Designing for Column Family Databases

06016414 and 06026207: NoSQL Database Systems

Instructor: Asst. Prof. Dr. Praphan Pavarangkoon

Outline

- Designing for Column Family Databases
  - Guidelines for Designing Tables
  - Guidelines for Indexing
- Lab Session

Designing for Column Family Databases

- Users drive the design of a column family database.
- Designers follow the lead of users.
- It is users who determine the questions that will be asked of the database application:
  - How many new orders were placed in the Northwest region yesterday?
  - When did a particular customer last place an order?
  - What orders are en route to customers in London, England?
  - What products in the Ohio warehouse have fewer than the stock keeping minimum number of items?
- Like other NoSQL databases, design starts with queries.

Designing for Column Family Databases (cont.)

- Queries provide information needed to effectively design column family databases:
  - Entities
  - Attributes of entities
  - Query criteria
  - Derived values

Designing for Column Family Databases (cont.)

- Column family databases are implemented differently than relational databases.
- Thinking they are essentially the same could lead to poor design decisions.
- It is important to understand:
  - Column family databases are implemented as sparse, multidimensional maps.
  - Columns can vary between rows.
  - Columns can be added dynamically.
  - Joins are not used; data is denormalized instead.

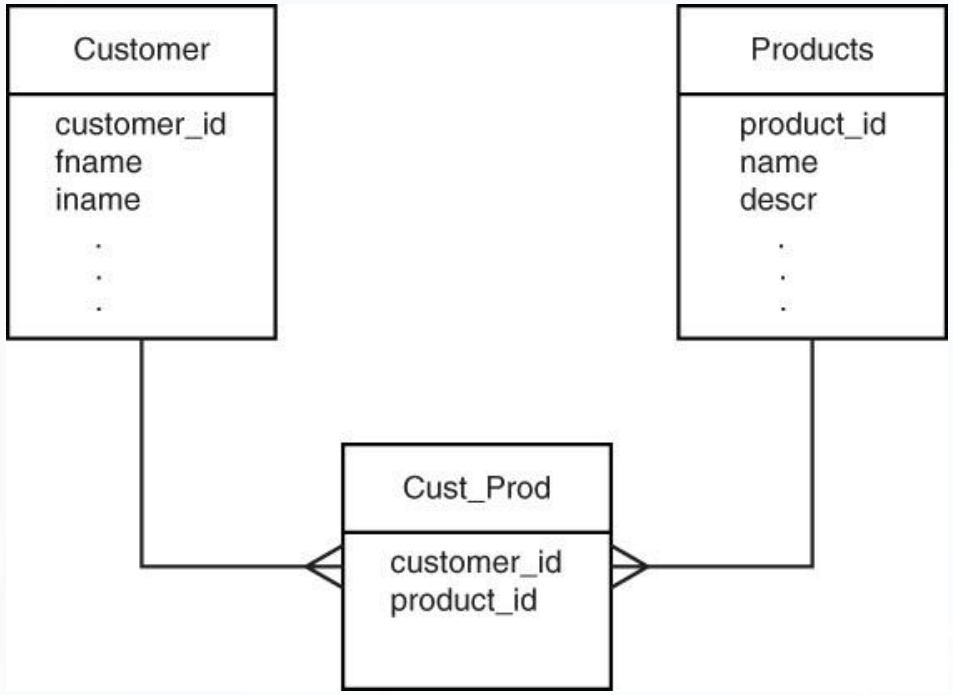
Guidelines for Designing Tables

- One of your first design decisions is to determine the tables in your schema.
- The following are several guidelines to keep in mind when designing tables:
  - Denormalize instead of join.
  - Make use of valueless columns.
  - Use both column names and column values to store data.
  - Model an entity with a single row.
  - Avoid hotspotting in row keys.
  - Keep an appropriate number of column value versions.
  - Avoid complex data structures in column values.

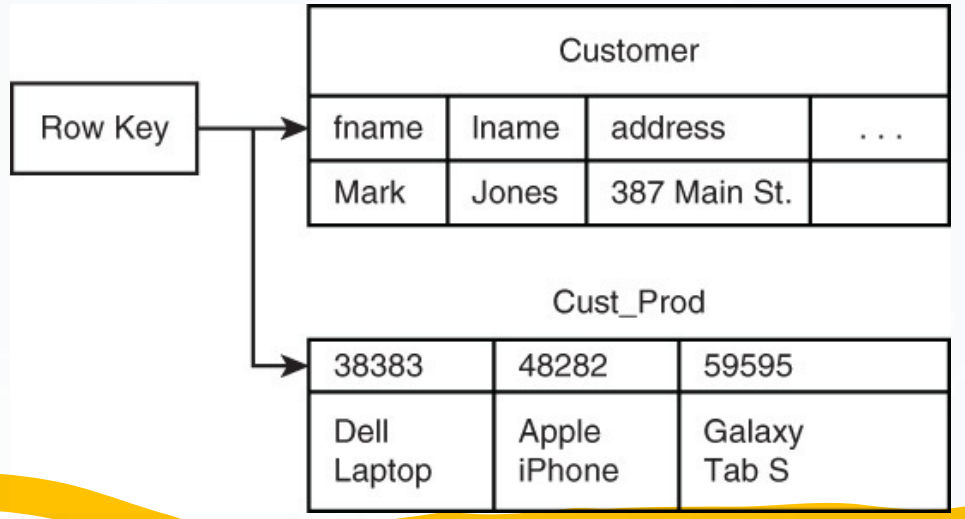
Denormalize Instead of Join

- Tables model entities, so it is reasonable to expect to have one table per entity.
- Column family databases often need fewer tables than their relational counterparts.
- This is because column family databases denormalize data to avoid the need for joins.

Denormalize Instead of Join (cont.)



