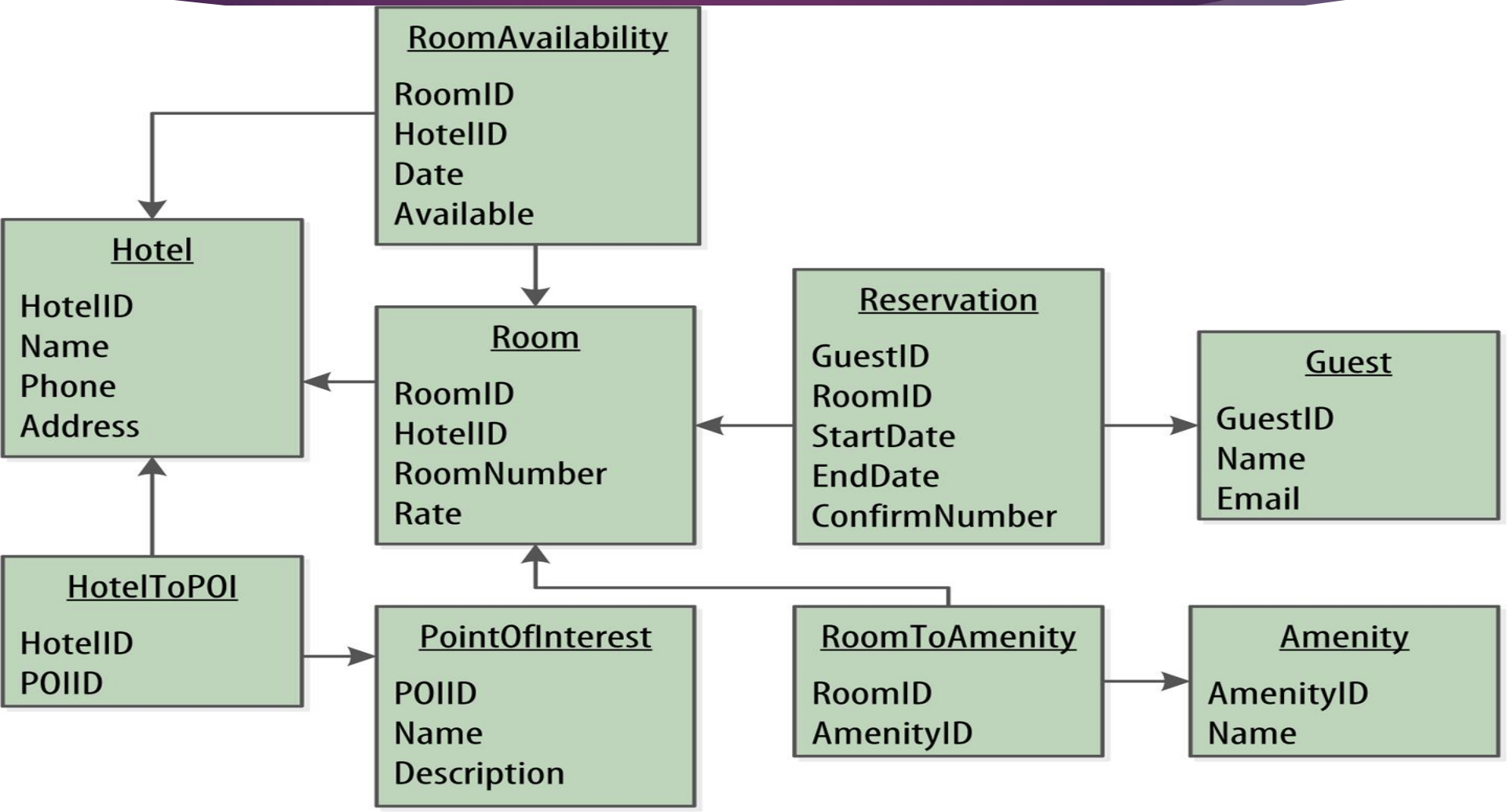
# Diagram


phone
address
name
id
Hotel
1
has
n
Room
id
number
n
holds
1
Reservation
id
confirmation number
start date
end date
m
is near
n
Point of Interest
id
name
description
1
offers
n
Amenity
id
name
1
is available
n
Room Availability
date
rate
n
is for
1
Guest
address
id
name
phone

# 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

# RDBMS Design



# Design Differences Between RDBMS and Cassandra

- **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.

# Design Differences Between RDBMS and Cassandra

- **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

# Design Differences Between RDBMS and Cassandra

- **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.

# Design Differences Between RDBMS and Cassandra

- **Denormalization**
- 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

# Design Differences Between RDBMS and Cassandra

- **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

# Design Differences Between RDBMS and Cassandra

- **Query-first design**
- 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

# Design Differences Between RDBMS and Cassandra

- **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 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

# Design Differences Between RDBMS and Cassandra

- **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 (ascending or descending).

