

## EIGHT WEEKS SUMMER TRAINING REPORT

### ON

**MODERN BIG DATA ANALYSIS WITH SQL**

**SPECILIZATION:**

Under the Guidance of Mr. **Lan** **Cook**

**School of Computer Science & Engineering Lovely Professional University, Phagwara From (June - July,2021)**

### Submitted By

**Name of student: Sandeep Kumar Registration Number: 11911267 Signature of the student:A picture containing text

Description automatically generated**

**DECLARATION:**

I hereby declare that I have completed my eight weeks summer training at Modern Big Data Analysis with SQL Specialization (Coursera) platform from April 30,2020 to June 25,2020 under the guidance of MR. **Lan** **Cook**. I have worked with full dedication during their 8 weeks of training and my learning outcomes fulfill the requirements of training for the award of degree of B.Tech. CSE, Lovely Professional University, Phagwara.

Sandeep Kumar ,11911267 Signature of student:

A picture containing text

Description automatically generated

Date – 25 Sept. 2021

**Table of contents**

|  |  |
| --- | --- |
| **Content** | **Page No.** |
| ACKNOWLEDGMENT, Chapter 1--INTRODUCTION | 5-7 |
| **Chapter 2 Foundation of Big Data Analysis-**  WHAT IS DATA, WHY ORGANIZE DATA  WHAT DOES A DBMS DO | 8-9 |
| RELATIONAL DATABASE AND SQL  SPECIAL TABLE TYPES  DATA TYPES | 9-12 |
| RELATIONAL DATABASE DESIGN.  FOREIGN KEYS  DATABASE NORMALIZATION | 13-14 |
| BIG DATA  DISTRIBUTED STORAGE  BIG DATA SYSTEM AND HOW THEY DIFFER FROM TRADITIONAL SYSTEM  BIG DATABASE BIG DATA STORES, AND SQL | 14-16 |
| **Chapter 3-- ANALYSING BIG DATA WITH SQL**  NEED OF SQL  SQL COMMANDS | 16-27 |
| **Chapter 4- Managing Big Data in Clustures And cloud Storages--**  BROWSING TABLES IN MEGASTORE  BROWSING TABLE WITH HUE  CREATING A NEW DATABASE  USING THE TABLE BROWSER  USING THE DATA SOURCE PANEL  CREATING A NEW, EMPTY TABLE  CREATING A NEW TABLE TO QUERY EXISTING DATA | 27-31 |
| CREATING DATABASES AND TABLES WITH SQL  CREATING A DATABASE  CREATING A TABLE  IMPALA METADATA REFRESH | 32-33 |
| DATA TYPE  INTEGER DATA TYPES  DECIMAL DATA TYPES  CHARACTER STRING DATA TYPES  OTHER DATA TYPES | 34-37 |
| EXAMINING DATA TYPES  EXAMINE THE TABLE SCHEMA  THE TYPE OF  FUNCTION IN IMPALA  OUT-OF-RANGE VALUES TEXT FILES  AVRO FILES PARQUET FILES  APACHE AVRO APACHE PARQUET | 37-44 |
| **Chapter 5---CONCLUSION**  LEARNING OUTCOMES | 45 |

**ACKNOWLEDGMENT**

I would like to express my gratitude towards my university as well as Coursera for providing me the golden opportunity to do this wonderful summer training regarding Modern Big Data Analysis with SQL Specialization, which also helped me in doing a lot of homework and learning. As a result, I came to know about so many new things. So, I am really thanking full to them.

Moreover, I would like to thank my friends who helped me a lot whenever I got stuck in some problem related to my course. I am thankful to have such a good support of them as they always have my back whenever I need.

Also, I would like to mention the support system and consideration of my parents who have always been there in my life to make me choose right thing and oppose the wrong. Without them I could never had learned and became a person who I am now.

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my sincere thanks to all of them.

**Modern Big Analysis with SQL Specialization:**



**Text

Description automatically generated**

**Text, letter

Description automatically generated**

### 

Chapter 1- **INTRODUCTION:**

**This Specialization teaches the essential skills for working with large-scale data using SQL.**

Maybe you are new to SQL, and you want to learn the basics. Or maybe you already have some experience using SQL to query smaller-scale data with relational databases. Either way, if you are interested in gaining the skills necessary to query big data with **modern distributed SQL engines**, this Specialization is for you.

Most courses that teach SQL focus on traditional relational databases, but today, more and more of the data that’s being generated is too big to be stored there, and it’s growing too quickly to be efficiently stored in commercial data warehouses. Instead, it’s increasingly stored in **distributed clusters** and **cloud storage**. These data stores are cost-efficient and infinitely scalable.

To query these huge datasets in clusters and cloud storage, you need a newer breed of SQL engine: distributed query engines, like **Hive**, **Impala**, **Presto**, and **Drill**. These are open-source SQL engines capable of querying enormous datasets. This Specialization focuses on Hive and Impala, the most widely deployed of these query engines.



**chC**

**Chapter 2 --Foundation of big data analysis with SQL**

**What is Data:**

In this specialization, data means digital data. Information that can be transmitted,

stored, and processed using modern digital technologies, like the Internet,

disk drives, and modern computer. Imagine that you walk by a movie theater and see a movie poster about a new movie you want to see. You enjoy the image and you can read to

find out about the actors in the movie. And the show times at the local theater. So you get information that is interesting and useful to you. But does the site of the movie poster itself comprise data? No, the poster presents an image to your eye. Which you read and interpret to get the actors' name and the show times. So an actual object or a direct sensory experience is not really data. But you can have data about those things. Data is a representation of something that captures some features and ignores others. Now data itself can be divided into two kinds, analog data and digital data. A photograph of the the movie poster taken with a film camera is not digital data. An image from photographic film results from chemical processes on photosensitive materials with essentially infinite continuous ranges of color on paper. In contrast, a digital scan of the movie poster or a photo taken with a smartphone is digital data. It can be saved, copied, sent to another smartphone, displayed on many screens, so on. A film photograph requires mechanical and chemical processes to reproduce. But a smartphone photograph is just numbers stored in the phone. These numbers can be copied to cloud storage, sent to another device, displayed on any number of screens. This may sound a bit philosophical but it's worthwhile to notice the transition from things to representations

**Why Organize Data:**

Remember the last movie you watched. You look at the image and with the superpower of your own human brain, you interpret what you see to answer some simple questions. Who are the actors in the movie?  What are the show times at this theater?  You organize the sight of the poster to get some information that interests you. If you take a photograph of the movie poster with your phone, you do have data, but the form of the data as image pixels

determines what use you can easily make of that data. think about what you need to do if I share with you my photographs of every movie poster on display in a city, and then you want to find all the movies starring one certain actor. Now, imagine a simple table or spreadsheet with each row containing the movie title, the actor names and the showtimes. This may not be as aesthetically pleasing as the pictures of posters, but it makes things much simpler if you want to find a movie with some actor or answer some simple questions about a movie.

Notice, of course, you cannot answer any questions about information that is not included in your data. For instance, your table with movie title, actor names and showtimes, cannot tell you whether there are tickets available for a certain showing of the movie.

What if instead of sharing my movie poster photos, I share all my photos including shots of restaurant menus, friends I meet, buildings I like. This may carry more information but it makes it even tougher for you to answer a simple question about what movies are showing with a certain actor. The important thing to learn here is that the organization of data has

a major impact on how easily you can use the data to answer questions. At this point, I'd like to establish a few terms. By the way, you may have colleagues who use these terms with a meaning similar to the definitions here but not identical. It's OK as long as you know that the meanings given here are generally accepted and you can clear up the details with your colleagues if any confusions arise. A data store is a collection of data of any type. This is a general term and it can be used for collections of different sizes. For instance, I could call the collection of photos on my smartphone a data store. I might have a cloud storage account with photos, videos and text messages. This is a different, larger data store, and I could upload all my photos from my phone to this larger data store. The service that maintains my cloud storage also has a much larger data store of all the documents saved by all of its users

**What Does a DBMS Do?**

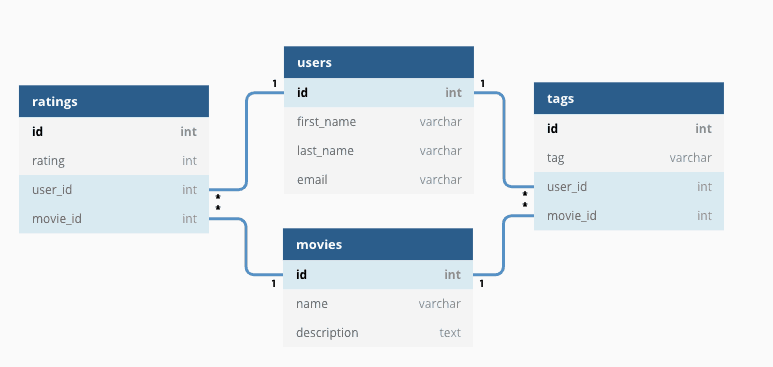
The DBMS helps you manage your data. The word "manage" can mean several things, but here I mean, the need to control access to your data. You want to set up different user accounts with different access to different parts of the data. For the restaurant database, your kitchen staff persons need to update and retrieve data about the food inventory, but not your employee pay records. Your accountant needs access to the pay records, and so on.ting .The initial electronic databases were created in the 1960s. Those were the early days of electronic computing systems for business. Using the best technologies at the time, engineers cobbled together ways to perform these four activities. And then in 1970, a man named E.F. Codd published an article that revolutionized the way we think about database systems, and that new way of thinking became common and remains incredibly useful today.

**Relational Database and SQL.**

E.F. Codd was working at an IBM research facility in San Jose, California in 1970 when he published his pioneering paper, “A Relational Model of Data for Large Shared Data Banks.” This paper appeared in the premier academic journal of computer science, the Communications of the Association for Computing Machinery, or just the CACM. Don't we love our acronyms? Codd's colleagues, Don Chamberlin and Ray Boyce, designed several computer languages expressly to implement Codd's ideas for working with data. In 1974, they settled on the language they called Structured Query Language, also called SQL, which may also be pronounced "sequel"—another acronym. Codd’s paper is the basis of relational database management systems or RDBMSs. SQL was especially made to work with RDBMSs. Nearly all of the popular database systems since the 1980s are relational systems and nearly all of those use SQL as their primary language. Some examples of SQL-based relational systems are Oracle, SQL Server,

MySQL, DB2, PostgreSQL, Microsoft Access, and SQLite.

Note, the big data era has seen the rise of other types of databases called "NoSQL" databases. That is a topic for later in this course. The great thing about SQL is that it's so simple and easy to learn. All four of the database activities from the previous video are their own simple commands in SQL. The holding areas for different kinds of data in SQL are called tables. When you design your database, you set up different tables for different kinds of data. There are simple commands or statements to create a table, change what types of records that table will hold, or discard the table from your database. The SQL commands for these are, simply enough, CREATE, ALTER, and DROP. These commands cover the activity of designing, or more accurately defining, your database, and together they form a category of SQL commands called Data Definition Language or DDL. Don't be intimidated by the word "language" in the label "Data Definition Language." It's just a category of SQL commands that performed one of the four activities I've already identified for a database management system.The next activity you want to perform is to update the data in your database. You want to add records to tables, change some of the data in those records, and remove records. The SQL commands for those actions are INSERT, UPDATE and DELETE. They form the SQL category of Data Manipulation Language or DML. Of course, once you set up your tables and then put some data into your database, you can do one of the main things people do with data: you can ask questions about it. The SELECT statement is the superstar of the SQL commands.

****

**Fig 2.1**

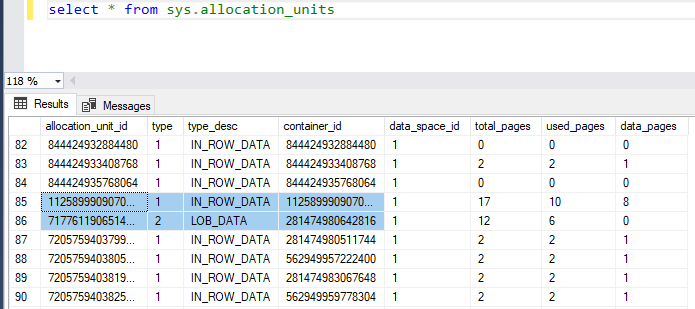
**Special Table Types.**

In addition to the basic user defined table, SQL Server provides us with the ability to work with other special types of tables. The first type is the **Temporary Table** that is stored in the tempt system database. There are two types of temporary tables: a local temporary table that has the single number sign prefix (#) and can be accessed by the current connection only, and the Global temporary table that has two number signs prefix (##) and can be accessed by any connection once created.

