BIG DATA ANALYTICS

Hive

Part-2

https://www.youtube.com/c/RASINENIMADANAMOHANA

Hive: *Outline*

- Introduction
- Installing Hive
  - The Hive Shell
- An Example
- Running Hive
- Comparison with Traditional Databases
- HiveQL
- Tables
- Querying Data
- User-Defined Functions
  - Writing a User Defined Functions
  - Writing a User Defined Aggregate Function

HiveQL

- Hive's SQL dialect, called **HiveQL**, is a mixture of SQL-92, MySQL, and Oracle's SQL dialect (dialect could be "syntax" or "query language").
- The level of SQL-92 support has improved over time, and will likely continue to get better.
- HiveQL also provides features from later SQL standards, such as window functions (also known as analytic functions) from SQL:2003.
- Some of Hive's nonstandard extensions to SQL were inspired by MapReduce, such as multitable inserts and the TRANSFORM, MAP, and REDUCE clauses.

HiveQL

High-level comparison of SQL and HiveQL

Feature	SQL	HiveQL
`Updates`	UPDATE, INSERT, DELETE	UPDATE, INSERT, DELETE
`Transactions`	Supported	Limited support
`Indexes`	Supported	Supported
`Data types`	Integral, floating-point, fixedpoint, text and binary strings, temporal	Boolean, integral, floatingpoint, fixed-point, text and binary strings, temporal, array, map, struct
`Functions`	Hundreds of built-in functions	Hundreds of built-in functions
`Multitable inserts`	Not supported	Supported
`CREATE TABLE...AS SELECT`	Not valid SQL-92, but found in some databases	Supported
`SELECT`	SQL-92	SQL-92. SORT BY for partial ordering, LIMIT to limit number of rows returned

HiveQL

High-level comparison of SQL and HiveQL cont'd.

Feature	SQL	HiveQL
`Joins`	SQL-92, or variants (join tables in the FROM clause, join condition in the WHERE clause)	Inner joins, outer joins, semi joins, map joins, cross joins
`Subqueries`	In any clause (correlated or noncorrelated)	In the FROM, WHERE, or HAVING clauses (uncorrelated subqueries not supported)
`Views`	Updatable (materialized or nonmaterialized)	Read-only (materialized views not supported)
`Extension points`	User-defined functions, stored procedures	User-defined functions, MapReduce scripts

HiveQL: Data Types

- Hive supports both primitive and complex data types.
- Primitives include numeric, Boolean, string, and timestamp types.
- The complex data types include arrays, maps, and structs.
- Hive's data types are listed in below Table.
- Note that the literals shown are those used from within HiveQL; they are not the serialized forms used in the table's storage format.

HiveQL: Data Types

- Hive supports both primitive and complex data types.
- Primitives include numeric, Boolean, string, and timestamp types.
- The complex data types include arrays, maps, and structs.
- Hive's data types are listed in below Table.
- Note that the literals shown are those used from within HiveQL; they are not the serialized forms used in the table's storage format.

HiveQL: Data Types

Table. Hive data types

Category	Type	Description	Literal examples
Primitive	BOOLEAN	True/false value	TRUE
	TINYINT	1-byte (8-bit) signed integer, from -128 to 127	1Y
	SMALLINT	2-byte (16-bit) signed integer, from -32,768 to 32,767	1S
	INT	4-byte (32-bit) signed integer, from -2,147,483,648 to 2,147,483,647	1
	BIGINT	8-byte (64-bit) signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	1L
	FLOAT	4-byte (32-bit) single-precision floating-point number	1.0
	DOUBLE	8-byte (64-bit) double-precision floating-point number	1.0

HiveQL: Data Types

Table. Hive data types cont'd.