Relational databases



Column family database

ใช้ประโยชน์จาก column ที่ไม่มีค่า

Make Use of Valueless Columns

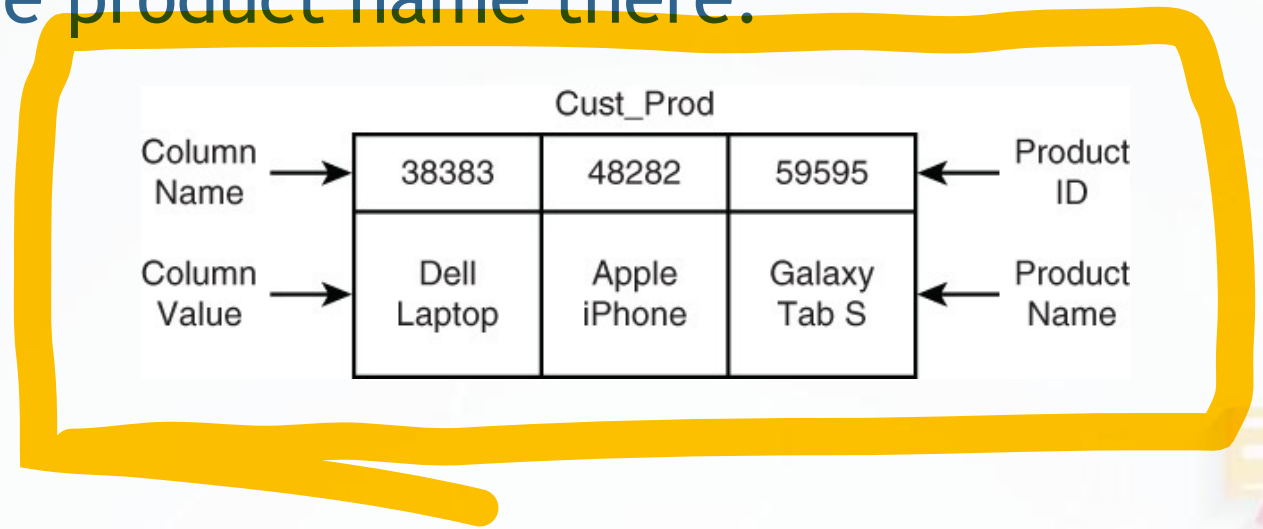
- Instead of having a column with a name like 'ProductPurchased1' with value 'PR_B1839', the table simply stores the product ID as the column name.
- Of course, you could store a value associated with a column name.
- If a column name indicates the presence or absence of something, you could assign a *T* or *F* to indicate true or false.
- This, however, would take additional storage without increasing the amount of information stored in the column.

Use Both Column Names and Column Values to Store Data

- A variation on the use of valueless columns uses the column value for denormalization.
- For example, in a database about customers and products, the features of the product, such as description, size, color, and weight, are stored in the products table.
- If your application users want to produce a report listing products bought by a customer, they probably want the product name in addition to its identifier.
- Because you are dealing with large volumes of data (otherwise you would not be using a column family database), you do not want to join or query both the customer and the product table to produce the report.

Use Both Column Names and Column Values to Store Data (cont.)

- The customer table includes a list of column names indicating the product ID of items purchased by the customer.
- Because the column value is not used for anything else, you can store the product name there.



Use Both Column Names and Column Values to Store Data (cont.)

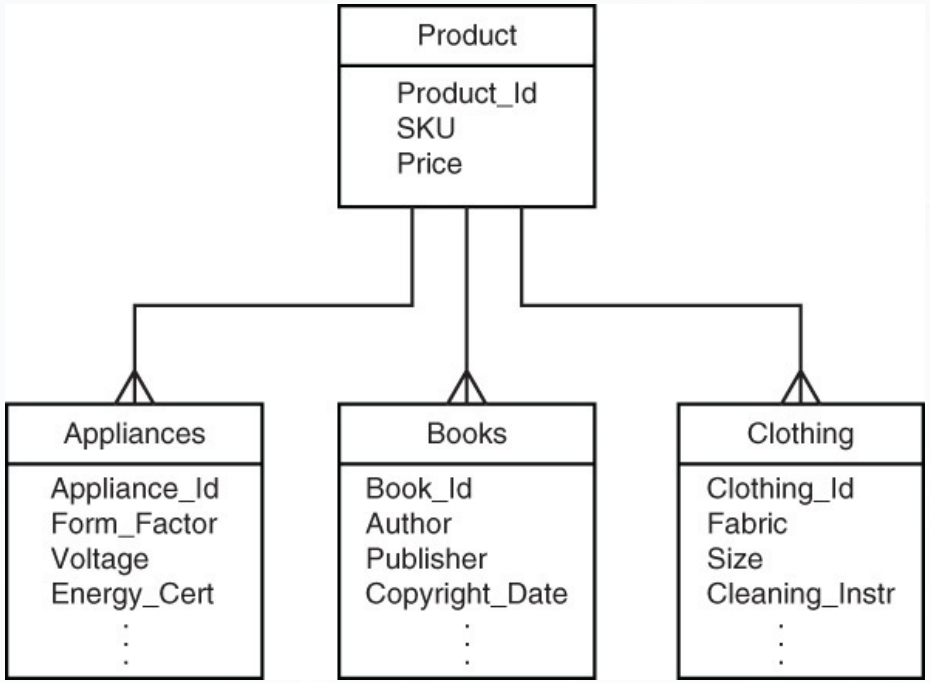
- Keeping a copy of the product name in the customer table will increase the amount of storage used. That is one of the downsides of denormalized data.
- The benefit, however, is that the report of customers and the products they bought is produced by referencing only one table instead of two.
- In effect, you are trading the need for additional storage for improved read performance.