A **Wide Table** is a table the uses the Sparse Column to for optimized storage for the NULL values, reducing the space consumed by the table and increasing the number of columns allowed in that table to 30K columns.

**System Tables** are a special type of table in which the SQL Server Engine stores information about the SQL Server instance configurations and objects information, that can be queried using the system views.

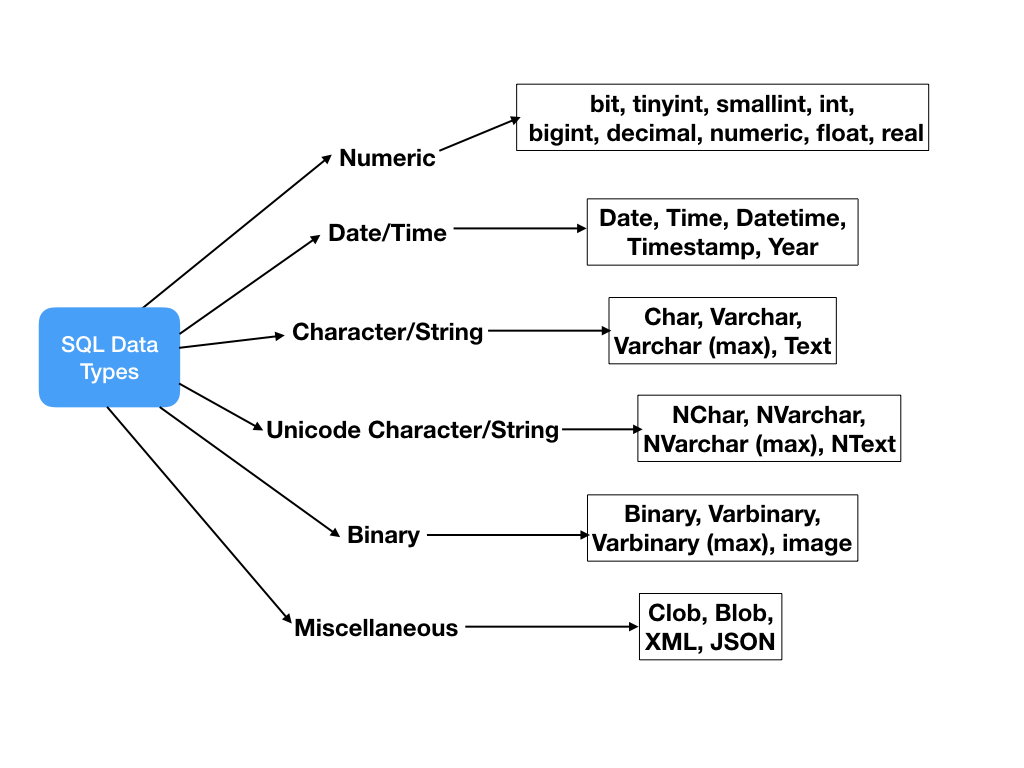
**Partitioned Tables** are tables in which the data will be divided horizontally into separate unites in the same filegroup or different filegroups, based on a specific key, to enhance the data retrieval performance.

****

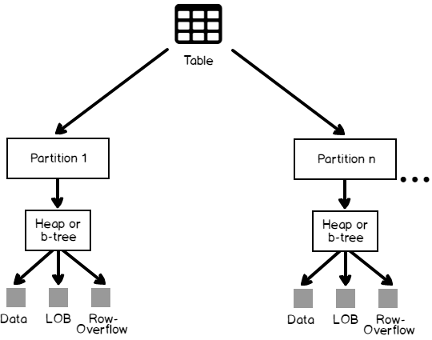
**Fig 2.2**

**Data Types.**

The data type of a column defines what value the column can hold: integer, character, money, date and time, binary, and so on. Each column in a database table is required to have a name and a data type. An SQL developer must decide what type of data that will be stored inside each column when creating a table. The data type is a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL

****

**Fig 2.3**



**Fig 2.5**

**Relational Database Design.**

**Primary keys:**

A primary key is the column or columns that contain values that uniquely identify each row in a table. A database table must have a primary key for Optima to insert, update, restore, or delete data from a database table. Optima uses primary keys that are defined to the database. However, you can also define primary keys to supplement those in the database.

A primary key is needed:

* In any table that is visited more than once in a process, for example, a child table that has two or more parent tables referenced in the Access Definition.
* To enable the Point and Shoot feature for a Start Table.

**Foreign keys**

A foreign key is a key used to link two tables together. This is sometimes also called as a referencing key.

A Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.

**Database Normalization**

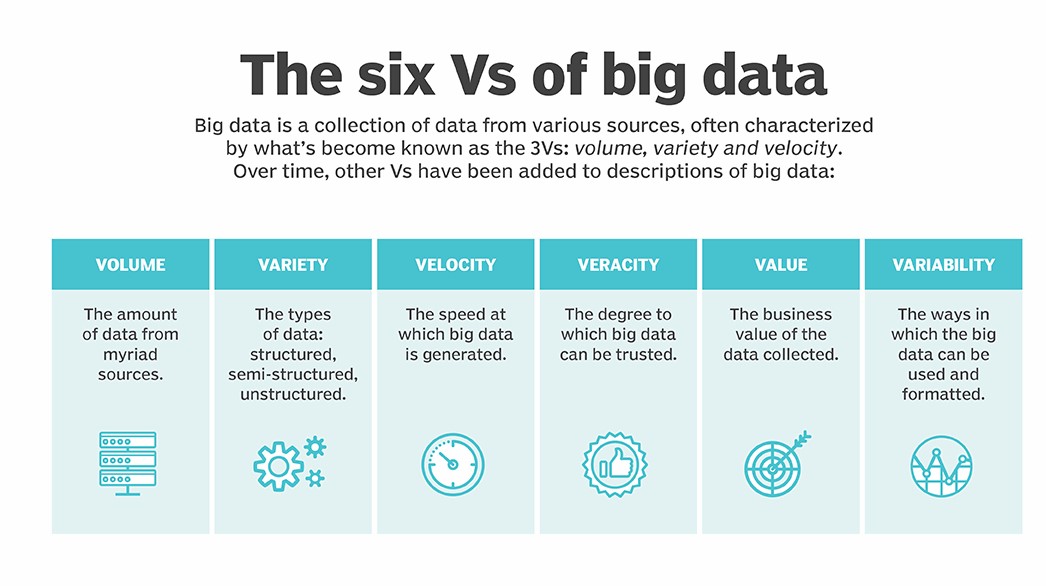
**Normalization** is a database design technique that reduces data redundancy and eliminates undesirable characteristics like Insertion, Update and Deletion Anomalies. Normalization rules divides larger tables into smaller tables and links them using relationships. The purpose of Normalization in SQL is to eliminate redundant (repetitive) data and ensure data is stored logically.

The inventor of the Relational Model Edgar Codd proposed the theory of normalization of data with the introduction of the First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form.

**Big Data**

The definition of big data is data that contains greater variety, arriving in increasing volumes and with more velocity. This is also known as the three Vs.

Put simply, big data is larger, more complex data sets, especially from new data sources. These data sets are so voluminous that traditional data processing software just can’t manage them. But these massive volumes of data can be used to address business problems you wouldn’t have been able to tackle before.

fig 2.6

**Distributed Storage**

Also known as distributed computing and distributed databases, a distributed system is a collection of independent components located on different machines that share messages with each other in order to achieve common goals.

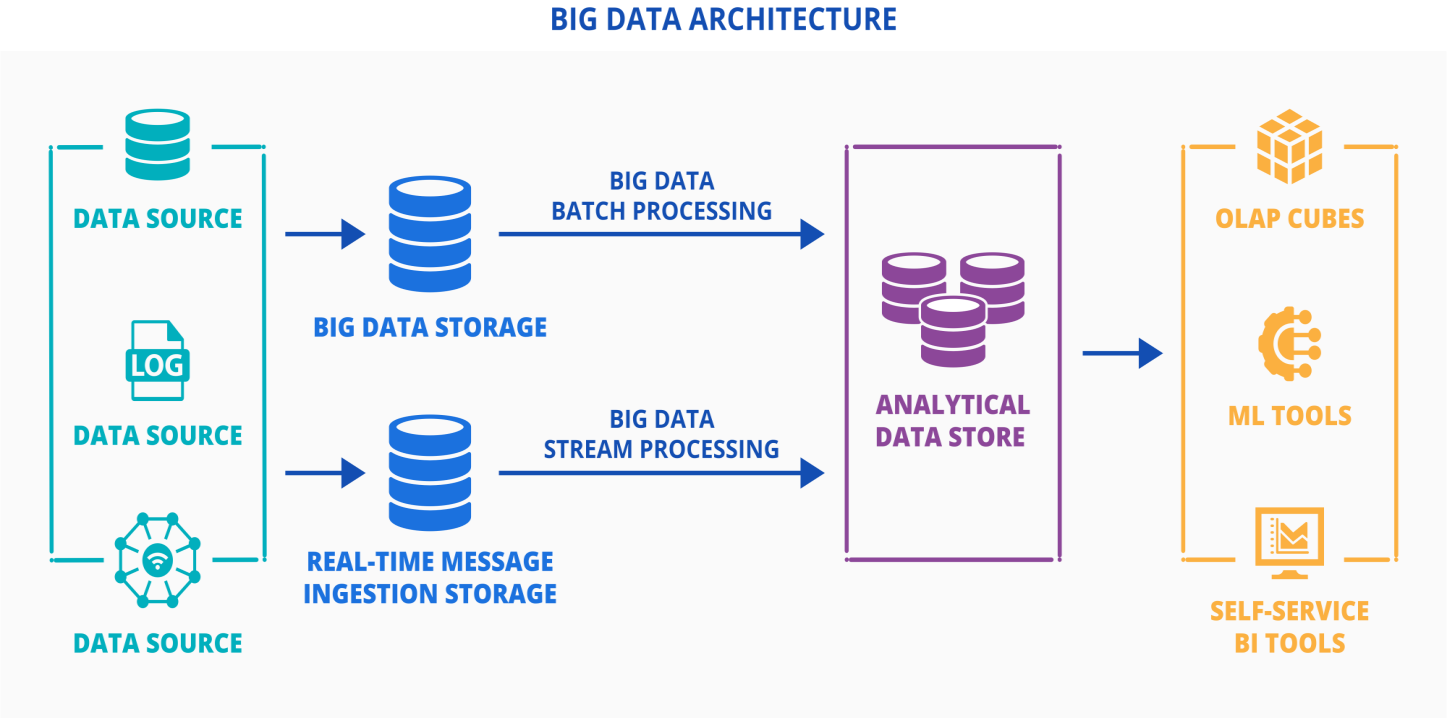
As such, the distributed system will appear as if it is one interface or computer to the end-user. The hope is that together, the system can maximize resources and information while preventing failures, as if one system fails, it won't affect the availability of the service.

**Big Data System and How they Differ from Traditional System**

The constraints you build into your data models, together with ACID-compliant transactions and stored procedures, allow you to build high-quality financial applications and other online transaction processing or OLTP systems, with your database at the center of your total application design. The strong business constraints on allowed data in your database can exert centralized control over what your business will accept as "good data," no matter what user interfaces or other programs you build out in your organization. The efficient enforcement of business rules, and the kinds of operational databases that this affords, represent one of the great successes of SQL and relational technology. Another strength of relational systems is that - just by storing your data in tables - you will automatically have structured data, not counting large text or binary fields. I'll discuss a bit later the kinds of analysis you can do with different kinds of data, but for now, I'll just say that a great many analytic techniques rely on data first being in a tidy, structured form. For a data analyst, having data in structured form to begin with is a great advantage, in that you can skip over some of the effort of preparing the data, and proceed more directly to analysis. Relational technology has become so widely adopted and mature that you can find good choices for relational database software that you can easily install and run on almost any computer. These include PostgreSQL and MySQL Community Edition, which are free database servers, and SQLite, a library that less programmers embed an ACID-compliant database in their own programs without needing a separate database server. This means that - when you stick to mostly structured data and your data size is small or medium - you can readily choose an RDBMS to handle your data storage, and the software will help to keep your data organized and useful. When your database system implements the DCL commands of GRANT and REVOKE properly, you can use these to manage security on your data:

**Big Database Big Data stores, and SQL**

Among the most successful big data systems, are big analytic systems or data warehouses. These includes Apache Impala, Apache Hive, Apache Drill and Presto. You will become skilled with Hive and Impala in the subsequent courses of this specialization. Hive and Impala each implement their own dialects of SQL. Hive QL for Hive, and Impala SQL for Impala. These dialects are similar to one another, and their uses in analytic systems emphasize a subset of SQL most suitable for data warehouses. Other large analytic database systems include Oracle data warehousing and Teradata. But these systems tend to have a much higher cost per terabyte of data than the others.



