Transactional System Vs Analytical System

Transactional Systems

Type of Data Handled - Day-to-day transactional data

Operations performed - Insert, Delete, Update

Example - ATM transactions, ecommerce transactions

Best suited systems to handle transactional data - RDBMS (Databases : Ex - Oracle, MySQL, etc,.) **Monolithic Systems**.

Analytical Systems

Type of Data Handled - Historical data

Operations performed - Majorly Read operations to analyze large volumes of data

Example - Analyzing Data of a Sales Campaign

Best suited systems to handle transactional data - Data Warehouses (Ex - Teradata, etc,.) **Distributed Systems** are a best fit to handle such large volumes of data.

Hive is an Open-source Data Warehouse used for analytical requirements to analyze historical data for insights.

Formal Definition of Apache Hive

Hive is a distributed, fault tolerant data warehouse that enables analytics at a massive scale and facilitates querying petabytes of data residing in distributed storage using Hive Query Language **- HQL** (SQL like syntax)

A Hive table comprises of

1. **Actual Data** (present in distributed storage)
2. **Metadata** (Schema or information of data present in Metastore DB)

Key Points :

1. Why is Metadata stored in a Database and not in a Datalake?

Limitations of Datalake

- Updates are hard to perform on a Datalake.
- Datalakes offers high throughput but not capable of providing low latency.

Since the metadata should possess the following properties,

a. Would require frequent modifications/updates
b. Should be possible to access the metadata very quickly

It is best to store the metadata in a metastore which is a Database and not in a Datalake.

2. Schema on Read and Schema on Write

RDBMS systems follow a Schema on Write :

- First the table is created with the desired structure.
- Data is Inserted into the table.
- Data Validation takes place while writing the data.

Hive, a DWH follows a Schema on Read:

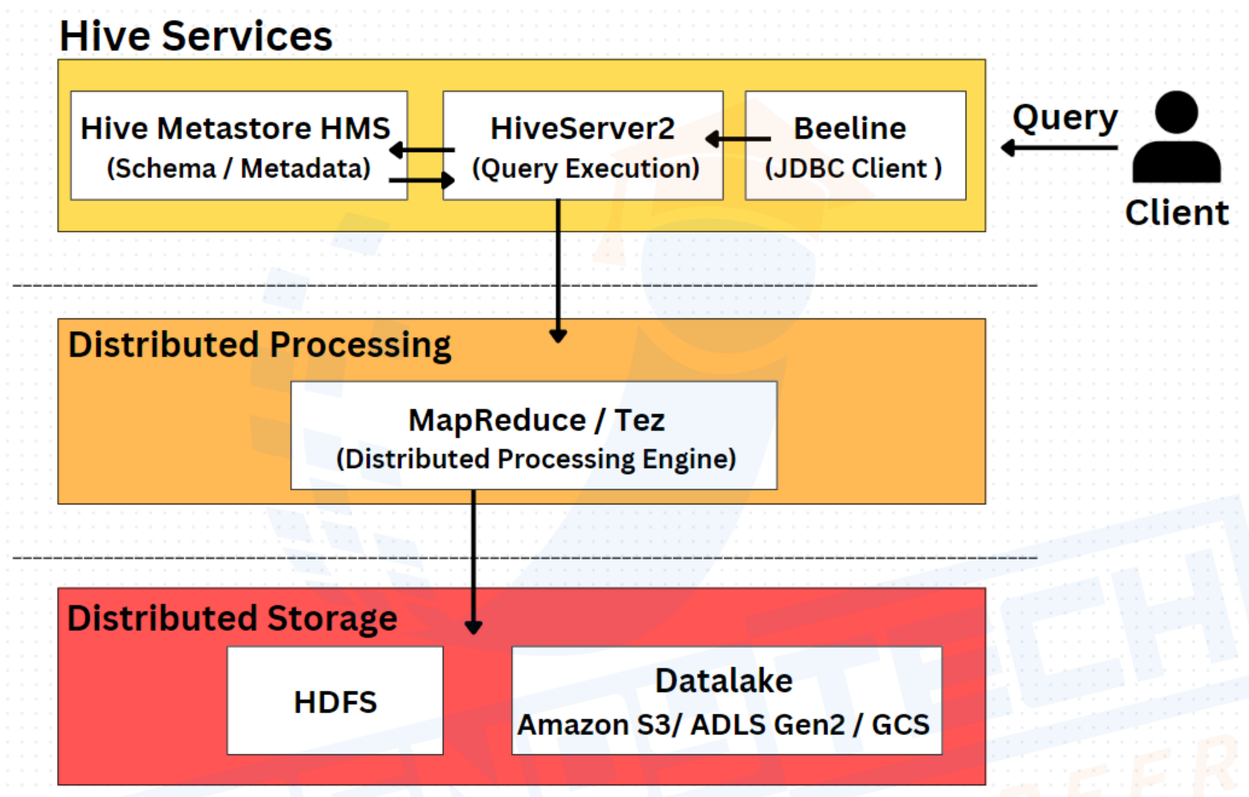
- The Data is present in a Distributed Storage like HDFS / Datalake in the form of files.
- Tables are created on top of the data for a tabular view to query the data using SQL kind of syntax. The table structure gets imposed while reading the data.
- Data Validation takes place while reading the data and imposing the table structure.