เพิ่มประสิทธิภาพในการ query (???????)

Model an Entity with a Single Row

- A single entity, such as a particular customer or a specific product, should have all its attributes in a single row.
- This can lead to cases in which some rows store more column values than others, but that is not uncommon in column family databases.

Model an Entity with a Single Row (cont.)



- Column family databases do not provide the same level of transaction control as relational databases.
- Typically, writes to a row are atomic. If you update several columns in a table, they will all be updated, or none of them will be.

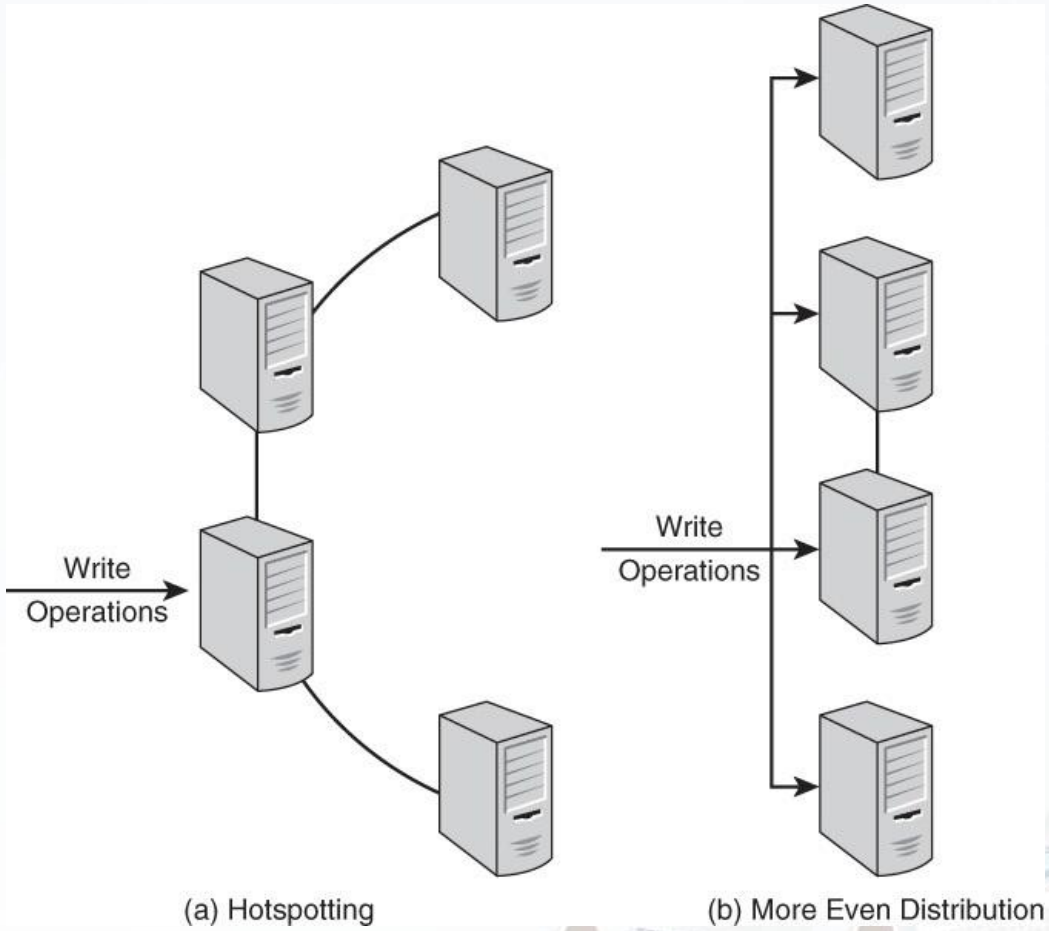
Avoid Hotspotting in Row Keys

- Distributed systems enable you to take advantage of large numbers of servers to solve problems.
- It is inefficient to direct an excessive amount of work at one or a few machines while others are underutilized.
- Hotspotting occurs when many operations are performed on a small number of servers.

Avoid Hotspotting in Row Keys (cont.)

- HBase uses lexigraphic ordering of rows.
- The data is stored in a file in sequential order.
- As HBase loads each record, it will likely write it to the same server that received the prior record and to a data block near the data block of the prior record.

- hash ข้อมูลแล้วเอามาจัดเรียง
- อะไรไม่รู้



Avoid Hotspotting in Row Keys (cont.)

- You can prevent hotspotting by hashing sequential values generated by other systems.
- Alternatively, you could add a random string as a prefix to the sequential value.
- This would eliminate the effects of the lexicographic order of the source file on the data load process.

Keep an Appropriate Number of Column Value Versions

- HBase enables you to store multiple versions of a column value.
- Column values are timestamped so you can determine the latest and earliest values.
- Like other forms of version control, this feature is useful if you need to roll back changes you have made to column values.
- HBase enables you to set a minimum and maximum number of versions.
- It will not remove versions if it would leave a column value with less than the minimum number of versions.

Keep an Appropriate Number of Column Value Versions (cont.)

- When the number of versions exceeds the maximum number of versions, the oldest versions are removed during data compaction operations.

Column Family		
Column $Name_1$	Column $Name_2$	Column $Name_3$
$value_{1a}$: $timestamp_{1a}$	$value_{2a}$: $timestamp_{2a}$	$value_{3a}$: $timestamp_{3a}$
$value_{1b}$: $timestamp_{1b}$	$value_{2b}$: $timestamp_{2b}$	$value_{3b}$: $timestamp_{3b}$
$value_{1c}$: $timestamp_{1c}$	$value_{2c}$: $timestamp_{2c}$	$value_{3c}$: $timestamp_{3c}$

Avoid Complex Data Structures in Column Values

- A JSON document about a customer, for example, might contain an embedded document storing address information, such as the following:

```
{
    "customer_id":187693,
    "name": "Kiera Brown",
    "address" : {
              "street" : "1232 Sandy Blvd.",
              "city" :  "Vancouver",
              "state" :  "Washington",
              "zip" :  "99121"
                 },
   "first_order" : "01/15/2013",
   "last_order" : " 06/27/2014"
}
```

Avoid Complex Data Structures in Column Values (cont.)