Category	Type	Description	Literal examples
Primitive	DECIMAL	Arbitrary-precision signed decimal number	`1.0`
	STRING	Unbounded variable-length character string	`'a', "a"`
	VARCHAR	Variable-length character string	`'a', "a"`
	CHAR	Fixed-length character string	`'a', "a"`
	BINARY	Byte array	`Not supported`
	TIMESTAMP	Timestamp with nanosecond precision	`1325502245000, '2024-08-29 03:04:05.123456789'`
	DATE	Date	`'2024-08-29'`

HiveQL: Data Types

Table. Hive data types cont'd.

Category	Type	Description	Literal examples
Complex	ARRAY	An ordered collection of fields. The fields must all be of the same type	`array(1, 2)`
	MAP	An unordered collection of key-value pairs. Keys must be primitives; values may be any type. For a particular map, the keys must be the same type, and the values must be the same type	`map('a', 1, 'b', 2)`
	STRUCT	A collection of named fields. The fields may be of different types	`struct('a', 1, 1.0), named_struct('col1', 'a', 'col2', 1, 'col3', 1.0)`
	UNION	A value that may be one of a number of defined data types. The value is tagged with an integer (zeroindexed) representing its data type in the union	`create_union(1, 'a', 63)`

HiveQL: Data Types

Primitive types

- Hive's primitive types correspond roughly to Java's, although some names are influenced by MySQL's type names (some of which, in turn, overlap with SQL-92's).
- There is a BOOLEAN type for storing true and false values.
- There are four signed integral types: TINYINT, SMALLINT, INT, and BIGINT, which are equivalent to Java's byte, short, int, and long primitive types, respectively (they are 1-byte, 2-byte, 4-byte, and 8-byte signed integers).
- Hive's floating-point types, FLOAT and DOUBLE, correspond to Java's float and double, which are 32-bit and 64-bit floating-point numbers.

HiveQL: Data Types

Primitive types

- The DECIMAL data type is used to represent arbitrary-precision decimals, like Java's BigDecimal, and are commonly used for representing currency values.
- DECIMAL values are stored as unscaled integers.
- The *precision* is the number of digits in the unscaled value, and the scale is the number of digits to the right of the decimal point.
- Example, `DECIMAL(5,2)` stores numbers between `-999.99` and `999.99`. If the scale is omitted then it defaults to `0`, so `DECIMAL(5)` stores numbers in the range `-99,999` to `99,999` (i.e., integers). If the precision is omitted then it defaults to `10`, so DECIMAL is equivalent to `DECIMAL(10,0)`. The maximum allowed precision is `38`, and the scale must be no larger than the precision.

HiveQL: Data Types

Primitive types

- There are three Hive data types for storing text. STRING is a variable-length character string with no declared maximum length. (The theoretical maximum size STRING that may be stored is `2 GB`, although in practice it may be inefficient to materialize such large values. Sqoop has large object support.
- VARCHAR types are similar except they are declared with a maximum length between `1` and `65355`; for example, `VARCHAR(100)`.
- CHAR types are fixed-length strings that are padded with trailing spaces if necessary; for example, `CHAR(100)`. Trailing spaces are ignored for the purposes of string comparison of `CHAR` values.

HiveQL: Data Types

Primitive types

- The BINARY data type is for storing variable-length binary data.
- The TIMESTAMP data type stores timestamps with nanosecond precision. Hive comes with UDFs for converting between Hive timestamps, Unix timestamps (seconds since the Unix epoch), and strings, which makes most common date operations tractable. TIMESTAMP does not encapsulate a time zone; however, the `to_utc_timestamp` and `from_utc_timestamp` functions make it possible to do time zone conversions.
- The DATE data type stores a date with year, month, and day components.

HiveQL: Data Types

Complex types

- Hive has four complex types: ARRAY, MAP, STRUCT, and UNION.
- ARRAY and MAP are like their namesakes in Java, whereas a STRUCT is a record type that encapsulates a set of named fields.
- A UNION specifies a choice of data types; values must match exactly one of these types.
- Complex types permit an arbitrary level of nesting.

