Row vs Column Oriented Databases

There are two ways to organize relational databases:

- **Row oriented**
- **Column oriented** (also known as **columnar** or **C-store**)

Row oriented databases are databases that organize data by record, keeping all of the data associated with a record next to each other in memory. Row oriented databases are the traditional way of organizing data and still provide some key benefits for storing data quickly. They are optimized for reading and writing rows efficiently.

Common row oriented databases:

- Postgres
- MySQL

Column oriented databases are databases that organize data by field, keeping all of the data associated with a field next to each other in memory. Columnar databases have grown in popularity and provide performance advantages to querying data. They are optimized for reading and computing on columns efficiently.

Common column oriented databases:

- Redshift
- BigQuery
- Snowflake

Row Oriented Databases

Traditional Database Management Systems were created to store data. They are optimized to read and write a single row of data which lead to a series of design choices including having a row store architecture.

In a row store, or row oriented database, the data is stored row by row, such that the first column of a row will be next to the last column of the previous row.

For instance, let's take this Facebook_Friends data:

Facebook_Friends

Name	City	Age
Matt	Los Angeles	27
Dave	San Francisco	30
Tim	Oakland	33

This data would be stored on a disk in a row oriented database in order row by row like this:

Select avg(age)

From friends

Matt	Los Angeles	27	Dave	San Francisco	30	Tim	Oakland	33

This allows the database write a row quickly because, all that needs to be done to write to it is to tack on another row to the end of the data.

Writing to Row Store Databases

Let's use the data stored in a database:

Matt	Los Angeles	27	Dave	San Francisco	30	Tim	Oakland	33

If we want to add a new record:

Jen	Vancouver	30

We can just append it to the end of the current data:

Matt	Los Angeles	27	Dave	San Francisco	30	Tim	Oakland	33	Jen	Vancouver	30

Row oriented databases are still commonly used for Online Transactional Processing (OLTP) style applications since they can manage writes to the database well. However, another use case for databases is to analyze the data within them. These Online Analytical Processing (OLAP) use cases need a database that can support ad hoc querying of the data. This is where row oriented databases are slower than C-store databases.

Reading from Row Store Databases

Row oriented databases are fast at retrieving a row or a set of rows but when performing an aggregation it brings extra data (columns) into memory which is slower than only selecting the columns that you are performing the aggregation on. In addition the number of disks the row oriented database might need to access is usually larger.

Extra data into Memory

Say we want to get the sum of ages from the Facebook_Friends data. To do this we will need to load all nine of these pieces of data into memory to then pull out the relevant data to do the aggregation.

Matt	Los Angeles	27	Dave	San Francisco	30	Tim	Oakland	33

This is wasted computing time.

Number of Disks accessed

Let’s assume a Disk can only hold enough bytes of data for three columns to be stored on each disk. In a row oriented database the table above would be stored as:

Disk 1		
Name	City	Age
Matt	Los Angeles	27

Disk 2		
Name	City	Age
Dave	San Francisco	30

Disk 3		
Name	City	Age
Tim	Oakland	33

To get the sum of all the people's ages the computer would need to look through all three disks and across all three columns in each disk in order to make this query.

So we can see that while adding data to a row oriented database is quick and easy, getting data out of it can require extra memory to be used and multiple disks to be accessed.

Column Oriented Databases

Data Warehouses were created in order to support analyzing data. These types of databases are read optimized.

In a C-Store, columnar, or Column-oriented database, the data is stored such that each row of a column will be next to other rows from that same column.

Let's look at the same data set again and see how it would be stored in a column oriented database.

Facebook_Friends

Name	City	Age
Matt	Los Angeles	27
Dave	San Francisco	30
Tim	Oakland	33

A table is stored one column at a time in order row by row:

Matt	Dave	Tim	Los Angeles	San Francisco	Oakland	27	30	33

Writing to a Column Store Databases

If we want to add a new record:

Jen	Vancouver	30

We have to navigate around the data to plug each column in to where it should be.

Select avg(age)

From emp;

Matt	Dave	Tim	Jen	Los Angeles	San Francisco	Oakland	Vancouver	27	30	33	30

If the data was stored on a single disk it would have the same extra memory problem as a row oriented database, since it would need to bring everything into memory. However, column oriented databases will have significant benefits when stored on separate disks.

If we placed the table above into the similarly restricted three columns of data disk they would be stored like this:

Disk 1		
Name		
Matt	Dave	Tim

Disk 2		
City		
Los Angeles	San Francisco	Oakland

Disk 3		
Age		
27	30	33

Reading from a Column store Database

To get the sum of the ages the computer only needs to go to one disk (Disk 3) and sum all the values inside of it. No extra memory needs to be pulled in, and it accesses a minimal number of disks.