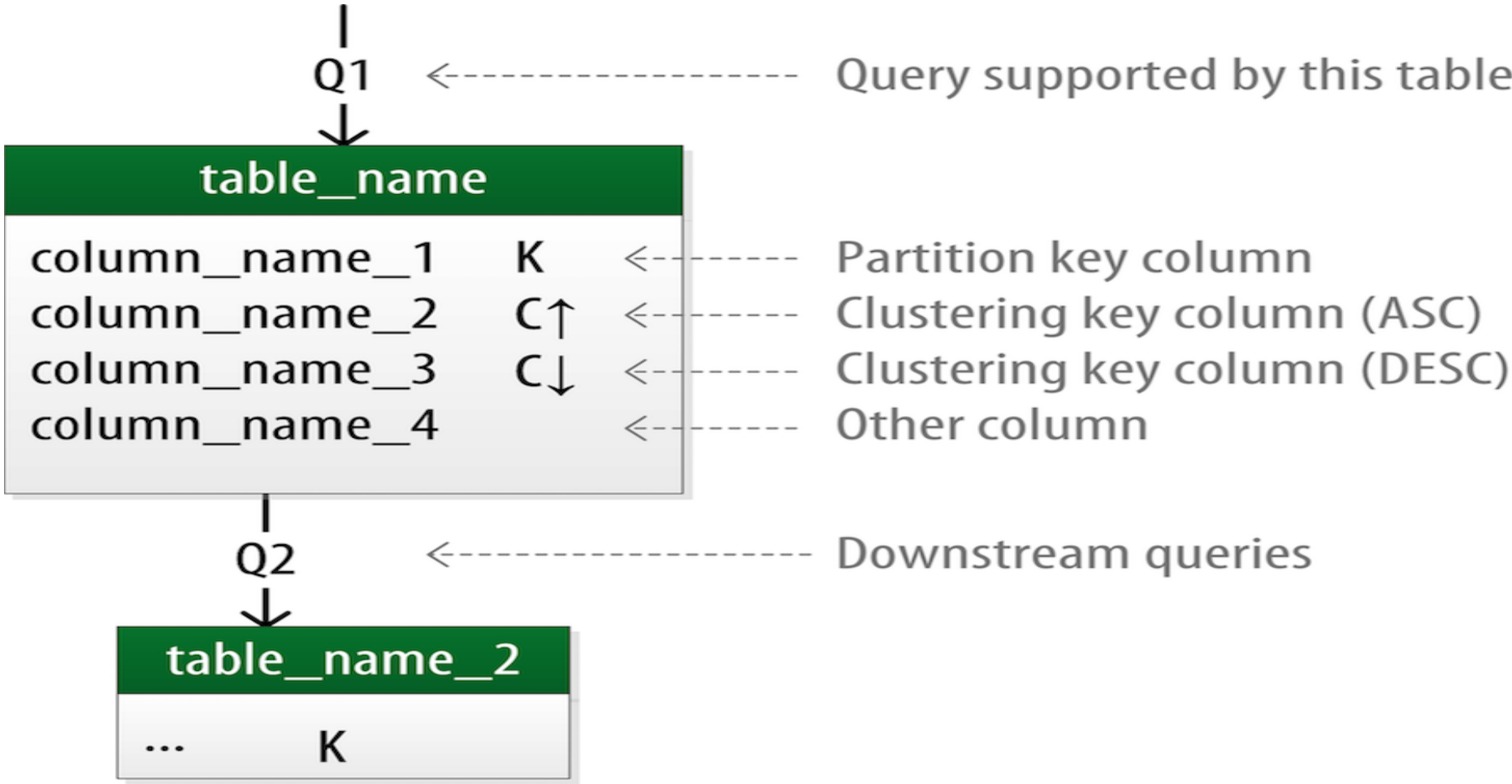
# Logical Data Modeling

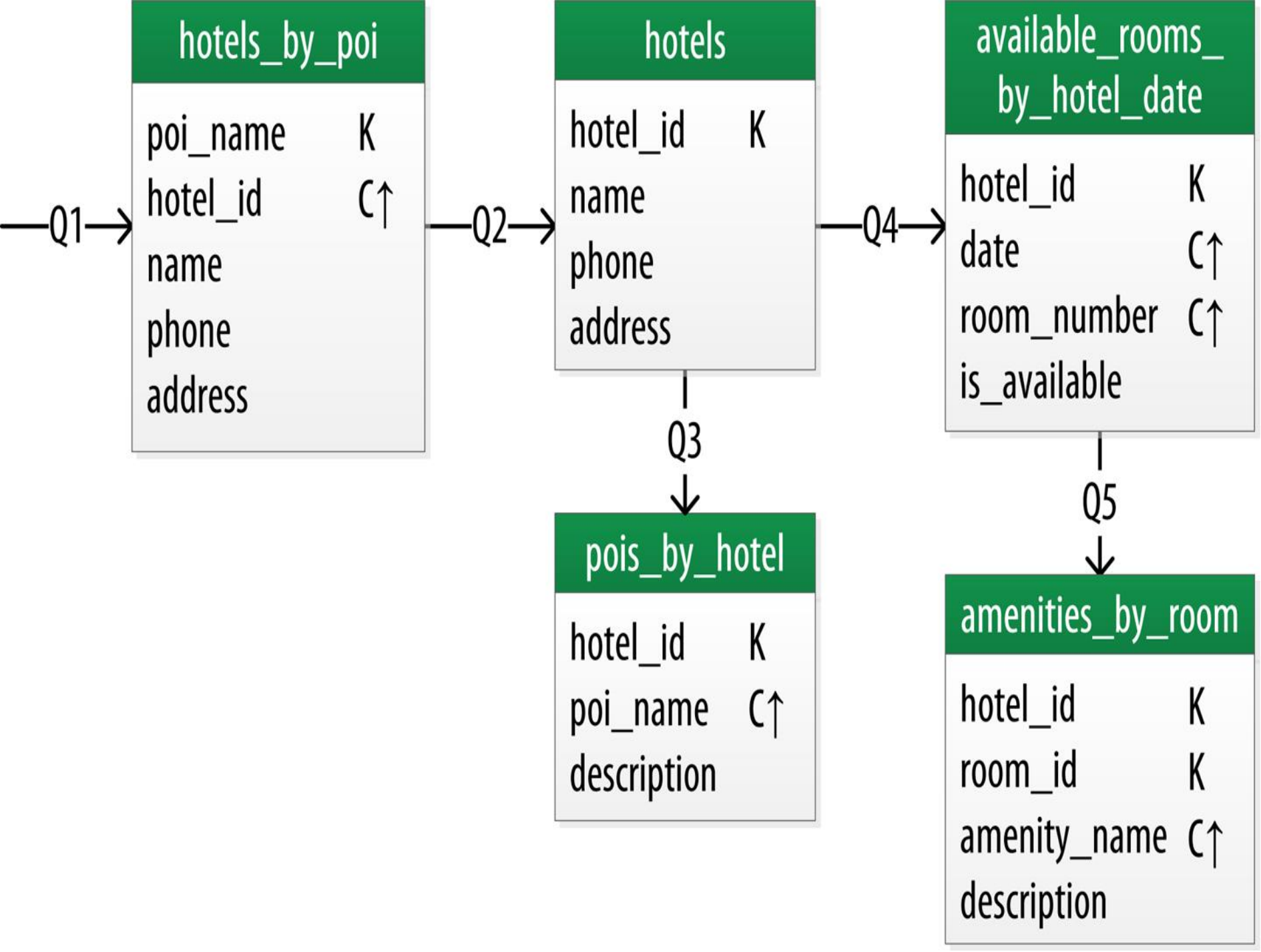
- To name each table, identify the primary entity type for which you are querying, and use that to start the entity name.
- If you are querying by attributes of other related entities, you append those to the table name, separated with _by_; for example, hotels_by_poi.
- Next, identify the primary key 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.

# Logical Data Modeling

- **INTRODUCING CHEBOTKO DIAGRAMS**
- Several individuals within the Cassandra community have proposed notations for capturing data models in diagrammatic form.
- We've elected to use a notation popularized by Artem Chebotko that provides a simple, informative way to visualize the relationships between queries and tables in your designs.

# CHEBOTKO DIAGRAMS




hotels_by_poi
poi_name K
hotel_id C↑
name
phone
address
Q1
Q2
hotels
hotel_id K
name
phone
address
Q3
pois_by_hotel
hotel_id K
poi_name C↑
description
Q4
available_rooms_by_hotel_date
hotel_id K
date C↑
room_number C↑
is_available
Q5
amenities_by_room
hotel_id K
room_id K
amenity_name C↑
description

# Hotel Logical Data Model

- 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 the table hotels_by_poi.
- You're searching by a named point of interest, so that is a clue that the point of interest should be a part of the primary key.
- 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 your 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.

# Hotel Logical Data Model

- Let’s also assume that according to your application workflow, your user will provide a name of a point of interest, but would benefit from seeing the description of the point of interest alongside hotel results.
- Therefore you include the poi_description as a column in the hotels_by_poi table, and designate this value as a static column since the point of interest description is the same for all rows in a partition

# Hotel Logical Data Model

- Now for the second query (Q2), you'll need a table to get information about a specific hotel. One approach would be to put all of the attributes of a hotel in the hotels_by_poi table, but you choose to add only those attributes required by your application workflow.
- From the workflow diagram, you note 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.