HiveQL: Data Types

Complex types

- Complex type declarations must specify the type of the fields in the collection, using an angled bracket notation, as illustrated in this table definition with three columns (one for each complex type):

```
CREATE TABLE complex (
c1 ARRAY<INT>,
c2 MAP<STRING, INT>,
c3 STRUCT<a:STRING, b:INT, c:DOUBLE>,
c4 UNIONTYPE<STRING, INT>
);
```

- If we load the table with one row of data for ARRAY, MAP, STRUCT, and UNION, the following query demonstrates the field accessor operators for each type:

```
hive> SELECT c1[0], c2['b'], c3.c, c4 FROM complex;

1 2 1.0 {1:63}
```

HiveQL: Operators and Functions

- The usual set of SQL operators is provided by Hive: relational operators (such as `x = 'a'` for testing equality, `x IS NULL` for testing nullity, and `x LIKE 'a%'` for pattern matching), arithmetic operators (such as `x + 1` for addition), and logical operators (such as `x OR y` for logical OR). The operators match those in MySQL, which deviates from SQL-92 because `||` is logical OR, not string concatenation. Use the `concat` function for the latter in both MySQL and Hive.
- Hive comes with a large number of built-in functions-too many to list here-divided into categories that include mathematical and statistical functions, string functions, date functions (for operating on string representations of dates), conditional functions, aggregate functions, and functions for working with XML (using the xpath function) and JSON.

HiveQL: Operators and Functions

- We can retrieve a list of functions from the Hive shell by typing SHOW FUNCTIONS.
- To get brief usage instructions for a particular function, use the DESCRIBE command:

```
hive> DESCRIBE FUNCTION length;

length(str | binary) - Returns the length of str or number
of bytes in binary data
```

HiveQL: Operators and Functions

Conversions

- Primitive types form a hierarchy that dictates the implicit type conversions Hive will perform in function and operator expressions.
- For example, a TINYINT will be converted to an INT if an expression expects an INT; however, the reverse conversion will not occur, and Hive will return an error unless the CAST operator is used.
- The implicit conversion rules can be summarized as follows:
  - Any numeric type can be implicitly converted to a wider type, or to a text type (STRING, VARCHAR, CHAR).
  - All the text types can be implicitly converted to another text type.
  - Perhaps surprisingly, they can also be converted to DOUBLE or DECIMAL.
  - BOOLEAN types cannot be converted to any other type, and they cannot be implicitly converted to any other type in expressions.
  - TIMESTAMP and DATE can be implicitly converted to a text type.

HiveQL: Operators and Functions

Conversions

- We can perform explicit type conversion using CAST.
- For example, `CAST('1' AS INT)` will convert the string `'1'` to the integer value `1`.
- If the cast fails-as it does in `CAST('X' AS INT)`, for example-the expression returns `NULL`.

Hive: *Outline*

- Introduction
- Installing Hive
  - The Hive Shell
- An Example
- Running Hive
- Comparison with Traditional Databases
- HiveQL
- Tables
- Querying Data
- User-Defined Functions
  - Writing a User Defined Functions
  - Writing a User Defined Aggregate Function

HiveQL: Tables

- A Hive table is logically made up of the data being stored and the associated metadata describing the layout of the data in the table.
- The data typically resides in HDFS, although it may reside in any Hadoop filesystem, including the local filesystem or S3 (Simple Storage Service).
- Hive stores the metadata in a relational database and not in HDFS.

HiveQL: Tables

- Managed Tables and External Tables 
- Partitions and Buckets
- Storage Formats
- Importing Data
- Altering Tables
- Dropping Tables

HiveQL: Tables

Managed Tables and External Tables

- When we create a table in Hive, by default Hive will manage the data, which means that Hive moves the data into its warehouse directory.
- Alternatively, we may create an external table, which tells Hive to refer to the data that is at an existing location outside the warehouse directory.