- Using separate columns for each attribute makes it easier to apply database features to the attributes.
- Also, separating attributes into individual columns allows you to use different column families if needed.
- One of the most important considerations with regard to performance is indexing.

Guidelines for Indexing

- Indexes allow for rapid lookup of data in a table.
- For example, if you want to look up customers in a particular state, you could use a statement such as the following (in Cassandra query language, CQL):

```
SELECT
    fname, lname
FROM
    customers
WHERE
     state = 'OR';
```

- A database index functions much like the index in a book.
  - You can look up an entry in a book index to find the pages that reference that word or term.

index คือโครงสร้างข้อมูลที่ใช้เพิ่มความเร็วในการค้นหาข้อมูล (ยังไง???)

Guidelines for Indexing (cont.)

- Similarly, in column family databases, you can look up a column value, such as state abbreviation, to find rows that reference that column value.
- It is helpful to distinguish two kinds of indexes: primary and secondary. primary indexes สร้างขึ้นอัตโนมัติจาก primary key
  - Primary indexes are indexes on the row keys of a table. They are automatically maintained by the column family database system.
  - Secondary indexes are indexes created on one or more column values. index ที่ไม่ใช่ primary key มีได้มากกว่า 1 ตัวใน 1 ตาราง
- Not all column family databases provide automatically managed secondary indexes, but you can create and manage tables as secondary indexes in all column family database systems.

When to Use Secondary Indexes Managed by the Column Family Database System

- As a general rule, if you need secondary indexes on column values and the column family database system provides automatically managed secondary indexes, then you should use them.
- The primary advantage of using automatically managed secondary indexes is they require less code to maintain than the alternative.
- In Cassandra, for example, you could create an index in CQL using the following statement:

```
CREATE INDEX state ON customers(state);
```

When to Use Secondary Indexes Managed by the Column Family Database System (cont.)

- Cassandra will then create and manage all data structures needed to maintain the index.
- It will also determine the optimal use of indexes.
- For example, if you have an index on state and last name column values and you queried the following, Cassandra would choose which index to use first:

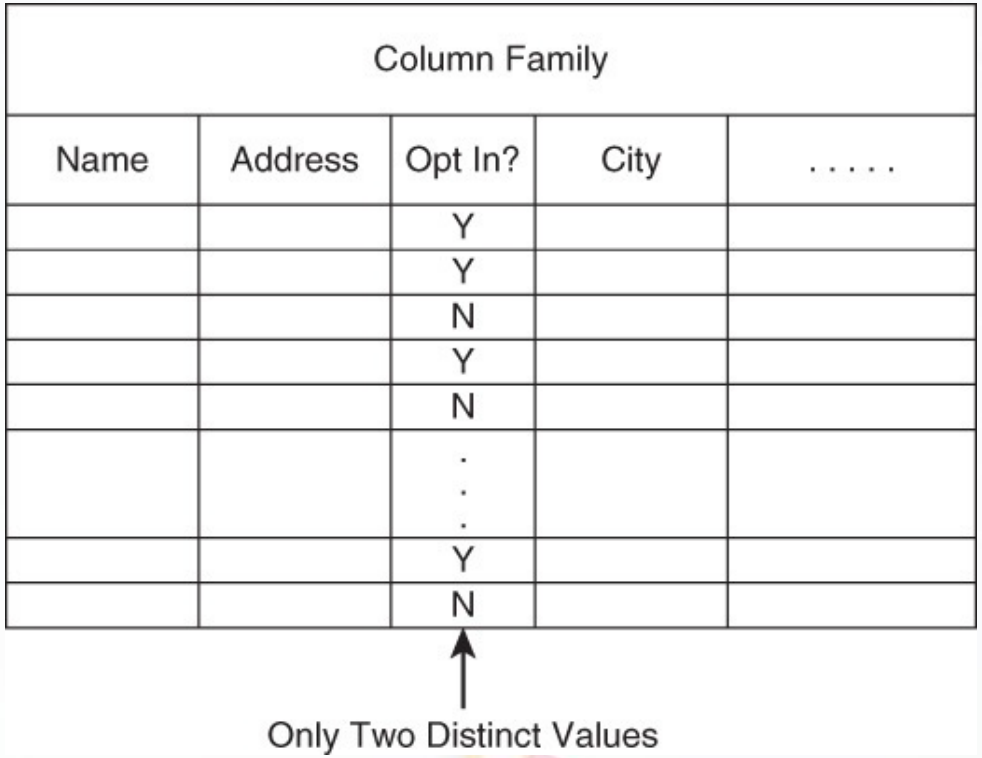
```
SELECT
    fname, lname
FROM
    customers
WHERE
    state = 'OR'
AND
    lname = 'Smith'
```

When to Use Secondary Indexes Managed by the Column Family Database System (cont.)

- There are times when you should not use automatically managed indexes.
- Avoid, or at least carefully test, the use of indexes in the following cases:
  - There is a small number of distinct values in a column.
  - There are many unique values in a column.
  - The column values are sparse.

When to Use Secondary Indexes Managed by the Column Family Database System (cont.)

- When the number of distinct values in a column (known as the cardinality of the column) is small, indexes will not help performance much—it might even hurt.



When to Use Secondary Indexes Managed by the Column Family Database System (cont.)