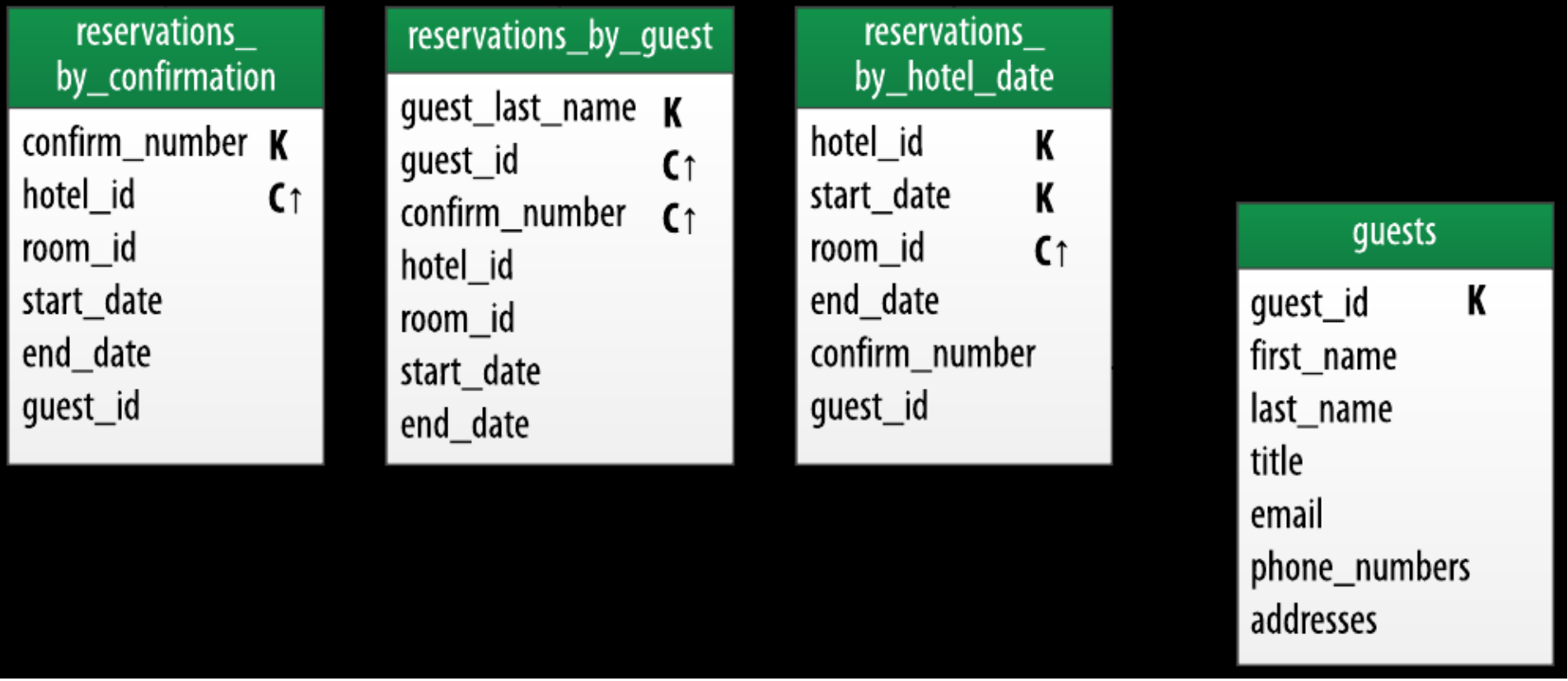
# Hotel Logical Data Model

- Another option would be 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 you did previously, you add the point of interest name as a clustering key to guarantee uniqueness.

# Hotel Logical Data Model

- At this point, let's now consider how to support query Q4 to help your users 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.
- You use the hotel_id as a primary key to group room data for each hotel on a single partition, which should help your search be super fast. Let's call this the available_rooms_by_hotel_date table.