Managed table:

- When we load data into a managed table, it is moved into Hive's warehouse directory.
- For example, this:

```
CREATE TABLE managed_table (dummy STRING);
LOAD DATA INPATH '/user/tom/data.txt' INTO table managed_table;
```

will *move* the file *hdfs://user/tom/data.txt* into Hive's warehouse directory for the `managed_table` table, which is *hdfs://user/hive/warehouse/managed_table*

HiveQL: Tables

Managed Tables and External Tables

Managed table: cont'd.

- If the table is later dropped, using:

```
DROP TABLE managed_table;
```

  the table, including its metadata and its data, is deleted. It bears repeating that since the initial `LOAD` performed a move operation, and the `DROP` performed a delete operation, the data no longer exists anywhere. This is what it means for Hive to manage the data.

HiveQL: Tables

Managed Tables and External Tables

External table:

- An external table behaves differently.
- We control the creation and deletion of the data.
- The location of the external data is specified at table creation time:

```
CREATE EXTERNAL TABLE external_table (dummy STRING)
LOCATION '/user/tom/external_table';
LOAD DATA INPATH '/user/tom/data.txt' INTO TABLE external_table;
```

  With the `EXTERNAL` keyword, Hive knows that it is not managing the data, so it doesn't move it to its warehouse directory. Indeed, it doesn't even check whether the external location exists at the time it is defined. This is a useful feature because it means we can create the data lazily after creating the table.

- When we drop an external table, Hive will leave the data untouched and only delete the metadata.

HiveQL: Tables

Managed Tables and External Tables

How do you choose which type of table to use?

- As a rule of thumb, if we are doing all our processing with Hive, then use managed tables, but if we wish to use Hive and other tools on the same dataset, then use external tables.
- A common pattern is to use an external table to access an initial dataset stored in HDFS (created by another process), then use a Hive transform to move the data into a managed Hive table. This works the other way around, too; an external table (not necessarily on HDFS) can be used to export data from Hive for other applications to use.
- Another reason for using external tables is when we wish to associate multiple schemas with the same dataset.

HiveQL: Tables

- Managed Tables and External Tables
- Partitions and Buckets 
- Storage Formats
- Importing Data
- Altering Tables
- Dropping Tables

HiveQL: Tables

Partitions and Buckets

- Hive organizes tables into *partitions* - a way of dividing a table into coarse-grained parts (rough/not detailed) based on the value of a *partition column*, such as a date.
- Using partitions can make it faster to do queries on slices of the data.
- Tables or partitions may be subdivided further into buckets to give extra structure to the data that may be used for more efficient queries.
- For example, bucketing by user ID means we can quickly evaluate a user-based query by running it on a randomized sample of the total set of users.

HiveQL: Tables

Partitions and Buckets

Partitions

- Partitioning is commonly used in scenarios like log files with timestamps.
- By partitioning by date, all records for the same date are stored together.
- This allows queries restricted to specific dates to run more efficiently, as only the relevant partitions need to be scanned.
- However, partitioning doesn't prevent broader queries; it's still possible to query the entire dataset across multiple partitions.
- A table may be partitioned in multiple dimensions. For example, in addition to partitioning logs by date, we might also subpartition each date partition by country to permit efficient queries by location.

HiveQL: Tables

Partitions and Buckets

Partitions cont'd.

- Partitions are defined at table creation time using the `PARTITIONED BY` clause, which takes a list of column definitions.
- For the hypothetical logfiles example, we might define a table with records comprising a timestamp and the log line itself:

```
CREATE TABLE logs (ts BIGINT, line STRING)
PARTITIONED BY (dt STRING, country STRING);
```

When we load data into a partitioned table, the partition values are specified explicitly:

```
LOAD DATA LOCAL INPATH 'input/hive/partitions/file1'
INTO TABLE logs
PARTITION (dt='2001-01-01', country='GB');
```

HiveQL: Tables

Partitions and Buckets

Partitions cont'd.