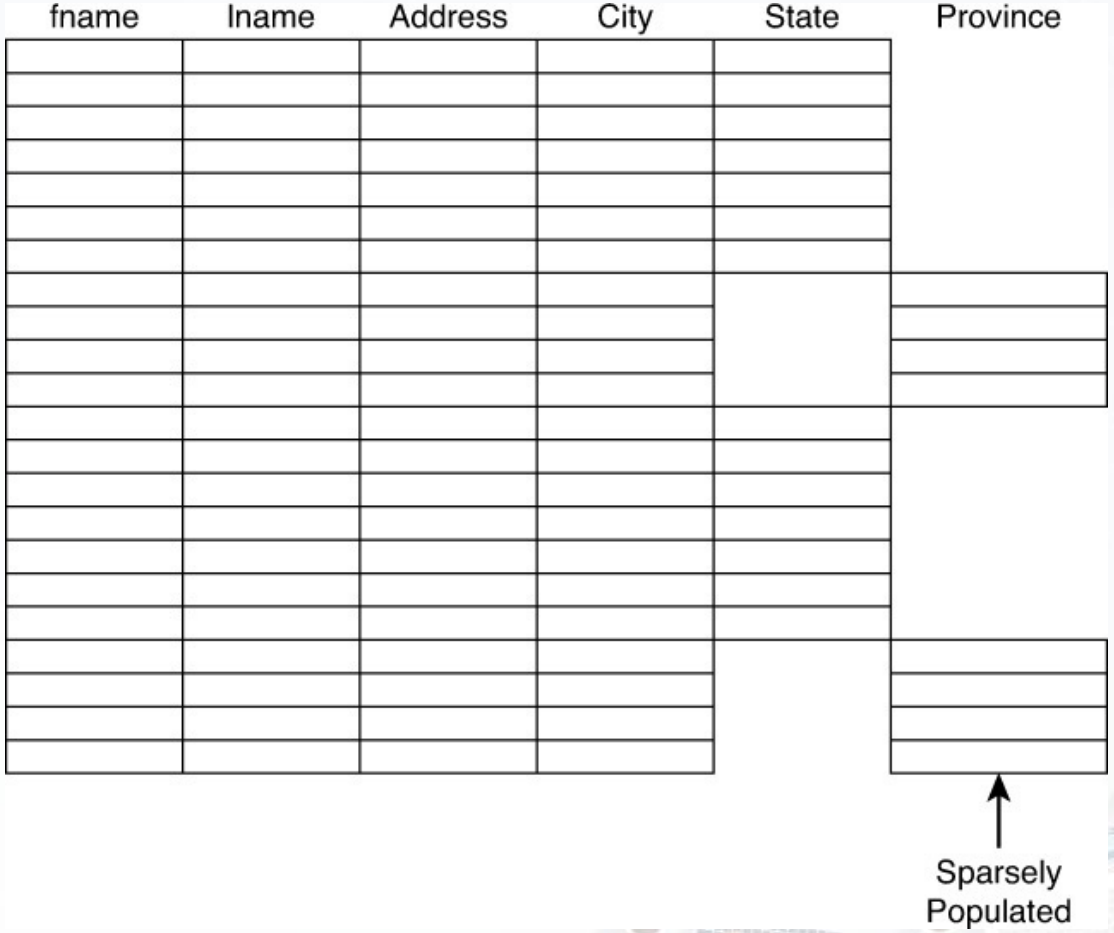
- At the other end of the cardinality spectrum are columns with many distinct values, such as street addresses and email addresses.
- Again, automatically managed indexes may not help much here.

Column Family				
Name	Address	City	State	Email
				ralken@gmail.com
				iman123@gmail.com
				dans37@yahoo.com
				marypdx@gmail.com
				gwashington@aol.com
				kcameron@future.com
				info@mybbiz.com . . . . . .

Many Distinct Values

When to Use Secondary Indexes Managed by the Column Family Database System (cont.)

- In cases where many of the rows do not use a column, a secondary index may not help.



When to Create and Manage Secondary Indexes Using Tables

- If your column family database system does not support automatically managed secondary indexes or the column you would like to index has many distinct values, you might benefit from creating and managing your own indexes.
- Indexes created and managed by your application use the same table, column family, and column data structures used to store your data.
- Instead of using a statement such as CREATE INDEX to make data structures managed by the database system, you explicitly create tables to store data you would like to access via the index.

When to Create and Manage Secondary Indexes Using Tables (cont.)

- Let’s return to the customer and product database.
- Your end users would like to generate reports that list all customers who bought a particular product.
- They would also like a report on particular products and which customers bought them.

When to Create and Manage Secondary Indexes Using Tables (cont.)

- In the first situation, you would want to quickly find information about a product, such as its name and description.
  - The existing product table meets this requirement.
- Next, you would want to quickly find all customers who bought that product.
  - A time-efficient way to do this is to keep a table that uses the product identifier as the row key and uses customer identifiers as column names.
  - The column values can be used to store additional information about the customers, such as their names.
- The necessary data is stored in the Cust_by_Prod table.

When to Create and Manage Secondary Indexes Using Tables (cont.)