**Fig 2.7**

**Chapter 3-- Analysing big data with SQL**

**What is SQL?**

SQL (Structured Query Language) is used to communicate with a database. It is the standard language for relational database management systems. SQL statements are used to perform tasks such as update data on a database or retrieve data from a database. Some common relational database management systems that use SQL are Oracle, Sybase, Microsoft SQL Server, Access, Ingres, etc. SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database.

  **Fig 3.1**

**Need of SQL:**

SQL is an essential skill for data analysts, and data scientists. It's the standard language for accessing tabular data. Whether we are working with data in a relational database, in a data warehouse, or in a set of files on a cluster or in Cloud storage, in all these cases we can use SQL to retrieve and analyse the data. Also data scientists, data engineers and Machine Learning developers who use other languages like Python and R, often also use SQL to connect their code to a wide variety of different data sources.

**SQL Commands**

Utility Statements

These are used to see what databases exist, switch into a particular database, see what tables are in it, and look at the columns in those tables.

* Command: “ SHOW DATABASES “

Description: lists the databases on the MySQL server

* Command: “USE ‘Database name “

Description: changes the current database

* Command: “SHOW TABLES “

Description: shows the list of tables in the current database

* Command: “DESCRIBE ‘Table name’ “

Description: shows what columns are in a table

‘SELECT’ Statement

The Select statement is the most important part of the SQL language. It is used to query the database and retrieve selected data that follow the conditions we want. The data that's returned by a SQL statement is called the result set or just the result. The options with a Select statement are so extensive that select forms its own category of SQL statements called queries.

Example: “SELECT \* from student table”

Here ‘\*’ means all the columns and the above command will return all the data in the table.

The building blocks of a SELECT statement are called clauses. A SELECT statement is made up of one or more clauses. Following are different clauses:

* ‘FROM’ Clause:

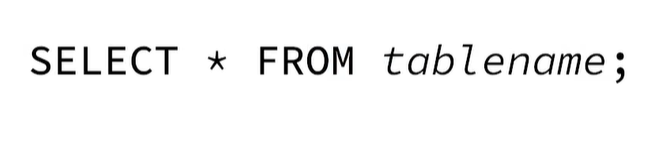
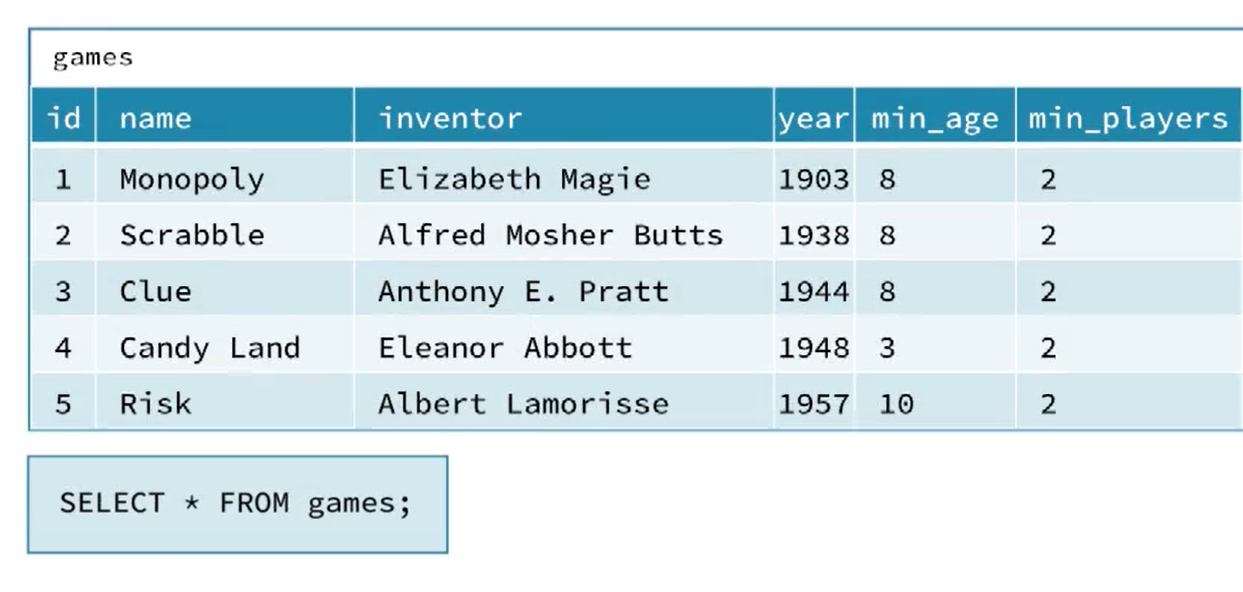


Figure 3.2

A SELECT statement begins with the keyword SELECT. The part of the statement starting at the beginning with the keywords SELECT and ending before the keyword FROM is called the SELECT clause. Everything that comes after the keyword SELECT in this clause is called the SELECT list. In a SELECT list, the asterisk symbol which is universally pronounced "star", has the special meaning all the columns. So when you use the star as your SELECT list, the results will contain all the columns from the table in the same order that they're defined in in the table. If we want to return some but not all of the columns, then instead of using the star, we use a list of the column names separated by commas.

Examples:

Fig 3.3

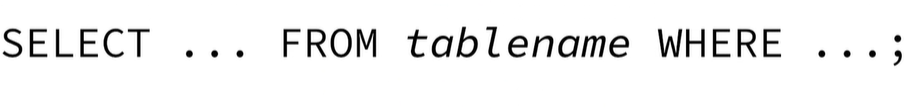
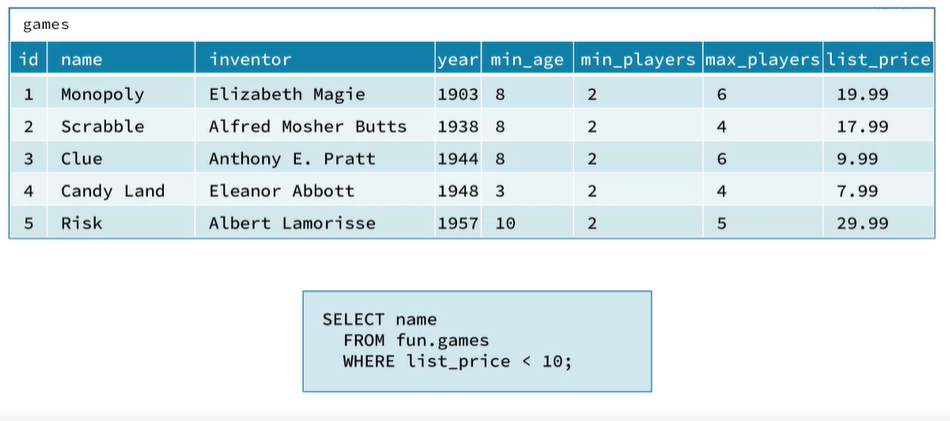
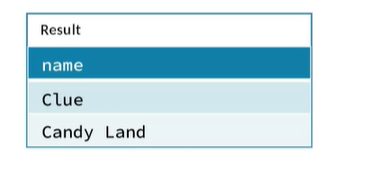
* ‘WHERE’ Clause :

Fig 3.4

The WHERE clause filters the rows of data based on one or more conditions. Typically these conditions are tests of the values in specified columns for all the rows of the data. In other words, the Where clause takes all the data in the table, tests which rows meet some criteria, and returns only those rows. The Where clause has no effect on which columns are returned only on which rows are returned. The Where clause is optional. If you run a Select statement that has a Select clause and a From clause but no Where clause, then you get a result set that has as many rows as the table specified in the From clause. The Where clause is where you really start to do data analysis, because we can use it to answer questions in the form in which rows are these conditions true. These conditions are the question you're asking about the data, and the result you get back contains the answer to that question ready for you to interpret.

Example:





* ‘GROUP BY’ Clause:

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e if a particular column has same values in different rows then it will arrange these rows in a group. GROUP BY clause splits a table into groups of rows so that the aggregates can be computed for each group. The WHERE clause filters the rows of the employees table before they're grouped and aggregated. The GROUP BY statement is often used with aggregate functions to group the result-set by one or more columns.

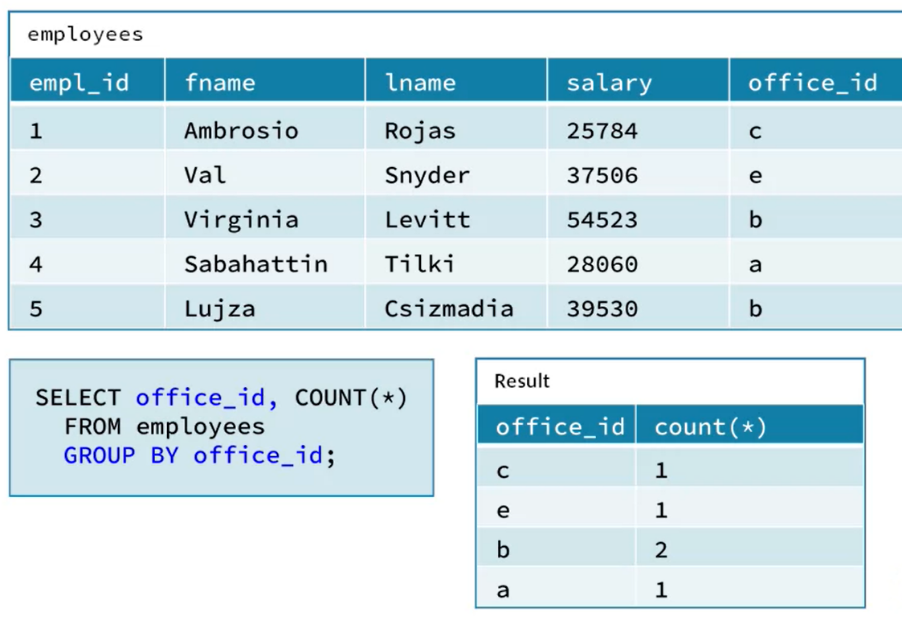
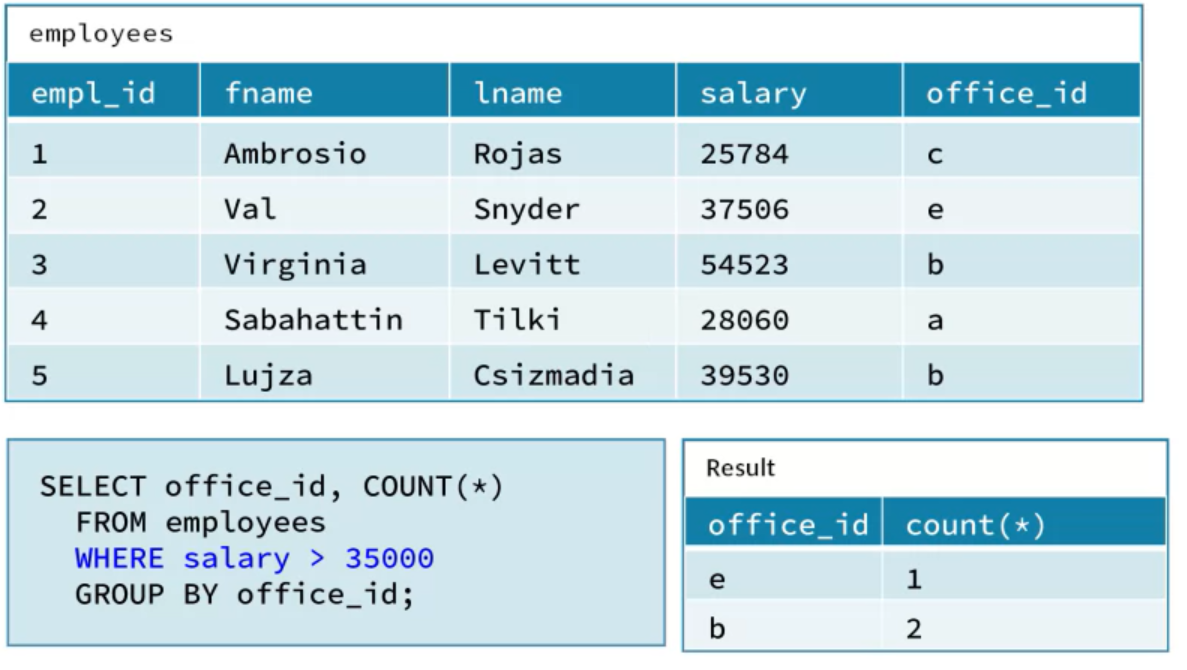
Examples:

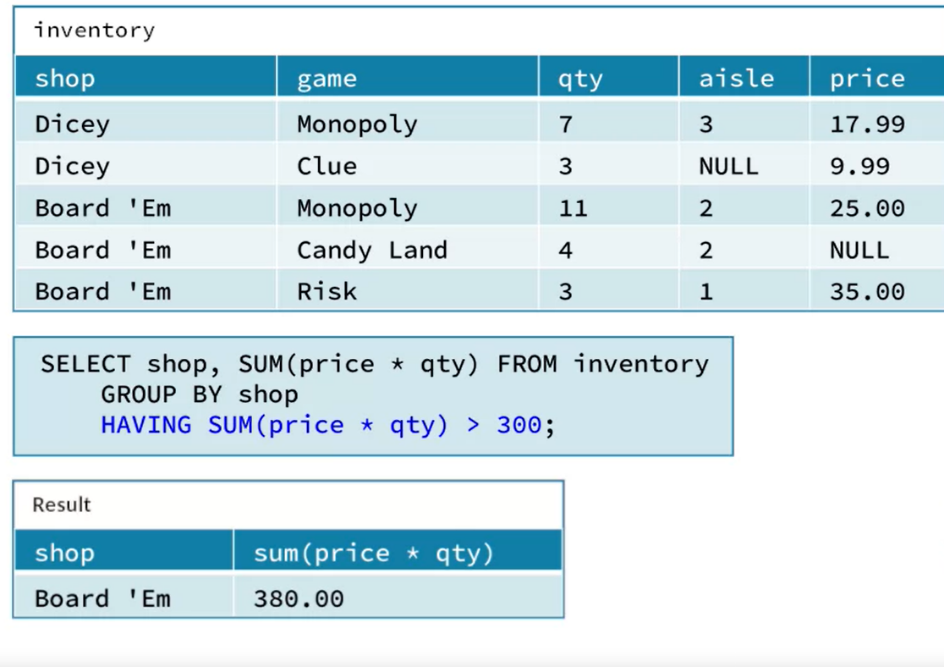
Figure 3.9



* ‘HAVING’ Clause :

Fig 3.7

The purpose of the HAVING clause is to filter groups in the data using criteria that are based on aggregates of the groups. The HAVING clause is intended to be used with the GROUP BY clause. It must come after the GROUP BY clause in a SELECT statement, and it's processed after the GROUP BY clause. So the rows that it filters represent groups. Just like in the WHERE clause, we can use logical operators like NOT to negate the truth of a Boolean expression, and AND or OR to combine multiple expressions into a single Boolean expression. The only major difference here is that you can use aggregate functions in the HAVING clause whereas you cannot use them in the WHERE clause.

Examples.

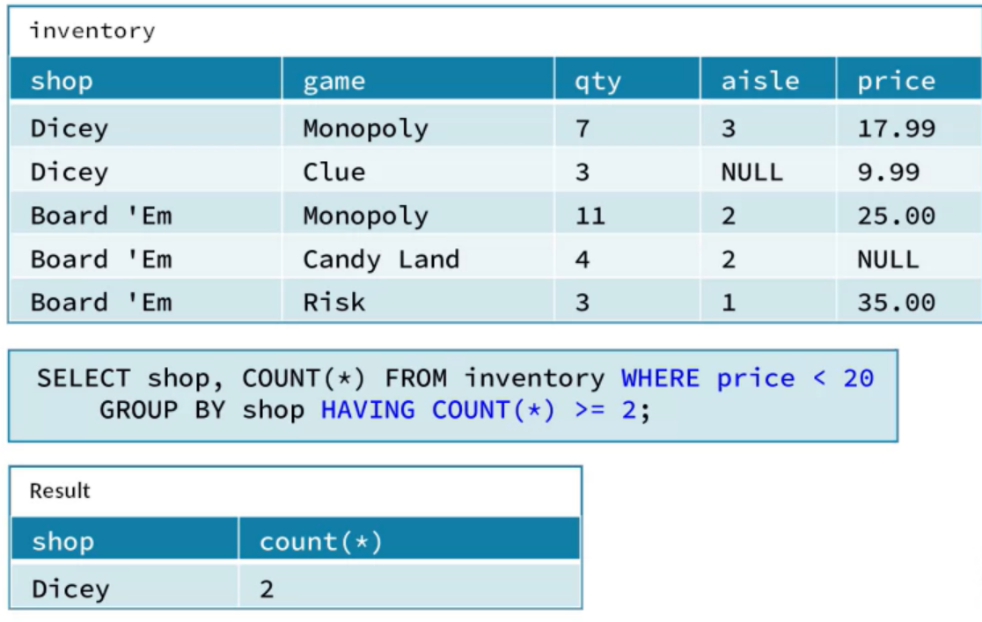


Figure 3.13

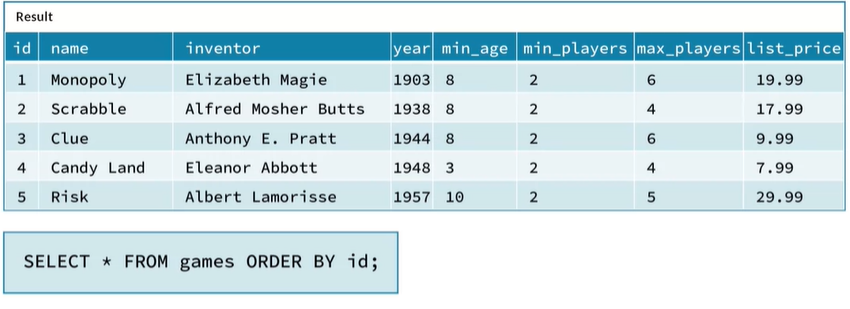
Fig 3.8

* ORDER BY Clause:

ORDER BY CLAUSE is used to show the rows of result set to be in a specific order. The order by clause takes the result from all the earlier clauses, select, from, where, and to group by, and it arranges those rows, sorts them in a specific order before returning them. To set the rows of result set to be in a specific order, the way to tell the SQL engine that is to use an order by clause. ORDER BY clause is used together in a select statement with other clauses where group by and having, but it must come after those clauses.

When ORDER BY clause is used in a query, the SQL engine returns the rows in order by the column or columns specified. The order they're returned in by default is ascending order. Ascending means from smallest to largest.

Examples:



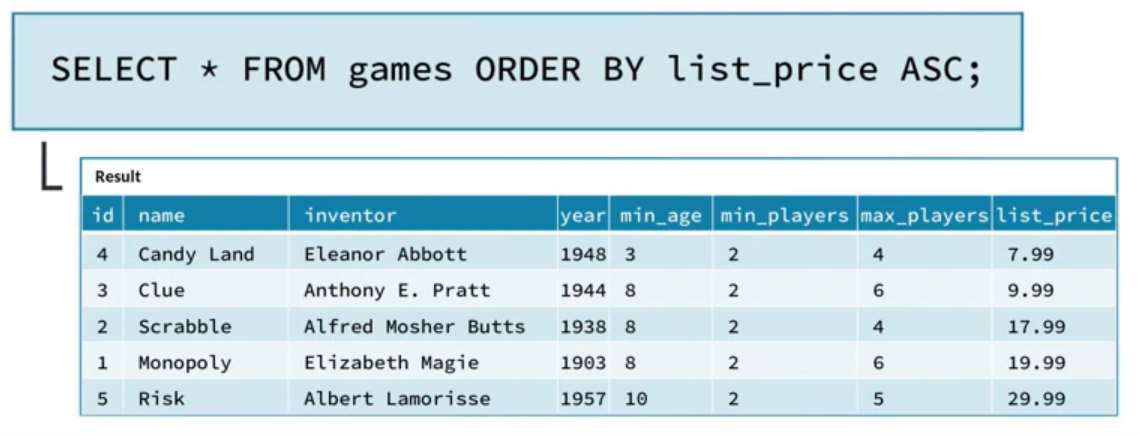
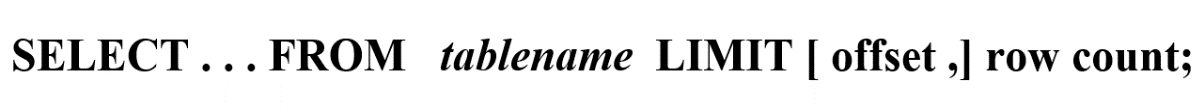


Fig 3.9

* LIMIT Clause:



The LIMIT clause is used in the [SELECT](https://www.mysqltutorial.org/mysql-select-statement-query-data.aspx) statement to constrain the number of rows to return. The LIMIT clause accepts one or two arguments. The values of both arguments must be zero or positive [integers](https://www.mysqltutorial.org/mysql-int/).

The offset specifies the offset of the first row to return. The offset of the first row is 0, not 1. The row count specifies the maximum number of rows to return.

Examples:

Figure 3.17

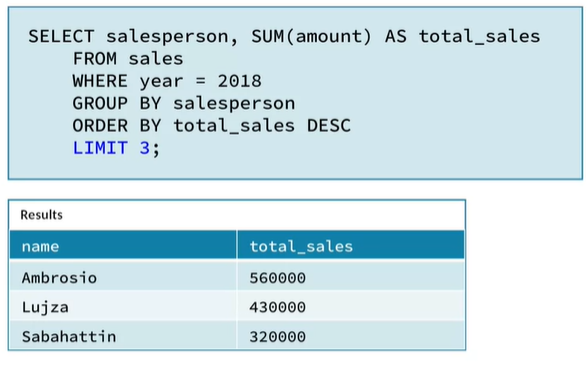


Fig 3.10

Fig 3.10

UNION and UNION ALL

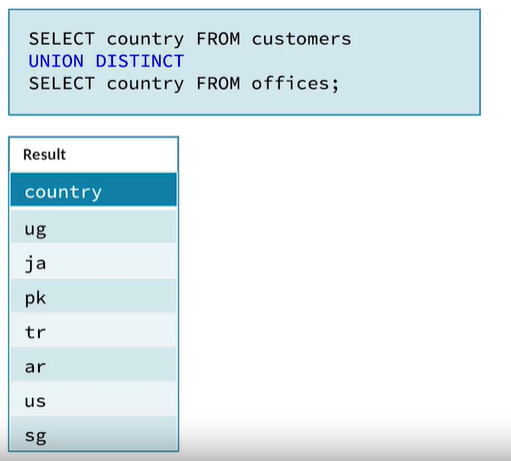
The UNION operator in SQL combines two result sets into one. It takes the rows returned by one SELECT statement and the rows returned by another SELECT statement, and it stacks them together. It combines them vertically. There are two variations of the UNION operator: UNION ALL and UNION DISTINCT

* UNION ALL

UNION ALL operator is used to combine the result sets of 2 or more SELECT statements. It does not remove duplicate rows between the various SELECT statements (all rows are returned).

Each SELECT statement within the UNION ALL must have the same number of fields in the result sets with similar data types.

Table

Description automatically generated

* UNION DISTICT

The UNION DISTINCT operator combines the result sets of two or more input queries. Union DISTINCT will remove duplicate records,

so, every row in the result set will be unique.

JOINS

A join in SQL combines data from two related tables into one result set. In the simplest sense, a join takes columns from one table and columns from another table, and it merges them together, it combines them horizontally. But a join does not to just throw together the columns from the two tables; it also matches the rows from the two tables.