Hive Services :

Hive Metastore (HMS) - Stores the Metadata / Schema.

HiveServer2 - Is a service that enables clients to execute queries against hive tables.

JDBC Client (Beeline) - To interact with HiveServer2



Two ways of Connecting to Hive :

1. hive command in the terminal
2. beeline command in the terminal (Preferred way as it is more secure)

   Once in the beeline terminal execute the following :

   !connect jdbc:hive2://m02.itversity.com:10000/;auth=noSasl

   (prompts for username and password)

Note:

- Set the property “hive.metastore.warehouse.dir” to indicate where the data should be stored. (The default location - /user/hive/warehouse is not accessible to all as it is a shared lab. When you try to access it, would throw a permission denied error)

  By using the following, you can set the property to the desired location

  set hive.metastore.warehouse.dir = <path-where-data-should-be-stored>

  Ex :

  ```
  set hive.metastore.warehouse.dir = /user/itv005857/warehouse;
  ```

Sample Commands :

- `create database trendytech_102;`
- `use database trendytech_102;`
- `show tables;`

-
  ```
  CREATE TABLE IF NOT EXISTS demo_table_1 (
  id INT,
  name STRING,
  age INT
  );
  ```

-
  ```
  INSERT INTO demo_table_1 VALUES
  (1, ‘John’, 25),
  (2, ‘Jane’, 30),
  (3, ‘Bob’, 22);
  ```
- To exit from Hive terminal (exit)
  To exit from Beeline terminal (!q)

Note :

1. Hive is meant for high throughput for processing large volumes of data for analytical purposes.
2. Hive table consists of - Actual Data and Metadata, both of these are stored separately.
3. Metadata is stored in a Database named Metastore.
4. The actual data will be stored in any of the distributed storages like HDFS / Amazon S3 / Azure Datalake Storage Gen2/ Google Cloud Storage, etc.
5. If the database needs to be dropped, it has to be empty. Firstly, if the database consists of any tables, the tables need to be dropped before dropping the database.

Types of Tables in Hive

1. **Managed Table**
   - In the case of a Managed table, both the Data and the Metadata are managed and controlled by Hive.

   When to go for Managed Tables?
   When there are no other external tools or technologies accessing the data. When hive is the sole owner of the data, then managed tables are a good fit.

2. **External Table**
   - In the case of an External table, only the Metadata is managed and controlled by Hive.

   When to go for External Tables?
   When there are other external tools or technologies accessing the data, then it is always a best practice to go for external tables to avoid the risk of losing data due to accidental deletion.

Note: When a table is dropped, in both the cases - External | Managed, the metadata will be deleted. However, the data still remains intact in case of an external table as it is stored externally in a distributed storage.