สร้างตารางเพิ่มแทนการทำ index

Customer

Row key	fname	lname	street	city	state
123	Jane	Smith	387 Main St	Boise	ID
287	Mark	Jones	192 Wellfleet Dr	Austin	TX
1987	Harsha	Badal	298 Commercial St	Provincetown	MA
2405	Senica	Washington	98 Morton Ave	Windsor	CT
3902	Marg	O'Malley	981 Circle Dr	Santa Fe	NM

Product

Row key	name	descr	qty_avail	category	
38383	Dell Latitude E6410	Laptop with ...	124	Computer	
48282	Apple iPhone	iPhone 6 with ...	345	Phone	
59595	Galaxy Tab S	Samsung tablet ...	743	Tablet	

Cust_by_Prod

Row key	123	287	1987	2405	3902
38383	Smith		Badal		
48282	Smith	Jones			O'Malley
59595				Washington	

Prod_by_Cust

Row key	38383	48282	59595		
123	Dell Latitude E6410	Apple iPhone			
287		Apple iPhone			
1987	Dell Latitude E6410				
2405			Galaxy Tab S		
3902		Apple iPhone			

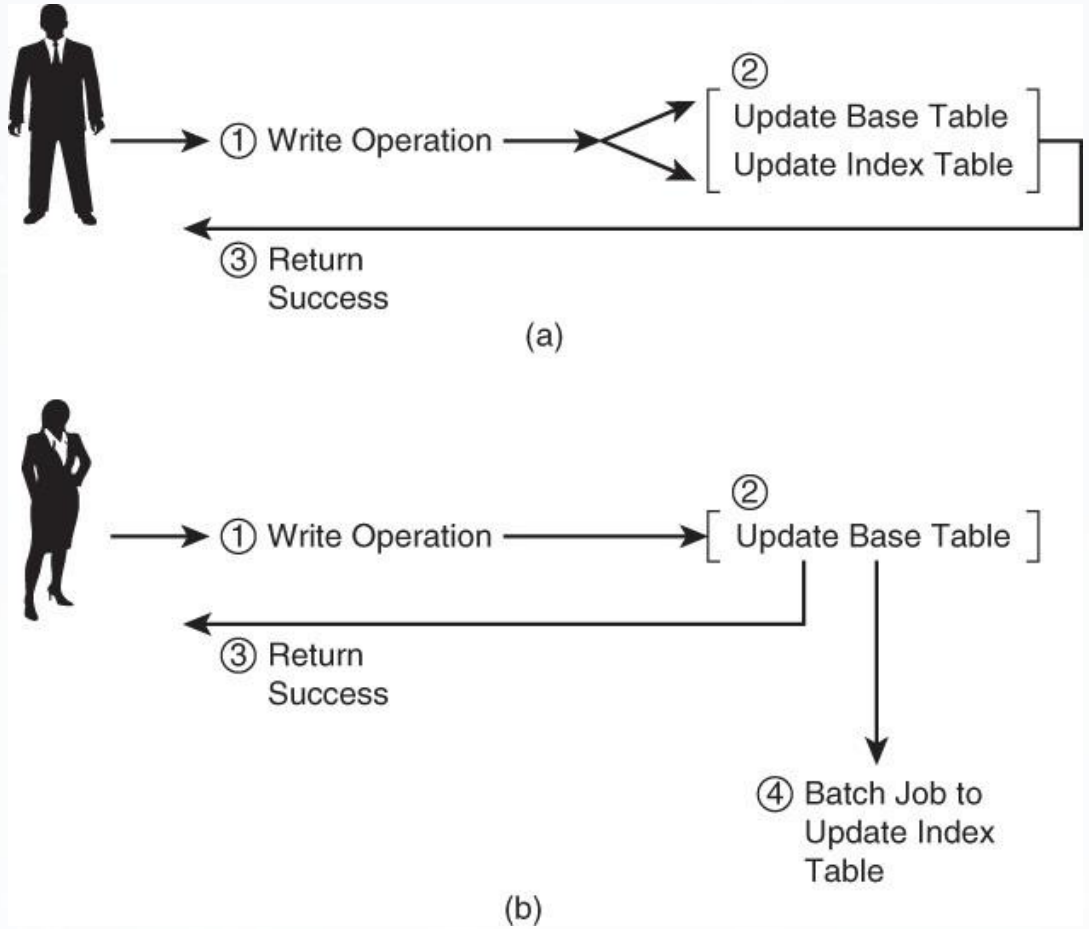
When to Create and Manage Secondary Indexes Using Tables (cont.)

- A similar approach works for the second report as well.
- To list all products purchased by a customer, you start with the customer table to find the customer identifier and any information about the customer needed for the report, for example, address, credit score, last purchase date, and so forth.
- Information about the products purchased is found in the Prod_by_Cust table.

When to Create and Manage Secondary Indexes Using Tables (cont.)

- When using tables as indexes, you will be responsible for maintaining the indexes.
- You have two broad options with regard to the timing of updates.
- You could update the index whenever there is a change to the base tables, for example, when a customer makes a purchase.
- Alternatively, you could run a batch job at regular intervals to update the index tables.

When to Create and Manage Secondary Indexes Using Tables (cont.)



Lab Session

- Tables, columns, data types, rows, partitions, keys, ordering
- Data Modeling in Apache Cassandra™ Fundamentals:
  - Learn how to create basic data models for Cassandra NoSQL database

Tables, columns, data types, rows, partitions, keys, ordering

- A *table* in Apache Cassandra™ shares many similarities with a table in a relational database.
  - It has named *columns* with *data types* and *rows* with *values*.
  - A *primary key* uniquely identifies a row in a table.
- There are also important differences.
  - In Cassandra, on one hand, a table is a set of rows containing values and, on the other hand, a table is also a set of partitions containing rows.
  - Specifically, each row belongs to exactly one partition and each partition contains one or more rows.
  - A primary key consists of a mandatory partition key and optional clustering key, where a partition key uniquely identifies a partition in a table and a clustering key uniquely identifies a row in a partition.

Tables, columns, data types, rows, partitions, keys, ordering (cont.)

- A table with *single-row partitions* is a table where there is exactly one row per partition.
  - A table with single-row partitions defines a primary key to be equivalent to a partition key.
- A table with *multi-row partitions* is a table where there can be one or more rows per partition.
  - A table with multi-row partitions defines a primary key to be a combination of both partition and clustering keys.
  - Rows in the same partition have the same partition key values and are *ordered* based on their clustering key values using the default ascendant order.