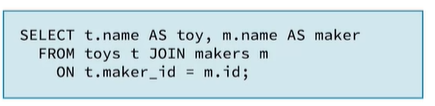
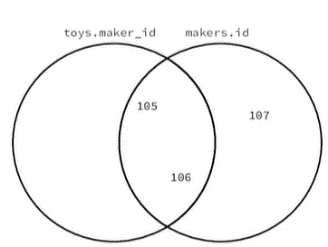
When we write a query in SQL that joins two tables, we specify what the relationship between these two tables is, and the SQL engine uses that relationship to match the rows.

The reason that joins are so important and so widely used in SQL is that related data are often stored in separate tables.

Different types of Joins are:

* Inner Join

The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e., value of the common field will be same.



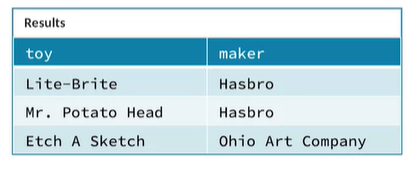


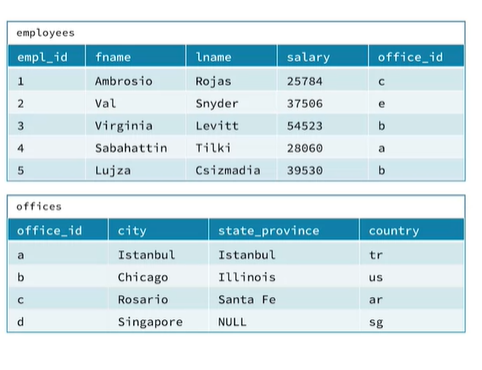
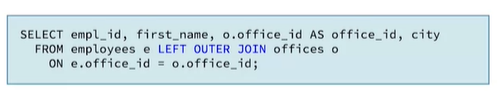
Figure 3.24

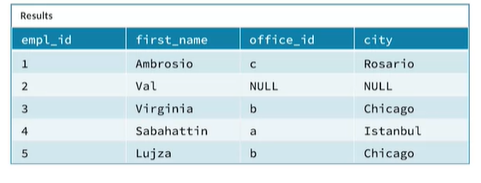
* Outer Join

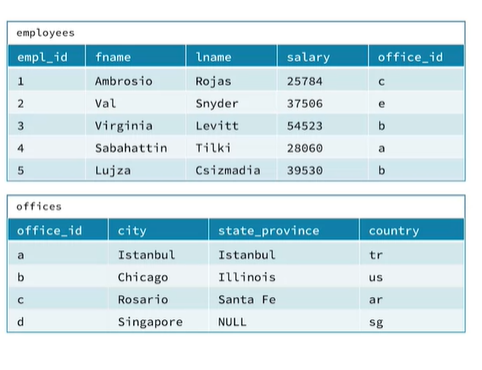
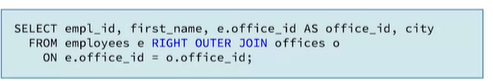
The Outer Join includes the matching rows as well as some of the non-matching rows between the two tables. An Outer join basically differs from the Inner join in how it handles the false match condition.

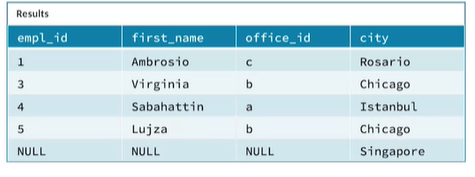
There are 3 types of Outer Join:

* Left Outer Join: Returns all the rows from the LEFT table and matching records between both the tables.





Right Outer Join: Returns all the rows from the RIGHT table and matching records between both the tables.



* Full Outer Join: It combines the result of the Left Outer Join and Right Outer Join.

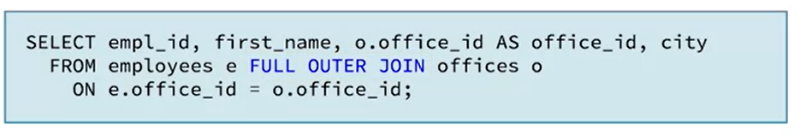
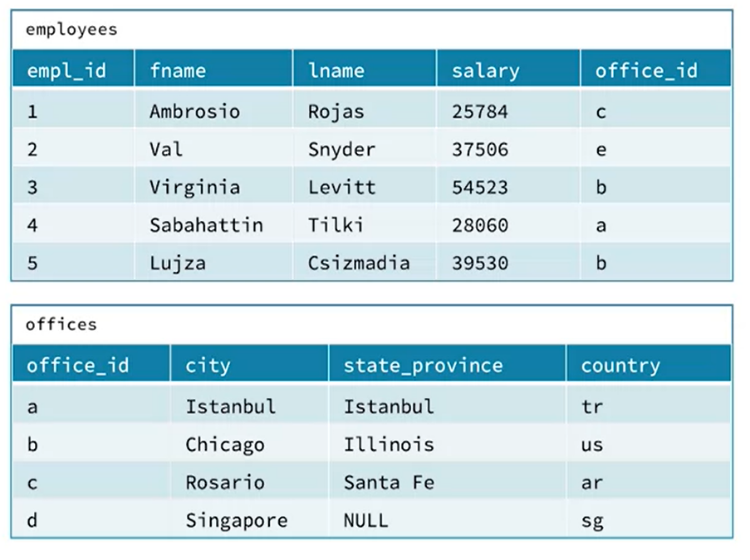
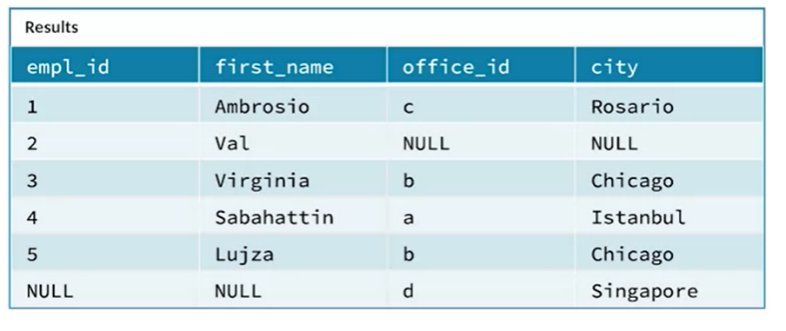
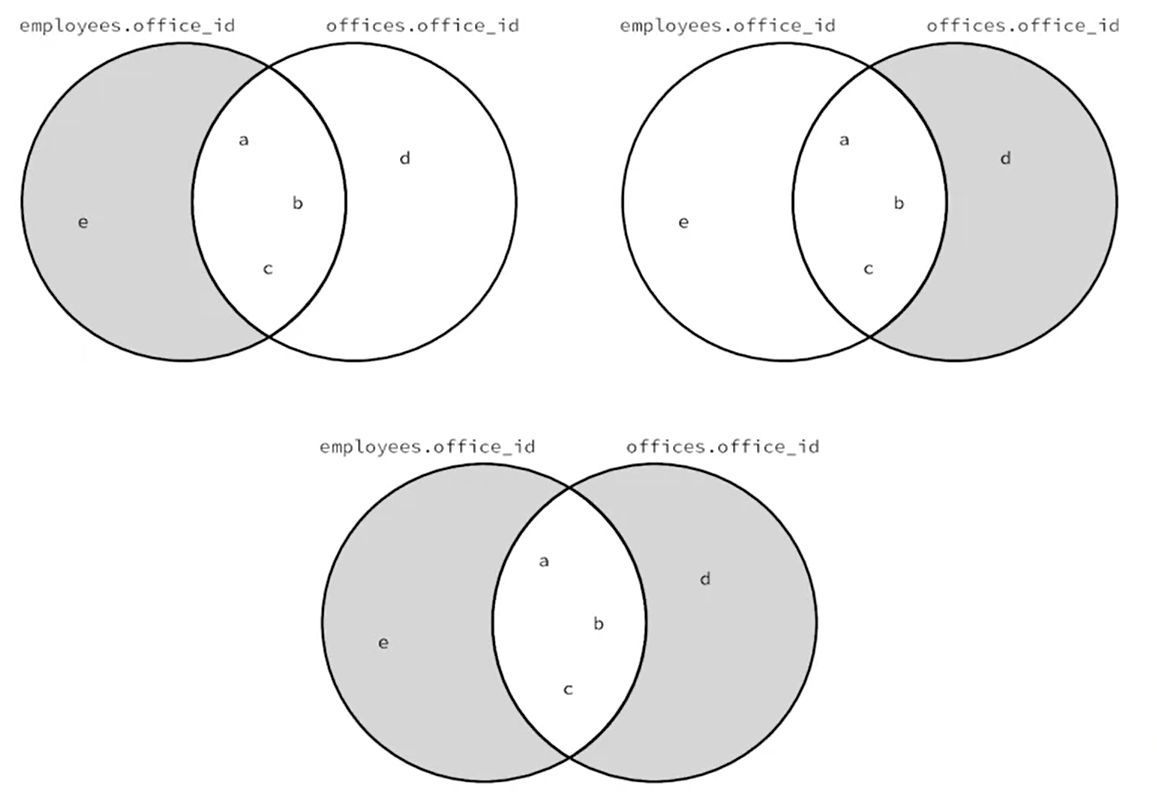


Figure 3.32





**v**

**Chapter 4-- Managing Big data in clustures and Cloud storages**

**Browsing Tables In Megastore:**

**Browsing Table with hue**Hue is a web browser-based analytics workbench that provides a user interface to Hive, Impala, and other tools.  It's widely used in the real world, and it's installed on the VM. Hue includes a number of different interfaces. There are just a few interfaces

Click the Menu icon in the upper left corner, then under browsers click Tables. Here in the table browser, you can click Databases to see what databases exist. You can see there are databases named default, fly, fun, toy, and wax. You can click the name of a database to see what tables are in it. for example we are clicking fun. You can see that the fun database has tables named card rank, card suit, games, and inventory. Then you can click on the name of the table to see more details about that table. If you go to the Columns tab, you can see that this games table has 8 columns, id, name, inventor, year, mean age, mean players, max players, and least price.You can also click the sample tab to see a sample of the data in this table.

# Creating a New Database

You can start the process for creating a database by going directly into the Table Browser, or by using the data source panel.

When you create a database using these methods, Hive or Impala creates the directory **/user/hive/warehouse/databasename.db**(where databasename is the name you entered) unless a different location is specified. Tables created in a database will be placed as a subdirectory within this database directory.

## Using the Table Browser

Enter the Table Browser by clicking the hamburger menu (three horizontal lines) and choosing **Browsers > Tables**. The main (center) panel will show the **default** database for Hive and Impala. To add a new database:

1. Click **Databases**in the breadcrumbs at the top. (See Figure 1 below)
2. Hover over the **+**symbol on the far right; it should say **Create a new database**. Click that symbol.
3. Give your database a name such as **test**, and (optionally) a description in the appropriate field
4. Optional: If you want to store the database files in a location other than HDFS (in S3, for example), uncheck the **Default Location**box and supply the location.
5. Click **Submit**.
6. The Task History pop-up will appear; the top should say **Creating database name**with a green line underneath. (See Figure 2 below.) Click the **x**on the right to dismiss the window.
7. Verify the creation was successful by clicking **Databases**in the breadcrumbs, and note that your new database now appears in the list. If you like, you can check the HDFS file structure to see that **/user/hive/warehouse/name.db**exists.
8. To drop (delete) your test database, check the box next to it and click the **Drop**button above the list, then click **Yes**. This will drop the database and delete the directory from HDFS. Be careful not to drop any databases that have tables in them!



**Figure 4.1**

IMG_257

**Figure 4. 2**

## Using the Data Source Panel

The data source panel appears with all areas of Hue, so you can easily use this method without the need to switch to the Table Browser first.

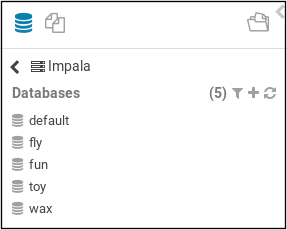
1. Be sure the data source panel is in the database mode (and not in the files mode). The database icon (stacked disks) should be blue. (See Figure 3 below.)
2. Navigate to the Impala source. (See Figure 4 below. You can also use Hive, and the database will be available for both engines.)
3. Hover over the **+**symbol; it should say **Create database**. Click that symbol. (Note: From this point on, the steps are the same as in the previous section.)
4. In the main panel, give your database a name such as **test**, and (optionally) a description in the appropriate fields.
5. Optional: If you want to store the database files in a location other than HDFS (in S3, for example), uncheck the **Default Location**box and supply the location.
6. Click **Submit**. As in the previous section, the Task History pop-up will appear. Click the **x**to dismiss it.