Creating and loading the data to a Hive Managed Table -

Example : Table creation

```
CREATE TABLE IF NOT EXISTS orders_managed_table (
order_id integer,
order_date string,
customer_id integer,
order_status string
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ‘,’
STORED AS TEXTFILE;
```

Example : Loading data to a Managed Table

LOAD DATA INPATH '/user/itv005857/hive_datasets/orders.csv' INTO TABLE **orders_managed_table;**

Note :

- Inserts are possible in case of hive but however, it would be time consuming. Hive is a data warehouse meant for Analytical processing, therefore not optimized for transactional operations.
- Updates and Deletes are not supported by default. However, in the newer version of Hive, it is possible to enable updates and deletes by setting certain properties to perform transactional operations. These properties include the following -

  1. Data should be in ORC format

  2. The table should be bucketed

  3. Property named **transaction** should be set to True.

- Hive supports 3 execution engines

  MapReduce, Spark, Tez

- The execution engine can be changed by setting the property : hive.execution.engine

  Example

  set hive.execution.engine = spark

Requirement - Daily analysis of the new data (A tabular view needs to be created for this new data file that gets added to a folder on a daily basis)

Solution - Creating a Hive External Table

Example : Table creation

CREATE EXTERNAL TABLE IF NOT EXISTS **orders_external_table** (

order_id integer,

order_date string,

```
customer_id integer,
order_status string
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ‘,’
STORED AS TEXTFILE;
LOCATION '/user/itv005857/hive_datasets/orders'
```

Note :

- Command to check the type of the table

    describe formatted <table-name>

- In case of external tables, on dropping the table, only the schema will be lost and the data will be intact in the external location. Hive doesn't have the permission to delete the data as the data would be accessed by other tools.

- If a new file gets added in an incremental manner to the folder where the data file is present, it is automatically detected and shows up when a select query is executed on the hive table.

- Adding a new record with the Insert command is possible in the case of Hive.

- A table gets created even if wrong data types are provided while creating the table. However, when executing a select query, the column with incorrect data type will consist of Nulls as values.

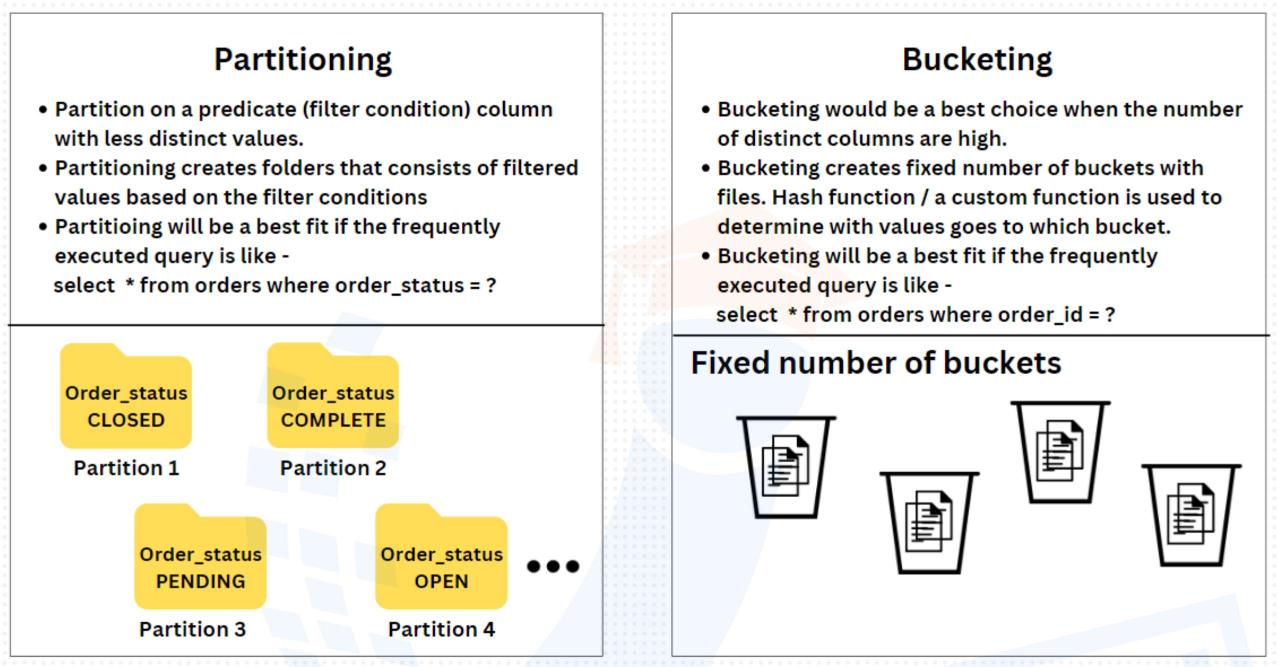
Hive Optimizations

There two levels of optimization possible in case of Hive -

1. **Table Structure Level - Partitioning and Bucketing**
2. **Query Level - Join Optimizations**