# Reservation Logical Data Model



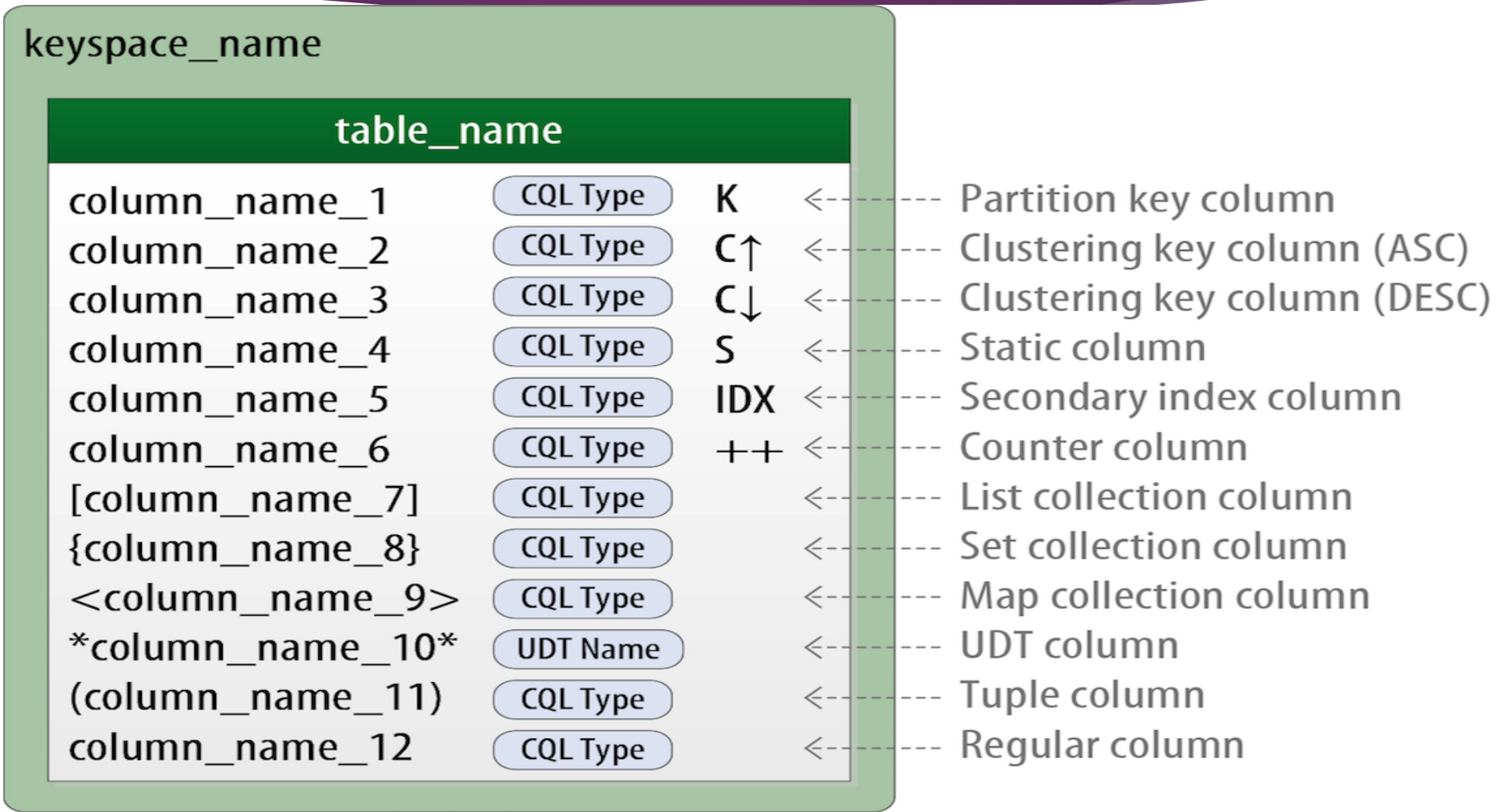
# Reservation Logical Data Model

- In order to satisfy Q6, the reservations_by_confirmation table supports the lookup of reservations by a unique confirmation number provided to the customer at the time of booking.
- If the guest doesn't have the confirmation number, 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.