7.You'll now be in the Table Browser, but in the **default**database. Verify the creation was successful by clicking **Databases** in the breadcrumbs, and note that your new database now appears in the list. You can also navigate in the data source panel back to the Impala source and see the list of databases there.

8.To drop your test database, check the box next to it and click the **Drop**button above the list, then click **Yes**.

IMG_258

**Figure 4.3**



**Figure 4.4**

# Creating a New, Empty Table

When you create a table, Hive or Impala creates the directory **/user/hive/warehouse/tablename**or, if it's in a database other than **default**,**/user/hive/warehouse/databasename.db/tablename**. This directory will be empty at first; you will learn in a later week of the course how to populate the table with data by loading data files into it.

You will learn how to drop these tables in a later lesson in this Week, so don't worry about dropping the examples you create now.

As with the database creation, you can start this process with the data source panel (Option A, below), or you can go directly to the Table Browser (Option B). Decide which you want to try from your experiences creating test databases:

Option A: On the data source panel on the left, navigate to the database where you want to put the table. In this case, choose the **default** database in the Impala source. Hover over the **+**symbol; it should say **Create table**. (See Figure 5.) Click that symbol.



**Figure 4.5**

Option B: Enter the Table Browser by clicking the hamburger menu (three horizontal lines) and choosing **Browsers > Tables**. The main (center) panel will show the **default** database. (You can switch to a different database if you like, but this isn't necessary for testing purposes. Stay in the **default**database.) Again, hover over the **+**symbol on the far right; it should say **Create a new table**. Click that symbol.

In the main panel, you have the opportunity to indicate a file in HDFS that will be the data for this table. This can be very handy, because Hue will try to guess the columns, including their data types, based on this file. You will see this later in this course, so for now, you'll create an empty table with no data.

Click the **Type**field and change it to **Manually**; then click the **Next**button

1. In the **DESTINATION** area, name your table. For this example, use **default.test**, or just **test** if you are in the **default**database.
2. For **PROPERTIES**, you can set the file format and storage location, if desired. For now, leave the defaults (**Text**format and **Store in Default location**checked).
3. Under **FIELDS**, you can specify the columns for your table; click **+Add Field** for each column and specify the name and data type. Give your test table a couple of columns, such as **id**and **title**. You can leave the field type as **string** for each. You'll learn more about choosing data types later in this course.
4. Click **Submit**. The Task History pop-up will appear; the top should say **Creating table name**(for the test case, **name** is **default.test**) with a green line underneath. Click the **x**on the right to dismiss the window.
5. Verify the creation was successful by looking at the data source panel. Navigate to the database (Impala **default**) if necessary, and note that your new table now appears in the list. You might need to click the refresh button (two curved arrows) to refresh the display; choose **Clear cache**if you do this.
6. For your test, check the HDFS file structure to see that **/user/hive/warehouse/test/** exists. This directory will be empty, because the table has no data in it.

# Creating a New Table to Query Existing Data

The steps above described how to create an empty table, with no data in it. You can also use Hue to create a table to query data that already exists in HDFS.

For example, in the HDFS directory **/old/castles/**, there is a file named **castles.csv**. Review this file: The fields are separated by commas, the names of the columns (**name**and **country**) are provided in the header line, and both columns contain character string data. Using the steps below, you can create a table to query this data.

1. Using the data source panel (in the database mode) or the table browser, click the +symbol to create a new table.
2. Click the **Type**field and change it to **Manually**; then click the **Next**button.
3. In the **DESTINATION**area, name your table. For your example table, use **default. Castles**.
4. For **PROPERTIES**, you can set the file format. The example file is a text file, so leave the default format (**Text**) checked.
5. Uncheck the **Store in Default location**checkbox. When you do this, you will see an **External location**field appear below it.
6. Enter the HDFS directory path (**/old/castles**for the example) into the **External location**field. Alternatively, you can click the **..**icon on the right side and use the dialog to select this folder. (See Figure 6 below.)
7. Directly below there, click the **Extras**icon, which looks like three sliders. You can add an optional description of the data location in the **Description**field (leave it blank for now). You can also specify the character that separates the fields here by checking the **Custom char delimiters**checkbox. Check the box and notice that three drop-down menus appear. You can specify different kinds of delimiters here. For this example, you only need the **Field** drop-down menu. The field separator is already set to **Comma (,)**which is the field separator (delimiter) used in the file **castles.csv**, so keep this selection.
8. Under **FIELDS**, specify the columns for this table; click **+Add Field**for each column and specify the name and data type. Recall that the file **castles.csv**, which contains the data for the example table, has two columns: **name**and **country**. Add both (in that order), both as **string** type
9. Click **Submit**. The Task History pop-up will appear; the top should say **Creating table dbname.tablename** with a green line underneath. Click the **x**on the right to dismiss the window.
10. Verify the creation was successful by looking at the data source panel. Navigate to the database (Impala **default**) if necessary, and note that the new table (**castles**) now appears in the list. You might need to click the refresh button (two curved arrows) to refresh the display; choose **Clear cache**if you do this.
11. Hover the cursor over the table name then click the **i** icon to the right of the table name. Click the **Sample** tab to verify that the data appears in the sample results.



**Figure 4.6**

# Creating Databases and Tables with SQL

In the previous reading, you learned how to use the Table Browser in Hue to create databases and tables through point-and-click actions. This method is sometimes convenient, but it’s often better to use SQL commands to create databases and tables.The SQL statements **CREATE DATABASE**and **CREATE TABLE**provide a more systematic way to create databases and tables. These commands give you greater power and flexibility, and they can be used in other Hive and Impala interfaces besides Hue. Using these commands, you also have the option to script and automate the creation of databases and tables, making these tasks more reproducible. As you go through this reading, or after you've read through it once, use the VM to create some test databases and tables. You can drop the databases and tables when you're done.

# Creating a Database

Creating the database is actually as simple as step 2 below. Step 1 is just to get you to the place where you can run that step, and step 3 is verification of success.

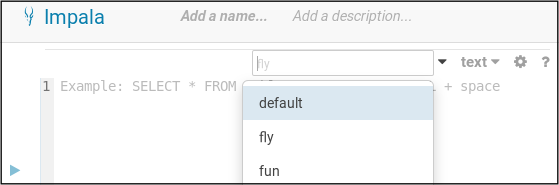
1. First, get to the Impala query editor in the VM by clicking the **Query**button. Since Hive and Impala share the Hive metastore, either will work, but using the Impala query editor is easier. (If you use Hive, Impala will not immediately acknowledge the new database. You will learn more about this later in this Week's lessons. For now, use Impala rather than Hive.)
2. Enter and execute the command: **CREATE DATABASE test;**Note: This will create a directory in HDFS in the default location: **/user/hive/warehouse/test.db**.
3. Verify the creation was successful using the data source panel to the left of the query editor panel. Look in the Impala or Hive databases; you should see six databases
4. (**default**, **fly**, **fun**, **test**, **toy**, and **wax**). If **test**does not appear, try refreshing the display using the refresh button (two curved arrows) at the top of this left panel. If you like, you can check the HDFS file structure to see that **/user/hive/warehouse/test.db** exists.
5. To drop your test database, enter and execute the command: **DROP DATABASE test;**

# Creating a Table

This is just a quick introduction to the **CREATE TABLE**statement, giving the most basic structure. The next lesson will break down the **CREATE TABLE**statement and cover each of the clauses in it. Don't worry about the details yet.

Step 3 below is the actual creation step, which will be the same for any tool you use to enter SQL commands. Steps 1 and 2 are to help you navigate to the appropriate area in Hue, and step 4 is verification of success.

1. Click the **Query**button to access the Impala query editor.
2. Check the active database and be sure it's the one you want for your new table. For testing, put it in the **default**database; if necessary, select **default**as the active database. (See Figure 1 below.)
3. Enter and execute this command: **CREATE TABLE test (col1 INT, col2 STRING);**The table name comes after **CREATE TABLE**, and then a list of the column names with their data types. The command below creates a table named **test**with two columns, an integer column named **col1**and a string column named **col2**. (See Notes below.)
4. Verify the creation was successful using the data source panel to the left of the query editor panel. If the selected database is not Impala's **default**database, navigate to it and check that your table **test**is listed. You might need to refresh the display by clicking the refresh button (the two curved arrows). If you like, you can check the HDFS file structure to see that **/user/hive/warehouse/test** exists.
5. To drop the table, enter and execute the command: **DROP TABLE test;**The table directory and any data it might have held will also be deleted. After dropping the table, verify that  **/user/hive/warehouse/test** no longer exists.
6. **CREATE TABLE test (col1 INT, col2 STRING);**



**Figure 4.7**

# Impala Metadata Refresh

While you can usually use either Hive or Impala to create end-query tables, there's an important difference between how Hive and Impala access the metastore. Hive retrieves metadata from

the metastore every time it builds a query, but Impala does not. Impala caches metadata in memory to reduce query latency. This Impala cache metadata consists of the structure and locations of tables retrieved from the metastore, and also additional information about table data files retrieved from the data storage system like HTFS or S3. Impala's metadata cache helps it

return query results as quickly as possible. But the cached metadata can get out of sync with the metadata in the metastore, and with the stored data files. This happens when changes are made outside of Impala. For example, when new tables are created using hive, when table data is imported using Hughes table browser, or when table data is added using HTFS command.

When changes like this occur outside of Impala, it's necessary to refresh Impala's metadata cache. There are different ways to do this depending on what changes were made outside of the Impala. The refresh command updates the information that Impala caches about the schema of a particular table, and the locations and files for that table in the data storage system.

Use this command if you have altered a table schema such as renaming a column or added data to the table. The syntax is refresh and the table name. But if you've added a new table to the database from outside of Impala, then you'll need to use a different command to update Impala's metadata cache. The command is invalidate metadata and the table name.

**Data Type**

Every column in a table has a data type. When you create a table with Hive or Impala, you specify those data types along with the names of the columns. In this lesson, you'll learn about

the data types that Hive and Impala support. These break down into  three main categories: integer data types, which represent whole numbers, decimal data types, which can

represent numbers with fractional parts, and character string data types which represent text.

Hive and Impala also support some various other data types that do not fall into these three categories.

# Integer Data Types

Hive and Impala support four integer data types: **TINYINT**, **SMALLINT**, **INT**, and **BIGINT**. These represent whole numbers with no fractional parts.

Larger integer types allow you to represent larger ranges of numbers, as shown in the table below, but processing them requires more memory, so you should generally use the smallest integer type that accommodates the full range of values in your data.

| Integer Type | Range |
| --- | --- |
| **TINYINT** | -128 to 127 |
| **SMALLINT** | -32,768 to 32,767 |
| **INT** | -2,147,483,648 to 2,147,483,647 (approximately 2.1 billion) |
| **BIGINT** | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (approximately 9.2 quintillion) |

Hive and Impala support four integer data types: **TINYINT**, **SMALLINT**, **INT**, and **BIGINT**. These represent whole numbers with no fractional parts.

Larger integer types allow you to represent larger ranges of numbers, as shown in the table below, but processing them requires more memory, so you should generally use the smallest integer type that accommodates the full range of values in your data.

Unlike in some relational databases, all integer types in Hive and Impala are signed; that is, they can represent positive or negative integer values. There are no unsigned integer types.

# Decimal Data Types

Hive and Impala support three decimal data types: **FLOAT**, **DOUBLE**, and **DECIMAL**. These types represent numbers that can include fractional parts.

The **FLOAT**and **DOUBLE**types represent floating-point numbers, which do not have a predetermined number of digits after the decimal point. **DOUBLE**offers greater range and precision than **FLOAT**, but processing **DOUBLE**s requires more memory than processing **FLOAT**s, so in general, choose **FLOAT**unless you need the range or precision that **DOUBLE** offers.