Partitioning Vs Bucketing

Both of these optimization techniques have a fundamental goal of dividing and structuring the underneath data in a way that the desired results can be fetched with scanning as minimal data as possible (Skips irrelevant data).


Partitioning
• Partition on a predicate (filter condition) column with less distinct values.
• Partitioning creates folders that consists of filtered values based on the filter conditions
• Partitioing will be a best fit if the frequently executed query is like - select * from orders where order_status = ?
Order_status CLOSED
Partition 1
Order_status COMPLETE
Partition 2
Order_status PENDING
Partition 3
Order_status OPEN
Partition 4
Bucketing
• Bucketing would be a best choice when the number of distinct columns are high.
• Bucketing creates fixed number of buckets with files. Hash function / a custom function is used to determine with values goes to which bucket.
• Bucketing will be a best fit if the frequently executed query is like - select * from orders where order_id = ?
Fixed number of buckets

Partitioning

Creating a Partitioned Table :

CREATE EXTERNAL TABLE IF NOT EXISTS **orders_partition_table** (

order_id integer,

order_date string,

customer_id integer

)

PARTITIONED BY (order_status string)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘,’

STORED AS TEXTFILE;

To see the partitions : **show partitions orders_partition_table;**

Loading data from Normal table to a Partitioned table :

insert into orders_partition_table partition(order_status) select order_id, order_date, customer_id, order_status from orders_managed_table;

This invokes a mapreduce job internally to load the data.

Key points :

- **Two types of partitioning : Static and Dynamic, Dynamic partitioning is mostly used in the industry.**
- **If the following query is executed, then only one folder will be scanned i.e., CLOSED and the other folders are skipped. This would improve the query performance time as only one folder will be scanned.**

  select * from orders_partition_table where order_status = "CLOSED";

- **Command to check an in detail query execution plan**

  explain extended select * from orders_partition_table where order status = "CLOSED";

- **Multi-level partitioning is also possible**

  Example : partition based on country followed by partition based on order status.

Bucketing

It works at the table structure level, where the data underneath will be divided into files rather than folders.

Usecase :

- Helps in faster Join operations.
- Bucketing is more beneficial for join optimizations than with filtering predicates.

Creating a Bucketed Table :

```
CREATE TABLE IF NOT EXISTS orders_bucket_table (
order_id integer,
order_date string,
customer_id integer
)
CLUSTERED BY (order_id) into 4 buckets
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ‘,’
STORED AS TEXTFILE;
```

Inserting data to a bucketed table :

insert into orders_bucket_table select order_id, order_date, customer_id, order_status from orders_managed_table;

The data will be loaded into 4 different buckets. A hash function (like a modulo function) is used to distribute the data into the fixed buckets.

- **Command to check an in detail query execution plan**

  explain extended select * from orders_bucket_table where order_id = 1234;

- **Partitioning followed by bucketing on the same data is possible but the reverse i.e., Bucketing followed by partitioning is not possible. (It is possible to create files in a folder but creating folders in a file is not possible)**