While this is a slight over simplification, it illustrates that by organizing data by column the number of disks that will need to be visited will be reduced and the amount of extra data that has to be held in memory is minimized. This greatly increases the overall speed of the computation.

There are other ways in which a column oriented database can get more performance.

Coding the data into more compact forms

Let’s first examine an encoding technique that can be used by row or column oriented databases. The example of one of the columns being for states of the United States will show dictionary and bitmap encodings.

- There are 50 so we could encode the whole database with 6 bits since this would provide us 64 unique patterns.
- To store the actual abbreviations would require 16 bits since this would provide us with 256 unique patterns for each of the two ASCII characters.
- Worst of all if we stored the full name the lengths would be variable and the amount of bits needed would be a lot more.

Now let's take a look at Run-length encoding. This allows you to replace any sequence of the same value with a count and value indicator. For instance we can replace aaaab with 4a1b. This becomes even more powerful when you create projections with columns that are sorted since all values that are the same are next to each other.

Compressing the data

If each piece of data is the same number of bits long then all of the data can be further compressed to be the number of pieces of data times that number of bits for a single piece of data.

Ordering the data

When doing ad hoc queries there are a number of different sort orders of the data that would improve performance. For instance, we might want data listed by date, both ascending and descending. We might be looking for a lot of data on a single customer so ordering by customer could improve performance.

In Row oriented databases, indexes can be created but data is rarely stored in multiple sort orders. However, in Column oriented databases you can have the data stored in an arbitrary number of ways. In fact, there are benefits beyond query performance. These different sorts ordered columns are referred to as projections and they allow the system to be more fault tolerant, since the data is stored multiple times.

Original Database (WS)

Disk 1		
Name		
Matt	Dave	Tim

Disk 2		
City		
Los Angeles	San Francisco	Oakland

Disk 3		
Age		
27	30	33

Name Ordered ASC (RS)

Disk 1		
Name		
Dave	Matt	Tim

Disk 2		
City		
San Francisco	Los Angeles	Oakland

Disk 3		
Age		
30	27	33

Name Ordered DESC (RS)

Disk 1		
Name		
Tim	Matt	Dave

Disk 2		
City		
Oakland	Los Angeles	San Francisco

Disk 3		
Age		
33	27	30

This seems like a complicated set of tables to update, and it is. This is why the architecture of a C-store database has a writeable store (WS) and a read optimized store (RS). The writeable store has the data sorted in the order it was added, in order to make adding data into it easier. We can easily append the relevant fields to our database as seen below:

Original Database (WS)

Disk 1			
Name			
Matt	Dave	Tim	Evan

Disk 2			
City			
Los Angeles	San Francisco	Oakland	Blacksburg

Disk 3			
Age			
27	30	33	20

Then the read-optimized store can have multiple projections. It then has a tuple mover which manages the relevant updates from the WS to the RS. It has to navigate the multiple projections and insert the data in the proper places.

Name Ordered ASC (RS)

Disk 1			
Name			
Dave	Evan	Matt	Tim

Disk 2			
City			
San Francisco	Blacksburg	Los Angeles	Oakland

Disk 3			
Age			
30	20	27	33

Name Ordered DESC (RS)

