HBASE DATA MODEL:  COLUMN-ORIENTED STORAGE

HBase uses column-oriented storage and denormalized data model.

Let’s say we have an application that manages notifications to the users of a social network.

Now there can be different kind of notifications like friend request notifications, comments, likes etc.

Consider a table that stores notification data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **TYPE** | **FOR USER** | **FROM USER** | **TIMESTAMP** |
| 1 | FRIEND REQUEST STATUS | SANTOSH | SAVITHA | 143610501 |
| 2 | COMMENT | SAMYUKTHA | SANTOSH | 143610602 |
| 3 | COMMENT | JOHN | TOM | 143610703 |
| 4 | LIKE | TOM | SANTOSH | 143610804 |

This is how data is stored in traditional databases. A table with a fixed schema is defined.

To insert data into the table or to retrieve data from the table you have to be aware of the schema of the table.

You need to be aware of the column names of the table to insert or retrieve data. Each row represents a single data point.

In a column-oriented database each cell represents a single data point. When you insert data into hbase you insert one cell at a time.

When you retrieve data, you retrieve one cell data at a time. In Hbase it is more appropriate to say that data is stored in a map.

The keys for the map are **<RowId, ColumnId>** and the value is the column value. If we try to visualize this this would look like this:

|  |  |  |
| --- | --- | --- |
| **ROW** | **COLUMN** | **VALUE** |
| 1 | TYPE | FRIEND REQUEST STATUS |
| 1 | FOR USER | SANTOSH |
| 1 | FROM USER | SAVITHA |
| 1 | TIMESTAMP | 143610501 |
| 2 | TYPE | COMMENT |
| 2 | FOR USER | SAMYUKTHA |
| 2 | FROM USER | SANTOSH |
| 2 | TIMESTAMP | 143610602 |
| 3 | TYPE | COMMENT |
| 3 | FOR USER | JOHN |
| 3 | FROM USER | TOM |
| 3 | TIMESTAMP | 143610703 |
| 4 | TYPE | LIKE |
| 4 | FOR USER | TOM |
| 4 | FROM USER | SANTOSH |
| 4 | TIMESTAMP | 143610804 |

HBase tables are in fact sorted map. Row id and column id are sorted, and the sorted nature make performing look ups faster.

Advantages of column-oriented storage:

1.Let’s say we have some notifications that have special attributes.

Example: With the friend request notification we may want to store the information about the friend who made the request.

Whether they are old or new user of the social network. How many mutual friends they have?

Comments and likes would have link or photo.

Each different set of notification has a different set of attributes.

In order to accommodate these additional attributes on a traditional database you would end up adding one column for every different attribute of every notification type.

Example:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ID** | **TYPE** | **FOR USER** | **FROM USER** | **TIMESTAMP** | **FRIEND TYPE** | **COMMENTED ON** |
| 1 | FRIEND REQUEST STATUS | SANTOSH | SAVITHA | 143610501 | NEW | - |
| 2 | COMMENT | SAMYUKTHA | SANTOSH | 143610602 | - | LINK |
| 3 | COMMENT | JOHN | TOM | 143610703 | - | PHOTO |
| 4 | LIKE | TOM | SANTOSH | 143610804 | - | - |

And the column would be filled only if the notification is of a particular type.

Like the friend type column would be filled only for friend request notification.

The “Commented on” column would be only filled for Comment type notification.

This results in tables that are very sparse. Sparse tables are not efficient in traditional database. You end up using disk space even for empty cells.

In a column-oriented store the cells which are empty can be skipped completely.

So, in the example above for notification id 1 you can skip "Commented On" column. Similarly, for notification id 2 you can skip the "Friend Type" column.