  Example

  Partition-by order_status and Bucket-by order_id

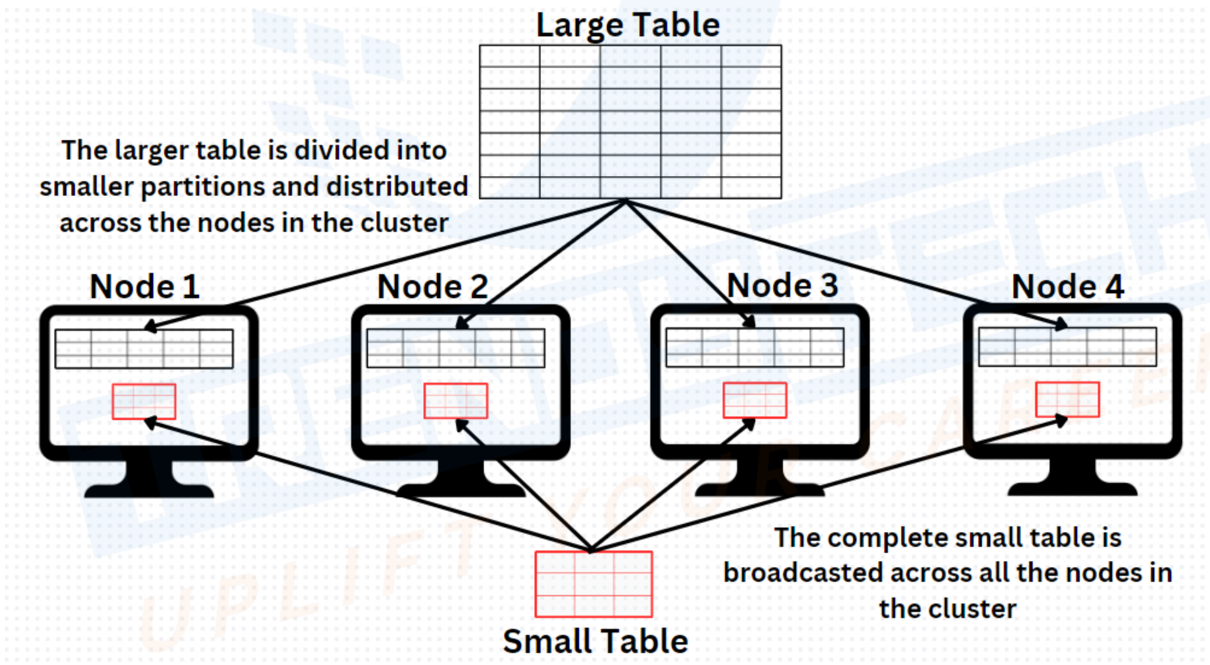
Query Level Optimization - Join Optimization

- The performance of an application can increase several folds if the complex join operations are optimized.
- Joins are costly operations that involve shuffling of data and thereby time consuming.
- Join operations invokes a MapReduce job internally.

1. One of the ways to optimize joins is by using **Map-side Join (~ broadcast joins in Spark)**

   One of the

   When one of the tables is small enough to fit into the memory, then the small table can be broadcasted across all the nodes in the cluster. In this case, each node would have a complete view of the small table and a partial view of the large table.


Large Table
The larger table is divided into smaller partitions and distributed across the nodes in the cluster
Node 1
Node 2
Node 3
Node 4
The complete small table is broadcasted across all the nodes in the cluster
Small Table

2. Optimizing the join of two large tables is possible with **Bucket Map Join.**

   The following **constraints** needs to be met for the bucket map join :

   - Both the tables should be bucketed on the join column.

   - The number of buckets in one table should be an integral multiple of the number of buckets in the other table.

3. Another refined way of optimizing joins is by using **Sort-Merge-Bucket Join(SMB)**

In the case of SMB Lot of pre-processing constraints are set before performing the joins which will eventually lead to performance gains.

These constraints include :

- Both the tables should be bucketed on the join column.

- Number of buckets in both tables should be exactly the same.

- Both the tables should be sorted on the join column.

Note:

- Joining operations will be much easier as the data will be sorted in case of Sort-merge-bucket join.

Internal working of Map-side Join :

Before the execution of the MapReduce job, a **Local Task** gets executed to broadcast the smaller table to all the nodes in the case of Map-side Join.

The Local Process includes the following steps -

1. A Hashmap (Data Structure) is created for the smaller table.
2. On the creation of the Hashmap, it will be put to HDFS.
3. The Hashmap present in the HDFS is then broadcasted to all the nodes in the cluster.
4. After successful broadcast, the Hashmap is now present in the local disk of all the nodes in the cluster (This is also termed as **Distributed Cache**)
5. Hashmap is then loaded in memory for each of the nodes.

Then the MapReduce Job Starts!