- At the filesystem level, partitions are simply nested subdirectories of the table directory.
- After loading a few more files into the logs table, the directory structure might look like this:

```
/user/hive/warehouse/logs
├── dt=2001-01-01/
│   ├── country=GB/
│   │   ├── file1
│   │   └── file2
│   └── country=US/
│       └── file3
└── dt=2001-01-02/
    ├── country=GB/
    │   └── file4
    └── country=US/
        ├── file5
        └── file6
```

HiveQL: Tables

Partitions and Buckets

Partitions cont'd.

- The `logs` table has two date partitions (`2001-01-01` and `2001-01-02`, corresponding to subdirectories called *`dt=2001-01-01`* and *`dt=2001-01-02`*); and two country subpartitions (GB and US, corresponding to nested subdirectories called *`country=GB`* and *`country=US`*). The datafiles reside in the leaf directories.
- We can ask Hive for the partitions in a table using `SHOW PARTITIONS`:

```
hive> SHOW PARTITIONS logs;
dt=2001-01-01/country=GB
dt=2001-01-01/country=US
dt=2001-01-02/country=GB
dt=2001-01-02/country=US
```

HiveQL: Tables

Partitions and Buckets

Partitions cont'd.

- The column definitions in the `PARTITIONED BY` clause are full-fledged table columns, called *`partition columns`*; however, the datafiles do not contain values for these columns, since they are derived from the directory names.
- We can use partition columns in `SELECT` statements in the usual way. Hive performs *`input pruning`* to scan only the relevant partitions.

Example:

```
SELECT ts, dt, line
FROM logs
WHERE country='GB';
```

will only scan *`file1`*, *`file2`*, and *`file4`*. Note, that the query returns the values of the `dt` partition column, which Hive reads from the directory names since they are not in the datafiles.

HiveQL: Tables

Partitions and Buckets

Buckets

- There are two reasons why we might want to organize our tables (or partitions) into buckets.
- The *first* is to enable more efficient queries. Bucketing imposes extra structure on the table, which Hive can take advantage of when performing certain queries. In particular, a join of two tables that are bucketed on the same columns-which include the join columns-can be efficiently implemented as a map-side join.
- The *second* reason to bucket a table is to make sampling more efficient. When working with large datasets, it is very convenient to try out queries on a fraction of our dataset while we are in the process of developing or refining them.

HiveQL: Tables

Partitions and Buckets

Buckets

- We use the `CLUSTERED BY` clause to specify the columns to bucket on and the number of buckets:

```
CREATE TABLE bucketed_users (id INT, name STRING)

CLUSTERED BY (id) INTO 4 BUCKETS;
```

Here we are using the user ID to determine the bucket (which Hive does by hashing the value and reducing modulo the number of buckets), so any particular bucket will effectively have a random set of users in it.

HiveQL: Tables

Partitions and Buckets

Buckets

- In the map-side join case, where the two tables are bucketed in the same way, a mapper processing a bucket of the left table knows that the matching rows in the right table are in its corresponding bucket, so it need only retrieve that bucket (which is a small fraction of all the data stored in the right table) to effect the join.
- This optimization also works when the number of buckets in the two tables are multiples of each other; they do not have to have exactly the same number of buckets.

HiveQL: Tables

Partitions and Buckets

Buckets

- The data within a bucket may additionally be sorted by one or more columns. This allows even more efficient map-side joins, since the join of each bucket becomes an efficient merge sort. The syntax for declaring that a table has sorted buckets is:

```
CREATE TABLE bucketed_users (id INT, name STRING)

CLUSTERED BY (id) SORTED BY (id ASC) INTO 4 BUCKETS;
```

How can we make sure the data in our table is bucketed?

Although it's possible to load data generated outside Hive into a bucketed table, it's often easier to get Hive to do the bucketing, usually from an existing table.