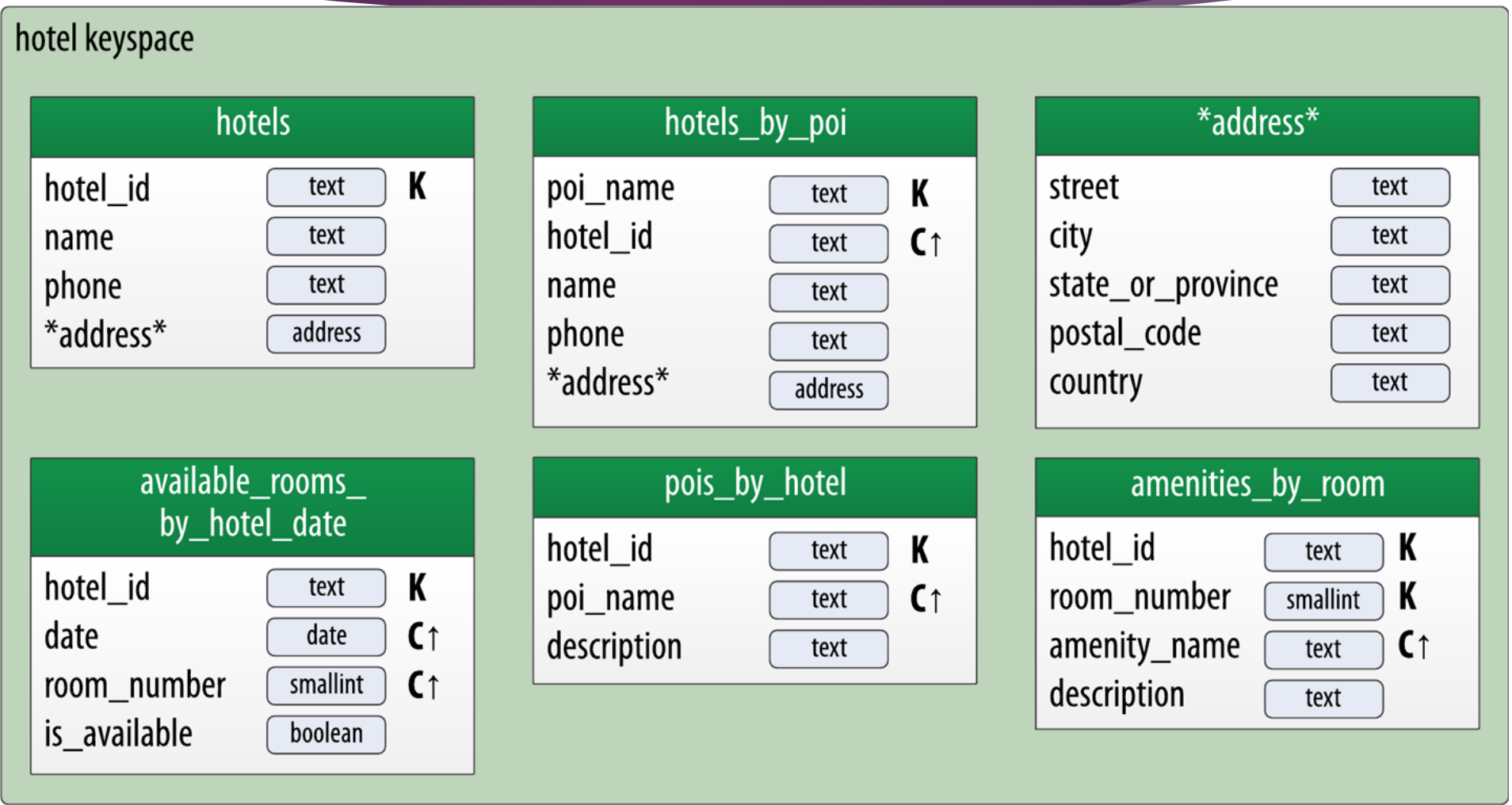
# Reservation Logical Data Model

- 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 the dates the hotel is sold out or undersold. Q8 supports the retrieval of reservations for a given hotel by date.