Because of the binary system used to store numbers, both **FLOAT** and **DOUBLE**data types can produce unexpected inaccuracies, even with seemingly simple arithmetic like 0.1 + 0.2. (See [The Floating Point Guide](https://floating-point-gui.de/" \t "https://www.coursera.org/learn/cloud-storage-big-data-analysis-sql/supplement/Caw0D/_blank) for more about this.)

**DOUBLE**is more accurate because **DOUBLE**uses 64 bits to store each number, while **FLOAT**uses only 32 bits. (So **DOUBLE**has double the number of bits.) This means **FLOAT**is typically accurate up to 7 digits, while **DOUBLE**is accurate up to 15 or maybe 16 digits.

The **DECIMAL**type represents numbers with fixed precision and scale. When you create a **DECIMAL**column, you specify the precision, p, and scale, s. Precision is the total number of digits, regardless of the location of the decimal point. Scale is the number of digits after the decimal place. To represent the number 8.54 without a loss of precision, you would need a **DECIMAL**type with precision of at least 3, and scale of at least 2.

The table below illustrates the difference in precision using results for π. Note that the **DECIMAL(17,16)** type means there is a total of 17 digits, with 16 of them after the decimal point.

| Data Type | Result for π (**bold**are accurate) |
| --- | --- |
| **FLOAT** | **3.141592**7410125732 |
| **DOUBLE** | **3.141592653589793**1 |
| **DECIMAL(17,16)** | **3.1415926535897932** |

Using the **DECIMAL** type, it is possible to represent numbers with greater precision than the **FLOAT** or **DOUBLE** types can represent. The maximum allowed precision and scale of the **DECIMAL**type are both 38. (Hive can allow larger values of precision and scale, but Impala does not support them.)

The table below describes the range of **DOUBLE**, **FLOAT**, and **DECIMAL(38,0)**. The ranges described below are the largest negative and largest positive number that each data type can represent.

| Data Type | Range |
| --- | --- |
| **FLOAT** | -3.40282346638528860 \* 10^38 to 3.40282346638528860 \* 10^38 |
| **DOUBLE** | -1.79769313486231570 x 10^308 to 1.79769313486231570 x 10^308 |
| **DECIMAL(38,0)** | -10^38 + 1 to 10^38 - 1 |

For representing currency, you should use **DECIMAL**instead of **FLOAT**or **DOUBLE**; this prevents loss of precision, which is typically of paramount importance with financial data. Another choice for currency is to use an integer type to represent, for example, the number of cents, instead of storing dollars with fractional parts.

# Character String Data Types

Hive and Impala support three character data types: **STRING**, **CHAR**, and **VARCHAR**. These types represent alphanumeric text values.

The **STRING**data type represents a sequence of characters with no specified length constraint.\*

If you’re familiar with relational databases, you are probably more accustomed to the character types **CHAR**and **VARCHAR**, which have a specified length.

The **CHAR**type represents fixed-length character sequences, with a precise specified length. Values longer than the specified length are truncated. Values shorter than the specified length are padded with spaces. If you assign the 13-character value **Impala rules!**to a **CHAR**column with length 16, then Hive and Impala will pad that value with three spaces to make it 16 characters long: **Impala rules!␣␣␣**(The three symbols shown in this example represent spaces.)

The **VARCHAR**type represents character sequences with a maximum specified length.

Values longer than the maximum are truncated, but values shorter than the maximum are not padded with spaces. If you attempt to assign the 13-character value **Impala rules!**in a **VARCHAR**column with a maximum length of 10, then Hive and Impala will truncate that value to 10 characters, discarding the last three characters: **Impala rul**. However, if the maximum length is 13 or more, the stored value will be exactly **Impala rules!**(with no extra spaces as you would get with the **CHAR**type).

The table here summarizes these examples.

| Data Type | Description | Value (attempting **Impala rules!**) |
| --- | --- | --- |
| **STRING** | Any number of characters | **Impala rules!** |
| **CHAR(10)** | Exactly 10 characters | **Impala rul** |
| **CHAR(16)** | Exactly 16 characters | **Impala rules!␣␣␣** |
| **VARCHAR(10)** | At most 10 characters | **Impala rul** |
| **VARCHAR(16)** | At most 16 characters | **Impala rules!** |

With **CHAR**types, trailing spaces are ignored in comparisons. With **VARCHAR**and **STRING**values, any trailing spaces are considered in comparisons. (This makes sense, since neither is automatically padded—trailing spaces are not considered to be “padding” in these cases.)

You should generally choose **STRING**over **CHAR**or **VARCHAR**. **STRING**offers greater flexibility and ease of use, and in some cases Hive and Impala have better performance and compatibility when using **STRING**columns. But if you have a particular need for string values with precise lengths or with maximum lengths, then you could use **CHAR**or **VARCHAR**.

**\*Footnote: Actual String Limits**

There actually are practical limits to the length of strings, though in most real-world applications, it's unlikely you'll ever come up against them. For example, in Impala, these are the considerations for lengths of strings (taken from [STRING Data Type](https://www.cloudera.com/documentation/enterprise/latest/topics/impala_string.html" \t "https://www.coursera.org/learn/cloud-storage-big-data-analysis-sql/supplement/JMs2M/_blank)​ in Cloudera's Impala documentation):

* The hard limit on the size of a **STRING**and the total size of a row is 2GB.

If a query tries to process or create a string larger than this limit, it will return an error to the user.

* The limit is 1GB on **STRING**when writing to Parquet files.
* Queries operating on strings with 32KB or less will work reliably and will not hit significant performance or memory problems (unless you have very complex queries, very many columns, and so on.)
* Performance and memory consumption may degrade with strings larger than 32KB.

# Other Data Types

Besides the integer, decimal, and character data types, Hive and Impala support several other simple data types.

The **BOOLEAN**type represents a Boolean value, that is, a **true**or **false**value.

The **TIMESTAMP**type represents an instant in time. **TIMESTAMP**s can represent values with up to nanosecond precision. They are interpreted as being in UTC, or Coordinated Universal Time, but Hive and Impala provide functions for conversion to local timezones.

Hive (but not Impala) provides a **DATE**type, representing a particular day in the form **YYYY-MM-DD**, without a time of day. With Hive, a **TIMESTAMP**can be stripped of the time of day by casting to a **DATE**type.

There’s also the **BINARY**type, which can represent any sequence of raw bytes, and also is supported only by Hive, not by Impala. This is analogous to the **VARBINARY**type in some relational databases.

The table below summarizes these types.

| Data Type | Description | Example Value |
| --- | --- | --- |
| **BOOLEAN** | True or false | **true** |
| **TIMESTAMP** | Instant in time | 2019-02-25 16:51:05 |
| **DATE**(Hive only) | Date without time of day | 2019-02-25 |
| **BINARY**(Hive only) | Raw bytes | N/A |

Hive and Impala also support complex types (**ARRAY**, **MAP**, and **STRUCT**), but they are an advanced topic for this course. If you complete the Honors lessons (Week 5), you will learn about complex types then.

# Examining Data Types

If you're unsure what data types are assigned to a column—perhaps it's an existing table that someone else created, you think you might have made a mistake, or you just forgot how you defined it—there are different ways to get this information.

# Examine the Table Schema

Of course, you can use Hue's Table Browser, or the data source panel on the left in Hue, to view the table's schema. You can view the names of the columns along with their data types. As you learned in the Week 2 reading, “Examining Table Structure,” you can also use the **DESCRIBE**or **DESCRIBE FORMATTED**commands. Both show what columns are in the table, with their data types and sometimes comments. **DESCRIBE FORMATTED**provides a bit more information, including the format and location of the table’s data files.

The **SHOW CREATE TABLE**command can also be used to see a table's definition. You can read the resulting **CREATE TABLE**command to see what the columns are, including what their data types are.

See “Examining Table Structure” in Week 2 if you need a refresher on those commands.

# The Type Of Function in Impala

In Impala (but not Hive), you can also use the **typeof**function in a **SELECT**statement to get the data type:

**SELECT typeof(colname) FROM tablename LIMIT 1;**

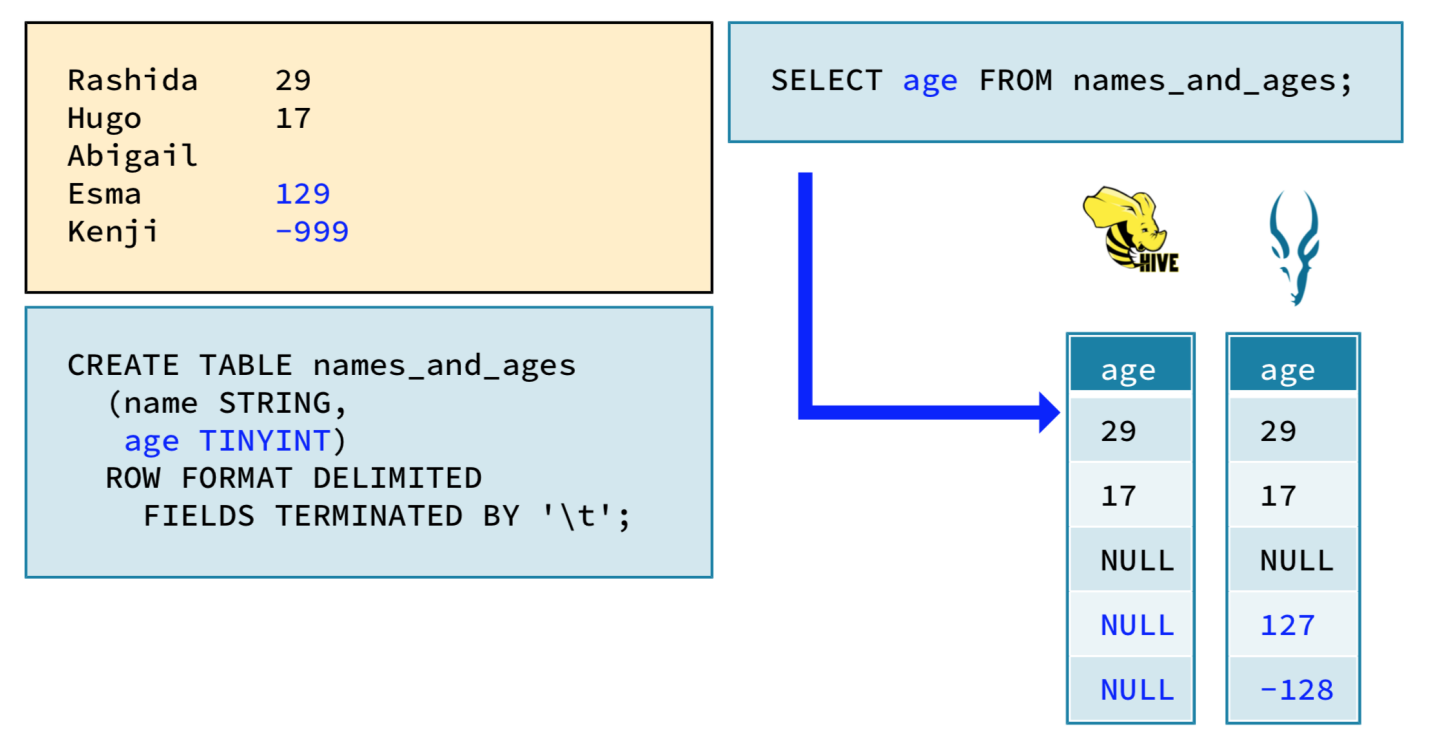
If you don't use **LIMIT 1** it will return one row for each row in the table.

This method also is useful to see what data type an expression returns. Directly examining a table, whether using Hue's graphical interface or by using a command, will not provide this information. Instead of **colname**in the query above, use the expression you're interested in. If the expression doesn't involve a column reference, Impala will allow you to leave off the **FROM**clause.

# Out-of-Range Values

Both Hive and Impala return **NULL** for **DECIMAL** values that are out of range for the target data type (such as **23.63** in a **DECIMAL(3,2)** column). They also both return **-Infinity** or **Infinity** for **FLOAT** and **DOUBLE** types when the value is too large, and zero when the value is close to zero but too small for the data type’s range. (These zero values may be rendered in slightly different ways, for example: **0**, **0.0**, **-0**, or **-0.0**.)