Bucket Map Join

This join operation is used when both the tables are large. Only one bucket needs to be loaded to memory at a time.

Requirement for the Bucket-Map Join -

1. Both the tables that need to be joined should be bucketed on Join Column.
2. The number buckets of one table should be an integral multiple of the other.

Creation of bucketed tables for the Bucket Map Join

```
CREATE TABLE IF NOT EXISTS customers_demo_BMJ (
customer_id INT,
customer_fname STRING,
customer_lname STRING,
username STRING,
password STRING,
address STRING,
city STRING,
state STRING,
pincode INT
)
CLUSTERED BY(customer_id) into 4 buckets
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ‘,’
STORED AS TEXTFILE;


CREATE TABLE IF NOT EXISTS orders_demo_BMJ (
order_id integer,
order_date string,
customer_id integer,
order_status string
)
CLUSTERED BY(customer_id) into 8 buckets
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ‘,’
STORED AS TEXTFILE;
```

Loading the data from Normal to Bucketed Table :

```
insert into orders_demo_BMJ select * from orders_external_table;

insert into customers_demo_BMJ select * from customers_external_table;
```

Invoking Join on these Tables

Before invoking the Bucket Map Join, the following properties need to be set to TRUE.

```
set hive.enforce.bucketing=true;
set hive.optimize.bucketmapjoin=true;
```

Query to Invoke Bucket Map Join

```
select o.*, c.* from orders_demo_BMJ o join
customers_demo_BMJ c on o.customer_id = c.customer_id limit 5;
```

Note :

- Property **hive.mapjoin.smalltable.filesize** is used to set the size of the small table for broadcast join. By default it is 25MB

Sort Merge Bucket Join (SMB)

A variation of Bucket Map Join used when both the tables are large. It has more constraints than the bucket map join.

Requirement for the Sort Merge Bucket Join -

1. Both the tables that need to be joined should be bucketed on Join Column.
2. The number of buckets of one table should be exactly equal to the other table.
3. Both the tables should be sorted on the join column.

Before invoking the Sort Merge Bucket Join, the following properties need to be set to TRUE.

```
set hive.input.format =
org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;
set hive.auto.convert.sortmerge.join=true;
set hive.auto.convert.sortmerge.join.noconditionaltask=true;
set hive.optimize.bucketmapjoin=true;
set hive.optimize.bucketmapjoin.sortedmerge=true;
set hive.enforce.bucketing=true;
set hive.enforce.sorting=true;
set hive.auto.convert.join=true;
```

CREATE TABLE IF NOT EXISTS customers_demo_SMB (

```
  customer_id INT,
  customer_fname STRING,
  customer_lname STRING,
  username STRING,
  password STRING,
  address STRING,
  city STRING,
  state STRING,
  pincode INT
)
clustered by(customer_id) sorted by (customer_id asc)  into 4 buckets
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

insert into customers_demo_b1 select * from customers_external;


CREATE TABLE IF NOT EXISTS orders_demo_SMB (
  order_id integer,
  order_date string,
  customer_id integer,
  order_status string
)
clustered by(customer_id) sorted by (customer_id asc) into 4 buckets
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

insert into orders_demo_b1 select * from orders_external;
```

explain extended select o.*,c.* from orders_demo_SMB o join customers_demo_SMB c on o.customer_id = c.customer_id limit 5;
(This indicates that the SMB Join was invoked for the join operation)

Transactional Tables / ACID Properties in Hive

Transactional data is best handled by Databases and a system that supports transactional data should be ACID compliant.

ACID stands for - Atomicity | Consistency | Isolation | Durability

Important requirements of a transactional system

- Systems supporting transactional data should be ACID compliant.

- The data should never fall in an inconsistent state, this ensures highest possible data reliability and integrity.

Atomicity : ensures that a transaction is treated as a single, indivisible unit of work. Either all the changes made in a transaction are committed to the database, or none of them are. If any part of the transaction fails, the entire transaction is rolled back, and the database remains unchanged.

Consistency : ensures that a transaction brings the database from one valid state to another. The database must satisfy a set of integrity constraints before and after the transaction. If a transaction violates any of these constraints, it is rolled back, and the database is left in its original consistent state.