HiveQL: Tables

Partitions and Buckets

Buckets

- Take an unbucketed `users` table:

```
hive> SELECT * FROM users;
0 Nat
2 Joe
3 Kay
4 Ann
```

HiveQL: Tables

Partitions and Buckets

Buckets

- To populate the bucketed table, we need to set the **`hive.enforce.bucketing`** property to true so that Hive knows to create the number of buckets declared in the table definition.
- Then it is just a matter of using the `INSERT` command:

```
INSERT OVERWRITE TABLE bucketed_users

SELECT * FROM users;
```

HiveQL: Tables

Partitions and Buckets

Buckets

- Physically, each bucket is just a file in the table (or partition) directory. The filename is not important, but bucket ***n*** is the ***n***th file when arranged in lexicographic order. In fact, buckets correspond to MapReduce output file partitions: a job will produce as many buckets (output files) as reduce tasks. We can see this by looking at the layout of the `bucketed_users` table we just created.
- Running this command:

```
hive> dfs -ls /user/hive/warehouse/bucketed_users;
```

shows that four files were created, with the following names (the names are generated by Hive):

```
000000_0
000001_0
000002_0
000003_0
```

HiveQL: Tables

Partitions and Buckets

Buckets

```
hive> dfs -ls /user/hive/warehouse/bucketed_users;
```

shows that four files were created, with the following names (the names are generated by Hive):

```
000000_0
000001_0
000002_0
000003_0
```

The first bucket contains the users with IDs 0 and 4, since for an INT the hash is the integer itself, and the value is reduced modulo the number of buckets-four, in this case:

```
hive> dfs -cat /user/hive/warehouse/bucketed_users/000000_0;
0Nat
4Ann
```

HiveQL: Tables

Partitions and Buckets

Buckets

We can see the same thing by sampling the table using the `TABLESAMPLE` clause, which restricts the query to a fraction of the buckets in the table rather than the whole table:

```
hive> SELECT * FROM bucketed_users
> TABLESAMPLE(BUCKET 1 OUT OF 4 ON id);
4 Ann
0 Nat
```

Bucket numbering is 1-based, so this query retrieves all the users from the first of four buckets.

HiveQL: Tables

Partitions and Buckets

Buckets

It's possible to sample a number of buckets by specifying a different proportion (which need not be an exact multiple of the number of buckets, as sampling is not intended to be a precise operation).

For example, this query returns half of the buckets:

```
hive> SELECT * FROM bucketed_users
> TABLESAMPLE(BUCKET 1 OUT OF 2 ON id);
4 Ann
0 Nat
2 Joe
```

HiveQL: Tables

Partitions and Buckets

Buckets

Sampling a bucketed table is very efficient because the query only has to read the buckets that match the `TABLESAMPLE` clause.

Contrast this with sampling a nonbucketed table using the `rand()` function, where the whole input dataset is scanned, even if only a very small sample is needed:

```
hive> SELECT * FROM users
> TABLESAMPLE(BUCKET 1 OUT OF 4 ON rand());
2 Joe
```

HiveQL: Tables

- Managed Tables and External Tables
- Partitions and Buckets
- Storage Formats 
- Importing Data
- Altering Tables
- Dropping Tables

HiveQL: Tables

Storage Formats

- There are two dimensions that govern table storage in Hive: the *row format* and the *file format*.
- The row format dictates how rows, and the fields in a particular row, are stored.
- In Hive, the way rows are formatted is determined by something called a *SerDe*, which stands for *Serializer-Deserializer*.
- When a SerDe acts as a deserializer (which happens when querying a table), it converts data from the file's bytes into objects that Hive can use to work with that row of data.
- When acting as a serializer (which happens during an INSERT or CTAS [**C**reate **T**able **A**s **S**elect] operation), the SerDe converts Hive's internal data into bytes that are written to the output file.

HiveQL: Tables

Storage Formats

- The file format dictates the container format for fields in a row.
- The simplest format is a plain-text file, but there are row-oriented and column-oriented binary formats also available.