Disk 1			
Name			
Tim	Matt	Evan	Dave

Disk 2			
City			
Oakland	Los Angeles	Blacksburg	San Francisco

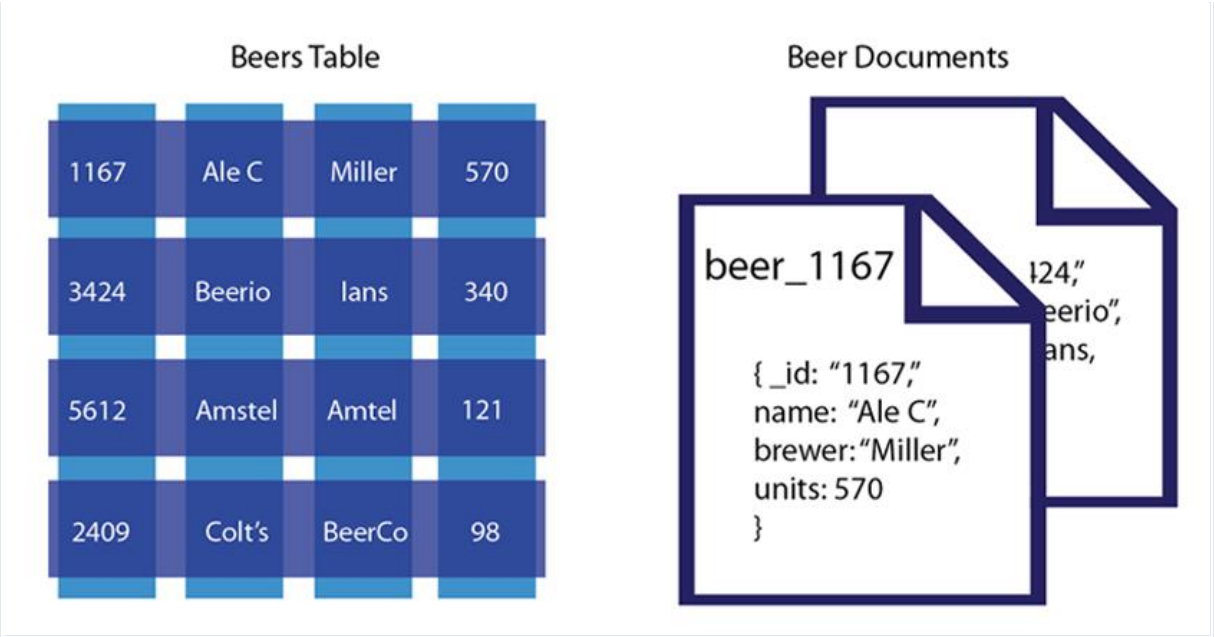
Disk 3			
Age			
33	27	20	30

This architecture means that while the data is being updated from the WS to the RS the partially added data must be ignored by queries to the RS until the update is complete.

Comparing Document Databases and Relational Databases

In a relational database system you must define a **schema** before adding records to a database. The schema is the structure described in a formal language supported by the database and provides a blueprint for the tables in a database and the relationships between tables of data. Within a table, you need to define constraints in terms of rows and named columns as well as the type of data that can be stored in each column.

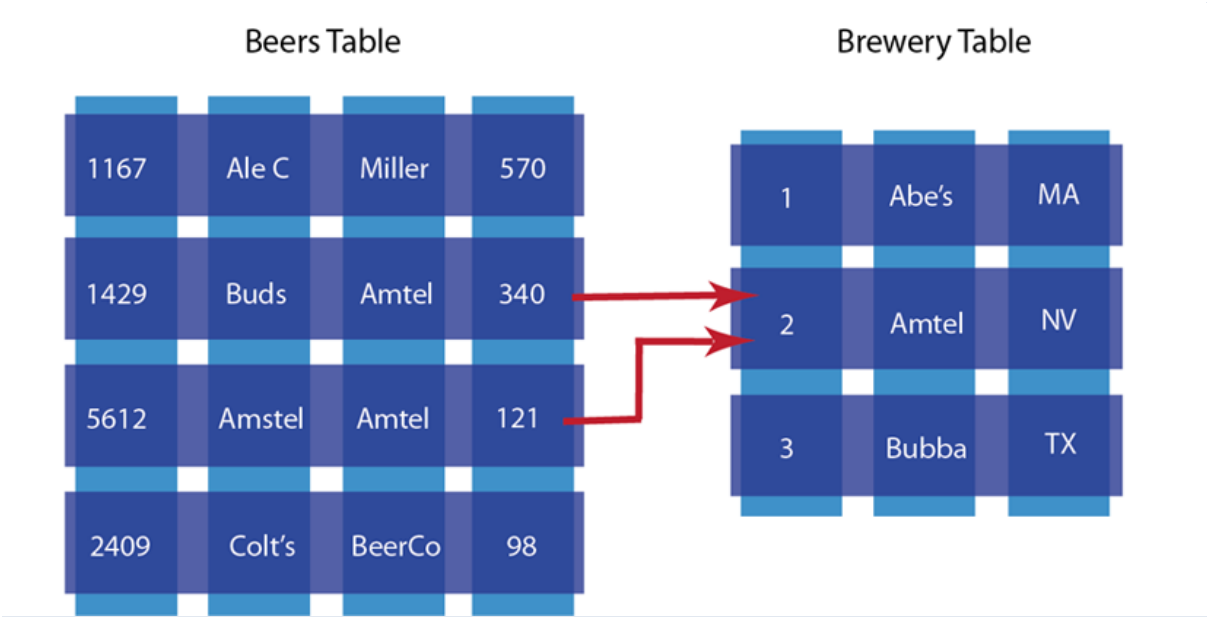
In contrast, a document-oriented database contains **documents**, which are records that describe the data in the document, as well as the actual data. Documents can be as complex as you choose; you can use nested data to provide additional sub-categories of information about your object. You can also use one or more document to represent a real-world object. The following compares a conventional table with document-based objects:



In this example we have a table that represents beers and their respective attributes: id, beer name, brewer, bottles available and so forth. As we see in this illustration, the relational model conforms to a schema with a specified number of fields which represent a specific purpose and data type. The equivalent document-based model has an individual document per beer; each document contains the same types of information for a specific beer.

