DEFINING THE SCHEMA

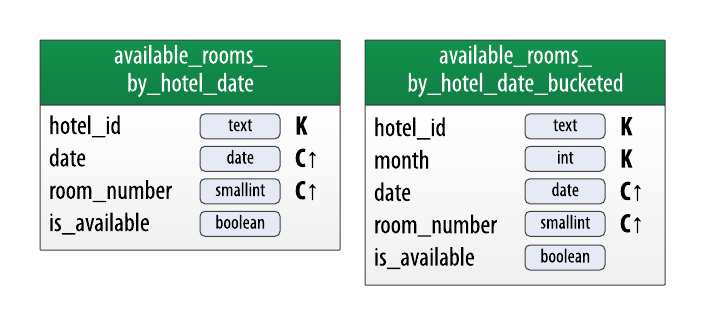
**Dividing the large partitions :**

The objective is to create tables that, in the event that a single partition is touched by a query, can yield the necessary data, or in the absence of that, the fewest number of partitions that can be used. Nonetheless, it is feasible to create broad partition-style tables that go close to Cassandra's inherent limitations, as seen by the examples. Table sizing analysis can identify potentially excessively big partitions in terms of number of items, disk size, or both.

The process of dividing a huge partition is as simple as adding a new column to the partition key. For the most part, it will be sufficient to move one of the current columns into the partition key. Adding a column to the table to serve as a sharding key is a third approach, but this requires more application logic.

Keeping with the available rooms example, each partition would then reflect the availability of rooms at a particular hotel on a certain date if the date column was added to the partition key for the table. Since the data for consecutive days will probably be on different nodes, this will undoubtedly result in divisions that are much smaller—possibly too small.

Another method for dividing the data into segments of a manageable size is called bucketing. To bucketize the table, for instance, you could add a month column (perhaps expressed as an integer) to the partition key. The figure below compares the current design to the original. The month column offers a useful method of organizing relevant data in a division that won't grow too big, even though it partially duplicates the date.



An alternative would be to append the room\_id to the partition key, which would allow each partition to reflect the room's availability on all days, if you were really committed to maintaining a broad partition design. As no query was found that inquired about the availability of a particular room, the first or second design method is best suited for the requirements of the application.

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 employe.details (

name,

id number,

mobile no.

salary given,

address,

country text );

CREATE TABLE employe,details\_by\_poi (

poi\_name text,

employe\_id text,

name text,

phone text,

address frozen<address>,

PRIMARY KEY ((poi\_name), employee\_id) )

WITH comment = ‘Q1. Find Employee details’

AND CLUSTERING ORDER BY (employee\_id ASC) ;

CREATE TABLE employee.details (

id text PRIMARY KEY,

name text,

phone text,

address frozen<address>,

pois set )

WITH comment = ‘Q2. Find information about a employee’;

CREATE TABLE employe.pois\_by\_employee (

poi\_name text,

employee\_id text,

description text,

PRIMARY KEY ((employe\_id), poi\_name) )

WITH comment = Q3. Find a employee’;

CREATE TABLE employe\_employee.details\_text (

employee\_id text,

date date,

id\_number smallint,

salary given boolean,

PRIMARY KEY ((employee\_id), date, salary given) )

WITH comment = ‘Q4. Find available employee by salary’;

CREATE TABLE employee.details\_by\_salary (

employee\_id text,

id\_number smallint,

salary boolean,

description text,

PRIMARY KEY ((employee\_id, mobile\_number), salary) )

WITH comment = ‘Q5. Find salary for a employee’;

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.