The map would look like this:

|  |  |  |
| --- | --- | --- |
| **ROW** | **COLUMN** | **VALUE** |
| 1 | TYPE | FRIEND REQUEST STATUS |
| 1 | FOR USER | SANTOSH |
| 1 | FROM USER | SAVITHA |
| 1 | TIMESTAMP | 143610501 |
| 1 | FRIEND TYPE | NEW |
| 2 | TYPE | COMMENT |
| 2 | FOR USER | SAMYUKTHA |
| 2 | FROM USER | SANTOSH |
| 2 | TIMESTAMP | 143610602 |
| 2 | COMMENTED ON | LINK |
| 3 | TYPE | COMMENT |
| 3 | FOR USER | JOHN |
| 3 | FROM USER | TOM |
| 3 | TIMESTAMP | 143610703 |
| 3 | COMMENTED ON | PHOTO |
| 4 | TYPE | LIKE |
| 4 | FOR USER | TOM |
| 4 | FROM USER | SANTOSH |
| 4 | TIMESTAMP | 143610804 |

Advantages with this kind of model:

1. You can store really sparse tables very efficiently.

2. You can accommodate dynamically changing attributes. Each row id can have its own set of column ids which is independent of each other.

This make database design very flexible.

Let’s take another example. Let’s say we have an EMPLOYEES database where we want to capture Employee Name, Address and Subordinates.

In a traditional RDBMS data model, we would model them as 3 separate tables:

1. One with employee data

2. One for address

3. One for mapping subordinates of employees

As shown below:

|  |  |  |
| --- | --- | --- |
| **EMPID** | **Name** | **AddressId** |
| 1 | Santosh | 1 |

|  |  |  |
| --- | --- | --- |
| **AddressId** | **Street** | **City** |
| 1 | Yelahanka | Bangalore |

|  |  |
| --- | --- |
| **EMPID** | **SUBORDINATEEMPID** |
| 1 | 2 |
| 1 | 5 |
| 1 | 9 |

This is a normalized model and this kind of design minimizes redundant storage of data. Normalization optimizes for storage.

In a Distributed system storage is cheap. Instead you need to minimize the number of disks seeks that you perform.

If you store data across different tables then you have to do disk seeks for each of those tables.

Instead if we can embed all 3 tables into a single table you can retrieve information about a single employee in one disk seek itself.

As shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **EmpID** | **EmpName** | **Address** | **Subordinates** |
| 1 | Santosh | <STRUCT> | <ARRAY> |

A Struct represents address:

"Street": "Yelahanka",

"City": "Bangalore"

Array represents Subordinates:

("Savitha","Samyuktha")

So, this is a denormalized design. However, this supports only Create, Read, Update and Delete operation. It does not support JOIN or GROUP BY.

HBASE DATA MODEL:  COLUMN-ORIENTED STORAGE

HBase uses column-oriented storage and denormalized data model.

Let’s say we have an application that manages notifications to the users of a social network.

Now there can be different kind of notifications like friend request notifications, comments, likes etc.

Consider a table that stores notification data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **TYPE** | **FOR USER** | **FROM USER** | **TIMESTAMP** |
| 1 | FRIEND REQUEST STATUS | SANTOSH | SAVITHA | 143610501 |
| 2 | COMMENT | SAMYUKTHA | SANTOSH | 143610602 |
| 3 | COMMENT | JOHN | TOM | 143610703 |
| 4 | LIKE | TOM | SANTOSH | 143610804 |

This is how data is stored in traditional databases. A table with a fixed schema is defined.

To insert data into the table or to retrieve data from the table you have to be aware of the schema of the table.

You need to be aware of the column names of the table to insert or retrieve data. Each row represents a single data point.

In a column-oriented database each cell represents a single data point. When you insert data into hbase you insert one cell at a time.

When you retrieve data, you retrieve one cell data at a time. In Hbase it is more appropriate to say that data is stored in a map.

The keys for the map are **<RowId, ColumnId>** and the value is the column value. If we try to visualize this this would look like this:

|  |  |  |
| --- | --- | --- |
| **ROW** | **COLUMN** | **VALUE** |
| 1 | TYPE | FRIEND REQUEST STATUS |
| 1 | FOR USER | SANTOSH |
| 1 | FROM USER | SAVITHA |
| 1 | TIMESTAMP | 143610501 |
| 2 | TYPE | COMMENT |
| 2 | FOR USER | SAMYUKTHA |
| 2 | FROM USER | SANTOSH |
| 2 | TIMESTAMP | 143610602 |
| 3 | TYPE | COMMENT |
| 3 | FOR USER | JOHN |
| 3 | FROM USER | TOM |
| 3 | TIMESTAMP | 143610703 |
| 4 | TYPE | LIKE |
| 4 | FOR USER | TOM |
| 4 | FROM USER | SANTOSH |
| 4 | TIMESTAMP | 143610804 |

HBase tables are in fact sorted map. Row id and column id are sorted, and the sorted nature make performing look ups faster.