In a document-oriented model, data objects are stored as documents; each document stores your data and enables you to update the data or delete it. Instead of columns with names and data types, we describe the data in the document, and provide the value for that description. If we wanted to add attributes to a beer in a relational model, we would need to modify the database schema to include the additional columns and their data types. In the case of document-based data, we would add additional key-value pairs into our documents to represent the new fields.

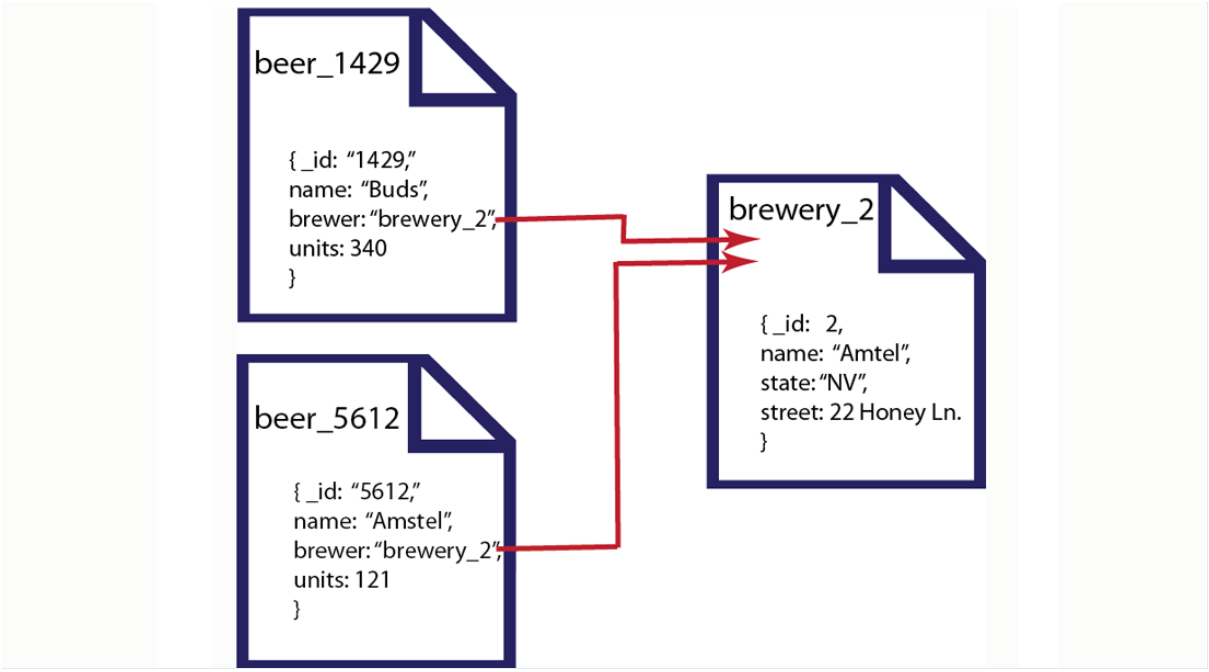
The other characteristic of relational database is data normalization — this means you decompose data into smaller, related tables. The figure below illustrates this:



In the relational model, data is shared across multiple tables. The advantage to this model is that there is less duplicated data in the database. If we did not separate beers and brewers into different tables and had one beer table instead, we would have repeated information about breweries for each beer produced by that brewer.

The problem with this approach is that when you change information across tables, you need to lock those tables simultaneously to ensure information changes across the table consistently. Because you also spread information across a rigid structure, it makes it more difficult to change the structure during production, and it is also difficult to distribute the data across multiple servers.

In the document-oriented database, we could choose to have two different document structures: one for beers, and one for breweries. Instead of splitting your application objects into tables and rows, you would turn them into documents. By providing a reference in the beer document to a brewery document, you create a relationship between the two entities:



In this example we have two different beers from the Amstel brewery. We represent each beer as a separate document and reference the brewery in the brewer field. The document-oriented approach provides several upsides compared to the traditional RDBMS model. First, because information is stored in documents, updating a schema is a matter of updating the documents for that type of object. This can be done with no system downtime. Secondly, we can distribute the information across multiple servers with greater ease. Since records are contained within entire documents, it makes it easier to move, or replicate an entire object to another server.