# Physical Data Modeling



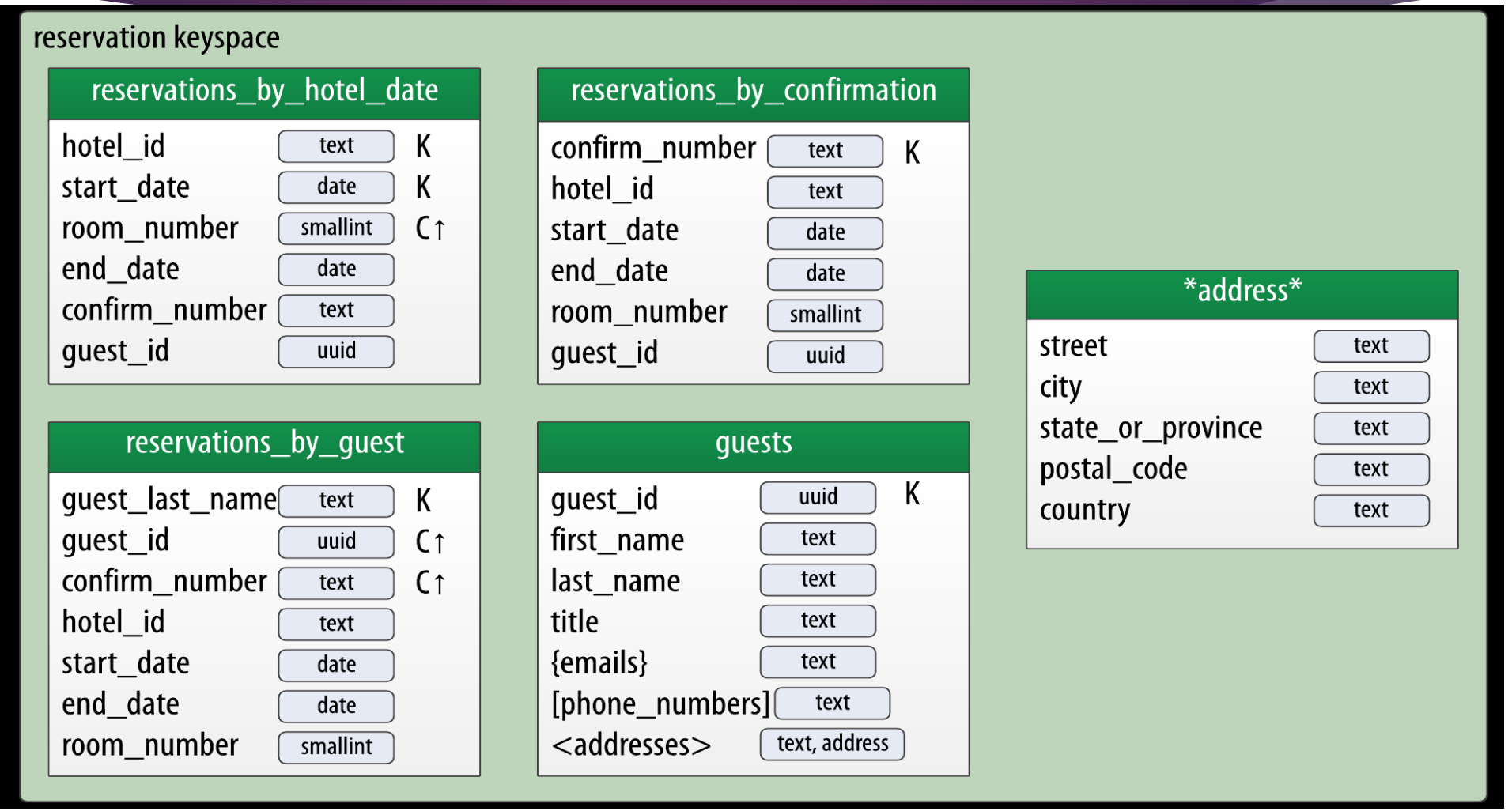
# Hotel Physical Data Model



# Hotel Physical Data Model

- To keep the design relatively simple, you 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.

# Reservation Physical Data Model



# Evaluating and Refining

- **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 two billion cells per partition, but you'll likely run into performance issues before reaching that limit. The recommended size of a partition is not more than 100,000 cells.

# Evaluating and Refining

- **Calculating Partition Size**
- In order to calculate the size of your partitions, you use the following formula:
- $Nv = Nr\ (Nc - Npk - Ns) + Ns$
- The number of values (or cells) in the partition (N ) is equal to the number of static columns (N ) plus the product of the number of rows (N ) and the number of of values per row.
- The number of values per row is defined as the number of columns (N ) minus the number of primary key columns (N ) and static columns (N ).

# Evaluating and Refining

- **Calculating Partition Size**
- The number of columns tends to be relatively static, although as you have seen, it is quite 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

# Evaluating and Refining

- **Calculating Partition Size**
- Let's take a look at one of your tables to analyze the partition size. Because it has a wide partition design with one partition per hotel, you choose the available_rooms_by_hotel_date table.
- The table has four columns total (N = 4), including three primary key columns (N = 3) and no static columns (N = 0). Plugging these values into the formula, you get:
- Nv = Nr (4 – 3 – 0) + 0 = 1Nr

# Evaluating and Refining

- 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, you make some estimates based on the application you're designing.
- The table is storing a record for each room, in each of your hotels, for every night.
- Let's assume that your system will be used to store 2 years of inventory at a time, and there are 5,000 hotels in the system, with an average of 100 rooms in each hotel

# Evaluating and Refining

- Since there is a partition for each hotel, the estimated number of rows per partition is as follows:
- Nr = 100 rooms /hotel × 730 days = 73, 000 rows
- This relatively small number of rows per partition is not an issue, but the number of cells may be.

# Evaluating and Refining

- **Calculating Size on Disk**
- In addition to calculating the size of your partitions, 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, you use the following formula to determine the size S of a partition:
- $St = \Sigma\ I\ sizeOf(cki) + \Sigma jsizeOf(csj) + Nr \times (\Sigma ksizeOf(crk) + \Sigma lsizeOf(ccl)) + Nv \times sizeOf(tavg)$