Advantages of column-oriented storage:

1.Let’s say we have some notifications that have special attributes.

Example: With the friend request notification we may want to store the information about the friend who made the request.

Whether they are old or new user of the social network. How many mutual friends they have?

Comments and likes would have link or photo.

Each different set of notification has a different set of attributes.

In order to accommodate these additional attributes on a traditional database you would end up adding one column for every different attribute of every notification type.

Example:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ID** | **TYPE** | **FOR USER** | **FROM USER** | **TIMESTAMP** | **FRIEND TYPE** | **COMMENTED ON** |
| 1 | FRIEND REQUEST STATUS | SANTOSH | SAVITHA | 143610501 | NEW | - |
| 2 | COMMENT | SAMYUKTHA | SANTOSH | 143610602 | - | LINK |
| 3 | COMMENT | JOHN | TOM | 143610703 | - | PHOTO |
| 4 | LIKE | TOM | SANTOSH | 143610804 | - | - |

And the column would be filled only if the notification is of a particular type.

Like the friend type column would be filled only for friend request notification.

The “Commented on” column would be only filled for Comment type notification.

This results in tables that are very sparse. Sparse tables are not efficient in traditional database. You end up using disk space even for empty cells.

In a column-oriented store the cells which are empty can be skipped completely.

So, in the example above for notification id 1 you can skip "Commented On" column. Similarly, for notification id 2 you can skip the "Friend Type" column.

The map would look like this:

|  |  |  |
| --- | --- | --- |
| **ROW** | **COLUMN** | **VALUE** |
| 1 | TYPE | FRIEND REQUEST STATUS |
| 1 | FOR USER | SANTOSH |
| 1 | FROM USER | SAVITHA |
| 1 | TIMESTAMP | 143610501 |
| 1 | FRIEND TYPE | NEW |
| 2 | TYPE | COMMENT |
| 2 | FOR USER | SAMYUKTHA |
| 2 | FROM USER | SANTOSH |
| 2 | TIMESTAMP | 143610602 |
| 2 | COMMENTED ON | LINK |
| 3 | TYPE | COMMENT |
| 3 | FOR USER | JOHN |
| 3 | FROM USER | TOM |
| 3 | TIMESTAMP | 143610703 |
| 3 | COMMENTED ON | PHOTO |
| 4 | TYPE | LIKE |
| 4 | FOR USER | TOM |
| 4 | FROM USER | SANTOSH |
| 4 | TIMESTAMP | 143610804 |

Advantages with this kind of model:

1. You can store really sparse tables very efficiently.

2. You can accommodate dynamically changing attributes. Each row id can have its own set of column ids which is independent of each other.

This make database design very flexible.

Let’s take another example. Let’s say we have an EMPLOYEES database where we want to capture Employee Name, Address and Subordinates.

In a traditional RDBMS data model, we would model them as 3 separate tables:

1. One with employee data

2. One for address

3. One for mapping subordinates of employees

As shown below:

|  |  |  |
| --- | --- | --- |
| **EMPID** | **Name** | **AddressId** |
| 1 | Santosh | 1 |

|  |  |  |
| --- | --- | --- |
| **AddressId** | **Street** | **City** |
| 1 | Yelahanka | Bangalore |

|  |  |
| --- | --- |
| **EMPID** | **SUBORDINATEEMPID** |
| 1 | 2 |
| 1 | 5 |
| 1 | 9 |

This is a normalized model and this kind of design minimizes redundant storage of data. Normalization optimizes for storage.

In a Distributed system storage is cheap. Instead you need to minimize the number of disks seeks that you perform.

If you store data across different tables then you have to do disk seeks for each of those tables.

Instead if we can embed all 3 tables into a single table you can retrieve information about a single employee in one disk seek itself.

As shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **EmpID** | **EmpName** | **Address** | **Subordinates** |
| 1 | Santosh | <STRUCT> | <ARRAY> |

A Struct represents address:

"Street": "Yelahanka",

"City": "Bangalore"

Array represents Subordinates:

("Savitha","Samyuktha")

So, this is a denormalized design. However, this supports only Create, Read, Update and Delete operation. It does not support JOIN or GROUP BY.