1. The default storage format: Delimited text
2. Binary storage formats: Sequence files, Avro datafiles, Parquet files, RCFiles, and ORCFiles
3. Using a custom SerDe: RegexSerDe
4. Storage handlers

HiveQL: Tables

Storage Formats

1. Default Storage Format in Hive: Delimited text

- Default format: When we create a table with no ROW FORMAT or STORED AS clauses, the default format is delimited text with one row per line.
- Row delimiter: Ctrl-A (ASCII code 1), less likely in data than tab characters.
- Collection item delimiter: Ctrl-B for ARRAY/STRUCT, Ctrl-C for MAP keys.
- Nested arrays: Outer array uses Ctrl-B, inner array uses Ctrl-C.
- 8 levels of delimiters: ASCII codes 1-8, only the first three can be overridden.

HiveQL: Tables

Storage Formats

1. Default Storage Format in Hive: Delimited text

- Example Syntax:

```
CREATE TABLE ...
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

- SerDe used: LazySimpleSerDe – deserializes fields lazily but stores data in a long-winded textual format.

HiveQL: Tables

Storage Formats

2. Binary storage formats: Sequence files, Avro datafiles, Parquet files, RCFiles, and ORCFiles

- Binary formats: Specified using the STORED AS clause in CREATE TABLE.
- Row-oriented formats: Avro datafiles (Avro is simply the name of a data serialization system used for efficient data exchange), sequence files.
- Column-oriented formats: Parquet, RCFile (Record Columnar File), ORCFile (Optimized Row Columnar File).
- *Row-oriented formats:* Suitable for processing many columns of a single row.
- *Column-oriented formats:* Best for accessing a few columns across many rows.

HiveQL: Tables

Storage Formats

2. Binary storage formats: Sequence files, Avro datafiles, Parquet files, RCFiles, and ORCFiles cont'd.

- Example: Creating a table in Parquet format:

```
CREATE TABLE users_parquet STORED AS PARQUET
AS
SELECT * FROM users;
```

- Compression in Avro format: Enabled by setting relevant properties.

HiveQL: Tables

Storage Formats

3. Using a custom SerDe: RegexSerDe

- Custom SerDe: Allows for loading data with a regular expression.
- Example:

```
CREATE TABLE stations (usaf STRING, wban STRING, name STRING)
ROW FORMAT SERDE
'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES
("input.regex" = "(\\d{6}) (\\d{5}) (.{29}) .*");
```

- Regular expression: Used for deserialization to convert text rows into columns.
- Limitations: `RegexSerDe` is not efficient for general-purpose storage.

HiveQL: Tables

Storage Formats

3. Using a custom SerDe: RegexSerDe cont'd.

- To populate the table, we use a LOAD DATA statement as before:

```
LOAD DATA LOCAL INPATH "input/ncdc/metadata/stations-fixed-width.txt"
INTO TABLE stations;
```

- When we retrieve data from the table the `SerDe` is invoked for deserialization, as we can see from this simple query, which correctly parses the fields for each row:

```
hive> SELECT * FROM stations LIMIT 4;
010000 99999 BOGUS NORWAY
010003 99999 BOGUS NORWAY
010010 99999 JAN MAYEN
010013 99999 ROST
```

HiveQL: Tables

Storage Formats

4. Storage handlers

- Storage handlers are used for storage systems that Hive cannot access natively, such as HBase.
- Storage handlers are specified using a STORED BY clause, instead of the ROW FORMAT and STORED AS clauses.

HiveQL: Tables

- Managed Tables and External Tables
- Partitions and Buckets
- Storage Formats
- Importing Data 
- Altering Tables
- Dropping Tables

HiveQL: Tables

Importing Data

- We can import data into a Hive table by using the `LOAD DATA` operation, which copies or moves files to the table's directory.
- Alternatively, we can fill a table with data from another Hive table using an `INSERT` statement.
- We can also create and populate a table at the same time using CTAS (short for `CREATE TABLE...AS SELECT`).

HiveQL: Tables

Importing Data: Inserts

- Example of an INSERT statement:

```
INSERT OVERWRITE TABLE target
SELECT col1, col2
FROM source;
```

- For partitioned tables, we can specify the partition to insert into by supplying a PARTITION clause:

```
INSERT OVERWRITE TABLE target
PARTITION (dt='2024-09-04')
SELECT col1, col2
FROM source;
```

HiveQL: Tables

Importing Data: Inserts

- We can specify the partition dynamically by determining the partition value from the SELECT statement:

```
INSERT OVERWRITE TABLE target
PARTITION (dt)
SELECT col1, col2, dt
FROM source;
```

This is known as a *dynamic partition insert*.

HiveQL: Tables

Importing Data: Multitable insert

- In HiveQL, we can turn the INSERT statement around and start with the FROM clause for the same effect:

```
FROM source

INSERT OVERWRITE TABLE target

SELECT col1, col2;
```

The reason for this syntax becomes clear when we see that it's possible to have multiple INSERT clauses in the same query.

This so-called *multitable insert* is more efficient than multiple INSERT statements because the source table needs to be scanned only once to produce the multiple disjoint outputs.

HiveQL: Tables

Importing Data: Multitable insert

- Example that computes various statistics over the weather dataset:

```
FROM records2
INSERT OVERWRITE TABLE stations_by_year
      SELECT year, COUNT(DISTINCT station)
      GROUP BY year
INSERT OVERWRITE TABLE records_by_year
      SELECT year, COUNT(1)
      GROUP BY year
INSERT OVERWRITE TABLE good_records_by_year
      SELECT year, COUNT(1)
      WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9)
      GROUP BY year;
```

There is a single source table (`records2`), but three tables to hold the results from three different queries over the source.

HiveQL: Tables

Importing Data: CREATE TABLE...AS SELECT

- The CREATE TABLE...AS SELECT (CTAS) command lets us store the result of a Hive query in a new table.
- This is useful when the output is too large to display on the console or when we need to do more processing on it.
- The new table's columns are based on the columns from the SELECT clause.
- For example:

```
CREATE TABLE target
AS
SELECT col1, col2
FROM source;
```

This creates a new table called target with two columns, col1 and col2, which have the same data types as in the source table.

The operation is atomic, meaning that if the SELECT query fails, the new table won't be created.

HiveQL: Tables

- Managed Tables and External Tables
- Partitions and Buckets
- Storage Formats
- Importing Data
- Altering Tables 
- Dropping Tables

HiveQL: Tables

Altering Tables

- Hive is flexible with table definitions, allowing changes even after the table is created. However, it's our responsibility to make sure the data matches the new structure.

Renaming a Table:

- We can rename a table using the ALTER TABLE command:

```
ALTER TABLE source RENAME TO target;
```

This not only updates the table's metadata but also renames its directory (e.g., from `/user/hive/warehouse/source to /user/hive/warehouse/target`). For external tables, only the metadata is updated, not the directory.

HiveQL: Tables

Altering Tables

Adding Columns:

- We can add new columns with:

```
ALTER TABLE target ADD COLUMNS (col3 STRING);
```

The new column is added after the existing columns. Since the data files aren't modified, queries will return `null` for the new column unless the data files already had extra fields. Since Hive doesn't allow updating existing records, we'll need another method to update the data files if needed.

Changing Column Metadata:

We can change a column's name or data type as long as the old data type can be interpreted as the new one.

HiveQL: Tables

- Managed Tables and External Tables
- Partitions and Buckets
- Storage Formats
- Importing Data
- Altering Tables
- Dropping Tables 

HiveQL: Tables

Dropping Tables

- The `DROP TABLE` statement deletes the data and metadata for a table. In the case of external tables, only the metadata is deleted; the data is left untouched.
- If we want to delete all the data in a table but keep the table definition, use `TRUNCATE TABLE`. For example:

```
TRUNCATE TABLE my_table;
```

  This doesn't work for external tables; instead, use `dfs -rmr` (from the Hive shell) to remove the external table directory directly.

- In a similar way, if we want to create a new, empty table with the same schema as another table, then use the `LIKE` keyword:

```
CREATE TABLE new_table LIKE existing_table;
```