However, there’s an important difference between how Hive and Impala handle out-of-range values in integer columns: Hive returns **NULL** for out-of-range integer values, whereas Impala returns the minimum or maximum value for the column’s specific integer data type. The example shown below illustrates this.

ffig 4.8

# Text Files

Text files are the most basic file type. Virtually any programming language can read and write data in text files, and many applications that data analysts use can work with comma- and tab-delimited text files. They are also human-readable. Values are represented as plain text strings, so you can simply open a text editor and view the data. This is useful when you’re investigating a problem.

However, there are downsides to using text files. When used to store large amounts of data, text files are inefficient. Representing numeric values as strings wastes a great deal of storage space. It’s difficult to represent binary data like images in a text file; this requires using techniques like Base64 encoding. Converting data between text representations and their native data types requires serialization and deserialization, which slows performance.

Overall, text files offer excellent interoperability but poor performance.

# Avro Files

Apache Avro is an efficient data serialization framework. Avro defines a file format that uses optimized binary encoding to store data efficiently. (For an explanation of binary encoding, see “[What Is Binary Encoding](https://www.wisegeek.com/what-is-binary-encoding.htm" \t "https://www.coursera.org/learn/cloud-storage-big-data-analysis-sql/supplement/LWxyk/_blank)” from wisegeek.com.) The Avro format is widely supported by big data tools, and it’s also designed to work across different programming languages and tools outside the typical big data system. Avro files are suitable for long-term data storage.

An Avro data file includes an embedded schema definition, which makes the file self-describing —the file itself provides information about what's in the file. Avro is also built to handle schema evolution. This means that it’s possible to add, remove, or modify columns in a Hive or Impala table without needing to make changes to the existing data stored within Avro files. The Avro framework will accommodate these schema changes, so the table and the existing data files will continue to be compatible,

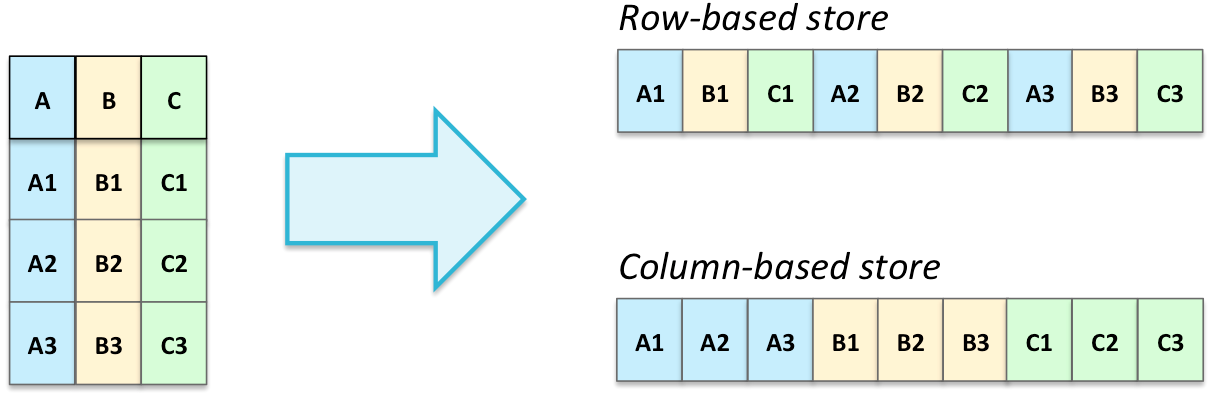
# Parquet Files

Apache Parquet is a columnar file format orginally developed by engineers at Cloudera and Twitter. Parquet was inspired by a project at Google called Dremel.

Columnar, or column-oriented, file formats organize data by column, rather than by row. This makes them more efficient when you need to process only one or a few columns. For example, see Figure 1 below. The rows (1, 2, 3) and columns (A, B, C) of the data are shown in a tabular structure on the left side. The images on the right side represent how this data could be stored in a row-oriented file format (top) and a column-oriented file format (bottom).

When the data is stored in a row-oriented format, the file organizes the data sequentially first by row (Row 1 consists of A1, B1, C1). When the data is stored like this, Hive and Impala must read each full row even if the query requires processing only one column.

When the data is stored in a column-oriented format, the file organizes the data sequentially first by column (Column A has values A1, A2, A3). When the data is stored like this, Hive and Impala can skip over the columns that are not part of the query, and quickly read only the values for the columns that are part of the query. This improves query performance, especially for tables with dozens or hundreds of columns.



Parquet is widely supported by big data tools, including Hive and Impala, and it’s designed to work across different programming languages. Like Avro, Parquet embeds a schema definition in the file, and it supports schema evolution.

Parquet uses advanced optimizations to store data more compactly and to speed up queries. It is most efficient when data is loaded all at once or in large batches; this enables Parquet to take advantage of repeated patterns in the data to store it more efficiently. (See Figure 2 below.)

| **id** | **name** | **city** |
| --- | --- | --- |
| 1 | Alice | Palo Alto |
| 2 | Bob | Sunnyvale |
| 3 | Bob | Palo Alto |
| 4 | Bob | Berkeley |
| 5 | Carol | Berkeley |

IMG_256

also uses compression. Compression encodes data in a way to take up less storage than an uncompressed file will, but there is a time cost when you read or write the file. That is, encoding for less storage means more time needed to compress (encode) the file before writing it, or to decompress (decode) the file after reading it.

Overall, Parquet offers excellent interoperability and excellent performance, making it a popular choice for columnar data storage.

**Apache Avro**

To create an Avro table (that is, a table that will use the Avro format) specify **STORED AS AVRO**.

Avro embeds a schema definition (column names and their data types) in the file itself, but you still must provide a schema definition in your **CREATE TABLE**statement. There are three ways to do this:

1. The usual way of specifying columns and their data types
2. Specifying an Avro schema file
3. Providing the schema as a JSON literal string
4. To create an Avro table in the usual way, provide the name and data type of each column, in parentheses, with the name-type pairs separated by commas:

**CREATE TABLE company\_email\_avro (id INT, name STRING, email STRING)**

**STORED AS AVRO;**

You can avoid listing the full schema in the **CREATE TABLE**statement by storing the table schema in a separate Avro schema file and linking to it as a table property. In your **CREATE TABLE**statement, specify **TBLPROPERTIES ('avro.schema.url'='/path/to/file.avsc');**where **file.avsc**is a JSON file containing an Avro schema.

For example, you can create the same table as above using **order\_details.avsc**with these contents:

**{"type":"record",**

**"name":"company\_email\_avro",**

**"fields":[**

**{"name":"id","type":["null","int"]},**

**{"name":"name","type":["null","string"]},**

**{"name":"email","type":["null","string"]}**

**]}**

(The instances of **"null"**preceding the names of the data types in this schema indicate that the columns can contain **NULL**values.)

Then the **CREATE TABLE**statement could look like this:

**CREATE TABLE company\_email\_avro**

**STORED AS AVRO**

**TBLPROPERTIES ('avro.schema.url'='/path/to/company\_email.avsc');**

The URL path can point to an Avro schema file in HDFS, as shown above. You can also point to other locations, such as an HTTP server using the **http://**protocol or an S3 bucket using the appropriate protocol for your configuration, such as **s3a://**.

Instead of using the Avro schema file, you can put the JSON for the schema into the **CREATE TABLE**statement as a table property using **'avro.schema.literal'**. The following will also create the same table as the previous examples:

CREATE TABLE company\_email\_avro

    STORED AS AVRO

    TBLPROPERTIES ('avro.schema.literal'=

     '{"type":"record",

         "name":"company\_email\_avro",

          "fields":[

                        {"name":"id","type":["null","int"]},

                        {"name":"name","type":["null","string"]},

                        {"name":"email","type":["null","string"]}

        ]}');

If the JSON schema of an Avro data file is not available to you, you can extract it from the Avro data file by using the **avro-tools**command-line utility. The syntax is:

avro-tools getschema /path/to/avro\_data\_file.avro

This command extracts the schema from the specified Avro data file and prints the JSON representation of the schema to the screen. You could then copy and paste this JSON into the **TBLPROPERTIES**clause as described above, or save this JSON to a plain text file using the **.avsc**extension then point to that file in the **TBLPROPERTIES**clause, as previously described. This **avro-tools**utility is installed on the course VM.

**Apache Parquet**

Like with Avro tables, you can create Parquet tables by specifying **STORED AS PARQUET**. You saw this in the Week 2 reading about the **STORED AS**clause. Parquet files also have the schema embedded in the file, but the table definition also needs to include the schema definition, for example:

**CREATE TABLE investors\_parquet**

**(name STRING, amount INT, share DECIMAL(4,3))**

**STORED AS PARQUET;**

With Parquet files, though, Impala (not Hive)provides a shortcut: **LIKE PARQUET '/path/to/file.parq'**. By using this shortcut, you can take the schema directly from a Parquet file—no need for a separate schema file as Avro requires:

CREATE TABLE investors\_parquet

     LIKE PARQUET '/user/hive/warehouse/investors\_parquet/investors.parq'

    STORED AS PARQUET;

Note that in this case, the table being created will use the specified file as one of its data files, because the table's file location will be **/user/hive/warehouse/investors\_parquet**and the referenced Parquet file is in that location. This does not need to be the case; you can use a file that exists elsewhere than where the table's files will be stored.

Note also that this is not the same as cloning a table using **LIKE tablename**.(See the “CREATE TABLE Shortcuts” reading from Week 2.) Using **LIKE tablename**points to an existing table and copies the schema information for that table from the metastore; using **LIKE PARQUET '/path/to/file.parq'**points to the path of a specific Parquet file and copies the schema information from that file. You may use the Parquet file for an existing table, but in that case, it's probably better to use **LIKE tablename**.

Cloudera's documentation for **CREATE TABLE**includes the following notes about using **LIKE PARQUET**:

* Each column in the new table has a comment stating that the SQL column type was inferred from a Parquet file.

The file format of the new table defaults to text, as with other kinds of **CREATE TABLE**statements. To make the new table also use Parquet format, include the clause **STORED AS PARQUET**in the **CREATE TABLE LIKE PARQUET**statement.

* If the Parquet data file comes from an existing Impala table, currently, any **TINYINT**or **SMALLINT**columns are turned into **INT**columns in the new table. Internally, Parquet stores such values as 32-bit integers.

**Chapter 5--** **Conclusion**:

* In this course, you'll get an in-depth look at the SQL SELECT statement and its main clauses. The course focuses on big data SQL engines Apache Hive and Apache Impala, but most of the information is applicable to SQL with traditional RDBMs as well; the instructor explicitly addresses differences for MySQL and PostgreSQL. By the end of the course, you will be able to • explore and navigate databases and tables using different tools; • understand the basics of SELECT statements; • understand how and why to filter results; • explore grouping and aggregation to answer analytic questions; • work with sorting and limiting results; and • combine multiple tables in different ways. To use the hands-on environment for this course, you need to download and install a virtual machine and the software on which to run it. Before continuing, be sure that you have access to a computer that meets the following hardware and software requirements: • Windows, macOS, or Linux operating system (iPads and Android tablets will not work) • 64-bit operating system (32-bit operating systems will not work) • 8 GB RAM or more • 25GB free disk space or more • Intel VT-x or AMD-V virtualization support enabled (on Mac computers with Intel processors, this is always enabled; on Windows and Linux computers, you might need to enable it in the BIOS) • For Windows XP computers only: You must have an unzip utility such as 7-Zip or WinZip installed (Windows XP’s built-in unzip utility will notwork

**LEARNING OUTCOMES**

Big data are the building blocks of programming and data handling. having a thorough knowledge about these helps you to be a strong and efficient coder. It helps to reduce the time needed to perform a function or operation. for example,

if you are searching for an element in an array having large amount of number then rather than using linear search you should use binary search as it first divides the content and then search for it resulting less time consumption.

In this course, you'll get an in-depth look at the SQL SELECT statement and its main clauses. The course focuses on big data SQL engines Apache Hive and Apache Impala, but most of the information is applicable to SQL

**What I have learnt from this course:**

* I have learnt Big data from basic level to advanced level.
* I have learnt topic wise implementation of different big data.
* It improved Knowledge in the Big Data skills.
* I learnt how to write more optimized code with minimum space and time complexity.
* Developed the analytical skills on data and learnt to use them efficiently.
* Learnt how to write most optimized code in time bound environment.
* Learnt different applications of Big data .

**REFERENCES**

* “[www.Coursera.com](http://www.Coursera.com)”
* Coursera course and video lectures.
* Tutorial point.com

\*\*\*