Tables, columns, data types, rows, partitions, keys, ordering (cont.)

- **Creating tables**
  - To create a table, Cassandra Query Language has the CREATE TABLE statement with the following simplified syntax:

```
CREATE TABLE [ IF NOT EXISTS ] [keyspace_name.]table_name
(
  column_name data_type [ , ... ]
  PRIMARY KEY (
   ( partition_key_column_name  [ , ... ] )
   [ clustering_key_column_name [ , ... ] ]
  )
)
[ WITH CLUSTERING ORDER BY
   ( clustering_key_column_name ASC|DESC [ , ... ] )
];
```

Tables, columns, data types, rows, partitions, keys, ordering (cont.)

- First, notice that a table is created within an existing keyspace.
  - If a *keyspace* name is omitted, the current working keyspace is used.
- Second, *keyspace*, table and column names can contain alphanumeric characters and underscores.
  - By default, names are case-insensitive, but case sensitivity can be forced by using double quotation marks around a name.
- Third, there are many CQL *data types*, including native, collection and user-defined data types.
  - For now, we will use some of the simplest and self-descriptive ones like TEXT, INT, FLOAT, and DATE.

Tables, columns, data types, rows, partitions, keys, ordering (cont.)

- Fourth, notice that there are additional parentheses around the partition key columns that can be omitted when the partition key has only one column (a.k.a. *simple partition key*) and are required when the partition key has more than one column (a.k.a. *composite partition key*).
- Finally, when a clustering key is defined, ordering is optionally specified in the last clause with ascendant order being the default.

Data Modeling in Apache Cassandra™ Fundamentals

- How to create tables
- How to handle table joins
- How to handle queries on non-primary key columns

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- **How to Create Tables**
  - Our user management system keeps track of users' profiles and supports authentication.
  - For authentication, when a user logs-in, we want to find their password based on their email address.
  - Once they login, we also want a unique identifier for the user.
  - So, for each user, we need this data.

Column Name	Column Type
email	text
password	text
user_id	UUID

  - We'll store this table in a keyspace named user_management.

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- Here are the commands to create the keyspace, set the default keyspace and create the table.

```
// Create the user_management keyspace
CREATE KEYSPACE user_management
  WITH REPLICATION = {
    'class' : 'SimpleStrategy',
    'replication_factor' : 1
  };
// Set user_management as the default keyspace
USE user_management;
// Create the user_credentials_by_email table
CREATE TABLE credentials_by_email (
  email      text,
  password   text,
  user_id    UUID,
  PRIMARY KEY(email)
);
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- The user_credentials_by_email table supports the user login use-case, but we need another table that keeps track of the user profile data.
- Here's the data we need for this table.

Column Name	Column Type
user_id	UUID
last_name	text
first_name	text

- When we create tables in Cassandra, we want to think about how we will use the table so that the table supports specific queries.

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- When we use this table, we want to retrieve the user profile data based on the user_id, so we make the user_id column the partition key.
- Since this table contains profile data and its partition key is user_id, we'll name this table profiles_by_id.
- Write and execute the CQL to create this table:

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- **What to do About Joining Tables**
  - In the previous step, we created a keyspace with two tables.
  - Let's review the keyspace.

```
DESCRIBE KEYSPACE user_management;
```

  - We see the two tables in the user_management keyspace share the user_id value.
  - When a user logs-in, we retrieve and verify the user's password, but we also retrieve the user_id value.
  - We can use this value to get the user's profile data.
  - If you are coming from a relational world, you might be tempted to try to join the two tables during the login query.
  - However, CQL does not support table joins.

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- In CQL, there are two ways to perform the join capability.
- The first is what we are doing with these tables - that is, we perform the join in the application.
- Let's insert some data into the tables and give it a try!

```
INSERT INTO credentials_by_email (email, password, user_id)
  VALUES(
    'aa@gmail.com',
    '@l3xL0v3$C@ss@ndr@',
    11111111-1111-1111-1111-111111111111
  );
INSERT INTO profiles_by_id (user_id, last_name, first_name)
    VALUES(
      11111111-1111-1111-1111-111111111111,
      'Artichoke',
      'Alex'
    );
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

```
INSERT INTO credentials_by_email (email, password, user_id)
  VALUES(
    'bb@gmail.com',
    'C@ss@ndr@R0ck$',
    22222222-2222-2222-2222-222222222222
  );
INSERT INTO profiles_by_id (user_id, last_name, first_name)
  VALUES(
    22222222-2222-2222-2222-222222222222,
    'Broccoli',
    'Bailey'
  );
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

```
INSERT INTO credentials_by_email (email, password, user_id)
  VALUES(
    'cc@gmail.com',
    'C@ss@ndr@43v3r',
    33333333-3333-3333-3333-333333333333
  );
INSERT INTO profiles_by_id (user_id, last_name, first_name)
  VALUES(
    33333333-3333-3333-3333-333333333333,
    'Cabbage',
    'Chris'
  );
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- Let's review the contents of the tables.

```
SELECT * FROM credentials_by_email;
SELECT * FROM profiles_by_id;
```

- Now, let's simulate Alex logging-in. First, we will query for Alex's user credentials. Then, assuming the password values match, we will retrieve Alex's profile.

```
// Simulate the login
SELECT * from credentials_by_email
  WHERE email = 'aa@gmail.com';
// Simulate retrieving the profile
SELECT * from profiles_by_id
  WHERE user_id = 11111111-1111-1111-1111-111111111111;
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- You try simulating Bailey logging-in by requesting Bailey's credentials followed by Bailey's profile.

- The point of this simulation is that the application can perform the join.

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- The second way to handle join capability in Cassandra is to perform the join *before* creating the table.
- For example, we could create a table, named users_by_email, with the following columns.

Column Name	Column Type
email	text
password	text
last_name	text
first_name	text

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- Write and execute the CQL to create the users_by_email table.

- We no longer need the user_id column. This approach has the benefit of requiring only one query when a user logs in.

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- **How to Handle Queries on Non-primary key Columns**
  - In the previous step we created the users_by_email table. Let's insert some rows.