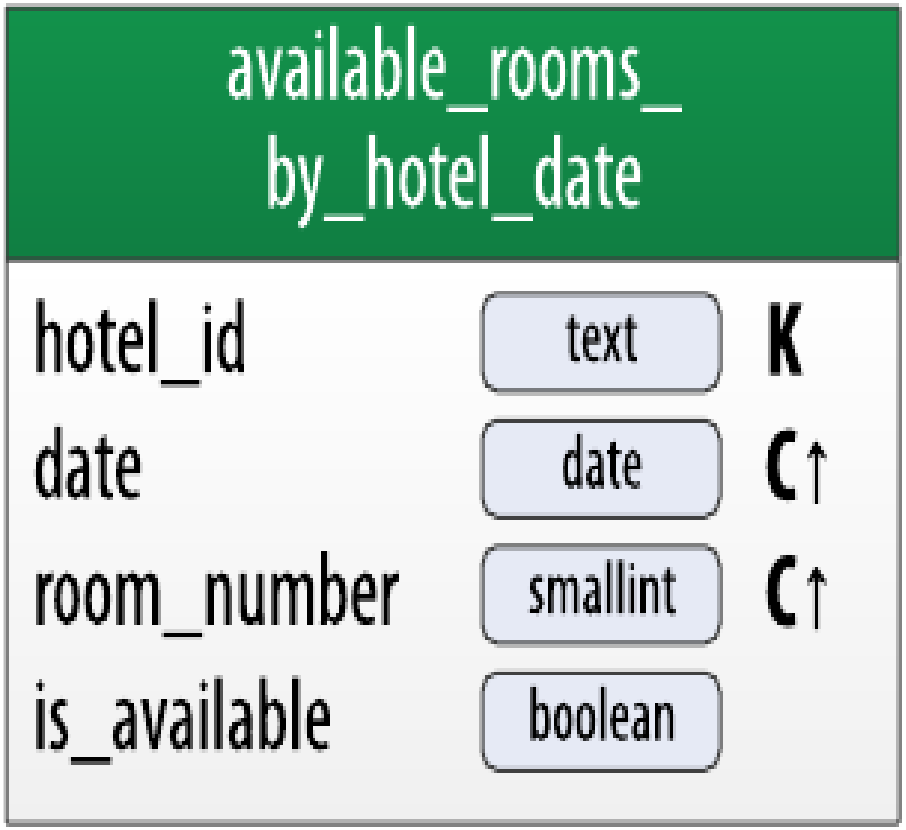
# Evaluating and Refining

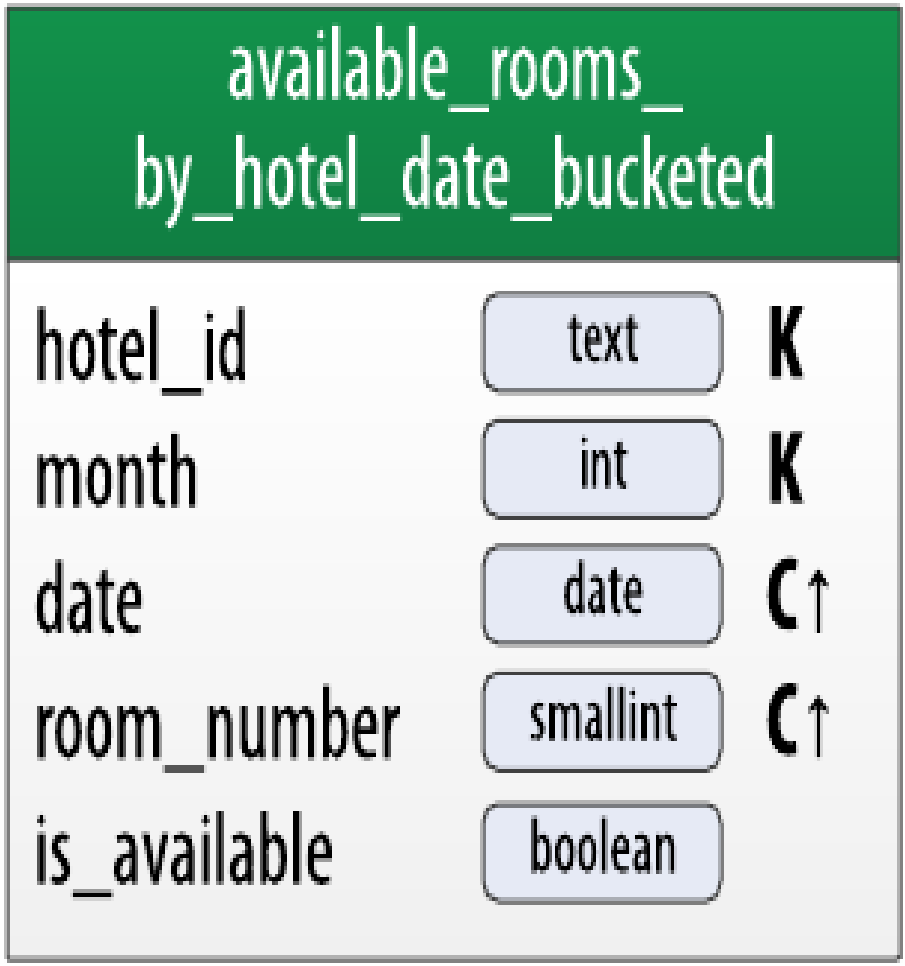
- This is a bit more complex than the previous formula, but let's break it down a bit at a time, starting with the notation:
- In this formula, c refers to partition key columns, c to static columns, c to regular columns, and c to clustering columns.
- The term refers to the average number of bytes of metadata stored per cell, such as timestamps. It is typical to use an estimate of 8 bytes for this value.
- You recognize the number of rows N and number of values N from previous calculations.
- The *sizeOf()* function refers to the size, in bytes, of the CQL data type of each referenced column.

# Evaluating and Refining

- The first term asks you to sum the size of the partition key columns. For this design, the available_rooms_by_hotel_date table has a single partition key column, the hotel_id, which you chose to make of type text.
- Assuming your 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 your static columns. This table has no static columns, so in your case this 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.

# Breaking Up Large Partitions



# Breaking Up Large Partitions

- 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.

# Breaking Up Large Partitions

- 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.
- 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

# Defining Database Schema

- 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, poi_description text STATIC, 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';

# Defining Database Schema

- 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';

# Defining Database Schema

- 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';

# Defining Database Schema

- 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';

# 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
- );

# here is the schema for the reservation keyspace:

- 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';

# here is the schema for the reservation keyspace:

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

# here is the schema for the reservation keyspace:

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

# here is the schema for the reservation keyspace:

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