```
INSERT INTO users_by_email (email, password, last_name, first_name)
  VALUES(
    'aa@gmail.com',
    '@l3xL0v3$C@ss@ndr@',
    'Artichoke',
    'Alex'
  );
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

```
INSERT INTO users_by_email  (email, password, last_name, first_name)
  VALUES(
    'bb@gmail.com',
    'C@ss@ndr@R0ck$',
    'Broccoli',
    'Bailey'
  );
INSERT INTO users_by_email (email, password, last_name, first_name)
  VALUES(
    'cc@gmail.com',
    'C@ss@ndr@43v3r',
    'Cabbage',
    'Chris'
  );
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- Now, let's write and execute a query to retrieve the row for cc@gmail.com.

```
SELECT * from users_by_email
  WHERE email = 'cc@gmail.com';
```

- As we mentioned in the previous step, this is the query the app would perform when a user logs-in.
- What if we wanted to find the user based on their first and last name? Based on our relational database experience, we might think we could perform a query like the following. Try it!

```
SELECT * FROM users_by_email
  WHERE last_name = 'Cabbage'
  AND first_name = 'Chris';
```

- We see that Cassandra cannot execute this query.

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- So, what if our app needs to perform this query?
- One secret to Cassandra data modeling is to understand that each query type may require its own table.
- In this case we will need to create a second table. We'll call the second table users_by_name. This table has the same rows as the users_by_email table, but it has a different partition key. Specifically, the partition key is the last_name and first_name columns.

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- See if you can create the users_by_name table with a compound partition key (by surrounding both column names in the primary key definition with parenthesis to designate the partition key) including the last and first names.
- Since we only need the password for the login use-case, we will exclude that column in this new table.

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- Let's add some rows into this table.

```
INSERT INTO users_by_name (last_name, first_name, email)
  VALUES(
    'Artichoke',
    'Alex',
    'aa@gmail.com'
  );
INSERT INTO users_by_name (last_name, first_name, email)
  VALUES(
    'Broccoli',
    'Bailey',
    'bb@gmail.com'
  );
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

```
INSERT INTO users_by_name (last_name, first_name, email)
  VALUES(
    'Cabbage',
    'Chris',
    'cc@gmail.com'
  );
```

- Now, see if you can query for Alex Artichoke.

```
SELECT * FROM users_by_name
  WHERE last_name = 'Artichoke'
  AND first_name = 'Alex';
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- You might think we're done, but this table has a problem.
- Consider adding a person who shares the name Alex Artichoke (what are the chances? :)), but has an email of aart@gmail.com.
- Let's insert this person's row.

```
INSERT INTO users_by_name
  (last_name, first_name, email)
  VALUES(
    'Artichoke',
    'Alex',
    'aart@gmail.com'
  );
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- Now, look at the contents of the entire table.

```
SELECT * FROM users_by_name;
```

- We see that the insert was really an upsert that merely changed the email address for Alex Artichoke. Or, in other words, we lost the row for Alex Artichoke with an email of aa@gmail.com;
- To avoid this undesired upsert behavior, we need to add a clustering column that makes the row unique. We can use email as the clustering column that makes the rows' primary key unique.

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

- Let's drop the users_by_name table, recreate it and re-add the users.

```
// Drop the old table
DROP TABLE users_by_name;
// Create the table with the new definition
CREATE TABLE users_by_name (
  last_name    text,
  first_name   text,
  email        text,
  PRIMARY KEY((last_name, first_name), email)
);
INSERT INTO users_by_name (last_name, first_name, email)
  VALUES(
    'Artichoke',
    'Alex',
    'aa@gmail.com'
  );
```

Data Modeling in Apache Cassandra™ Fundamentals (cont.)

```
INSERT INTO users_by_name (last_name, first_name, email)
  VALUES(
    'Broccoli',
    'Bailey',
    'bb@gmail.com'
  );
INSERT INTO users_by_name (last_name, first_name, email)
  VALUES(
    'Cabbage',
    'Chris',
    'cc@gmail.com'
  );
```

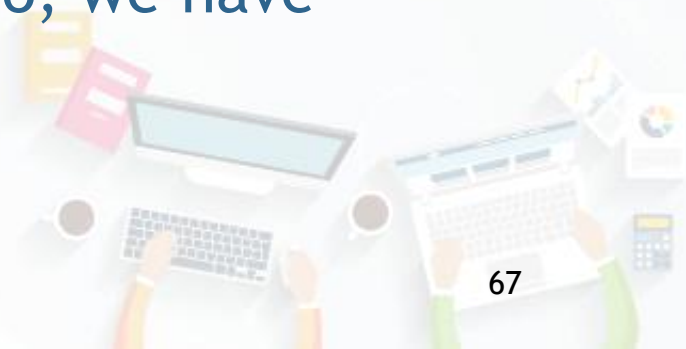
Data Modeling in Apache Cassandra™ Fundamentals (cont.)

```
INSERT INTO users_by_name
  (last_name, first_name, email)
  VALUES(
    'Artichoke',
    'Alex',
    'aart@gmail.com'
  );
```

- Finally, query the table for all the columns.

```
SELECT * FROM users_by_name;
```

- In the results from this query we see all four rows, so, we have successfully avoided the upsert.



Q & A