Isolation : ensures that the execution of one transaction is isolated from the execution of other transactions. This means that the intermediate state of a transaction is not visible to other transactions until the transaction is committed. Isolation prevents interference between concurrent transactions, maintaining the integrity of the database.

Durability : ensures that once a transaction is committed, its effects persist even in the event of a system failure. The changes made by the transaction are permanently stored in the database and will survive subsequent system crashes. Durability is often achieved through mechanisms like logging, where a record of the changes made in the transaction is maintained and can be used for recovery.

Key Point :
- We now know that Apache Hive is a Data warehouse and not a Database, if so, what is a need for performing transactions(Inserts, Updates & Deletes) in Hive with ACID compliance?

These are the scenarios where transactions needs to be performed in Hive -

Slowly Changing Dimensions : Let's take an example of a Star schema data warehouse which would consist of different types of tables like

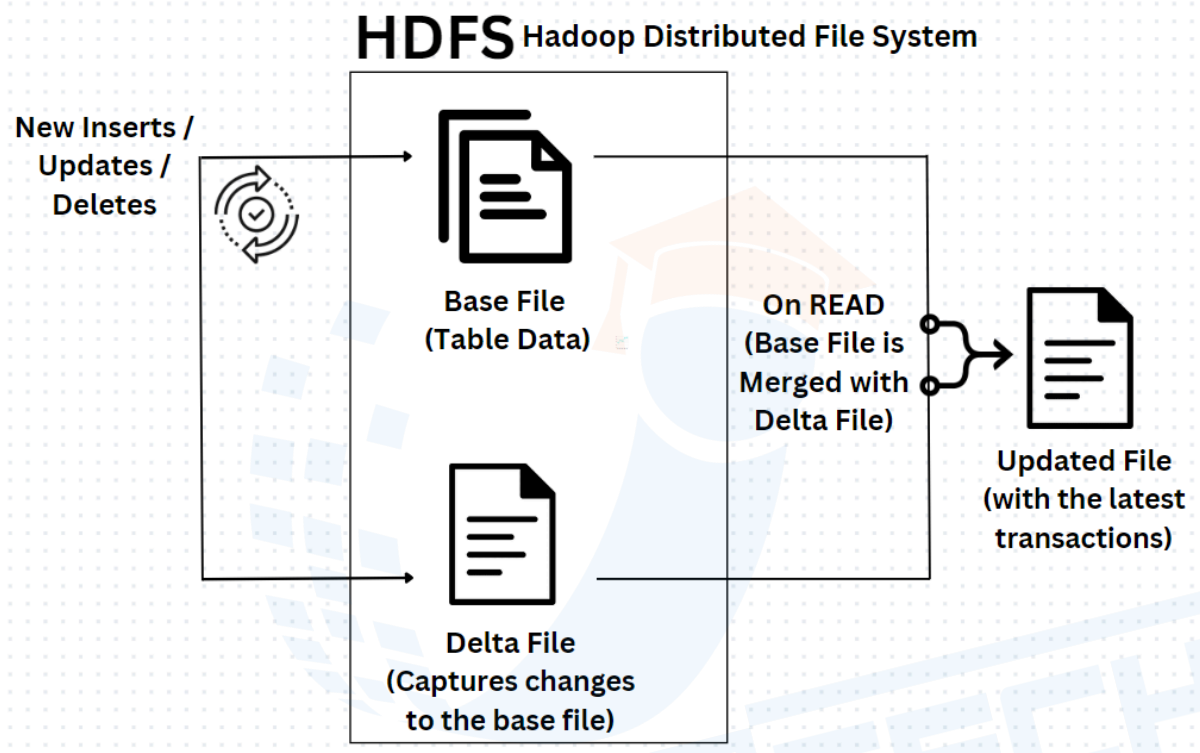
a. Fact table (Ex : Orders table, where most of the transactions take place with very frequent updates/inserts)

b. Dimension table (Ex : Customers table, where there are not very frequent changes taking place. Inserts happen only when new customers are added / Updates to the details of existing customers / Delete the records of non-returning customers. These operations are not very frequent and thereby termed as slowly changing dimensions)

- Basic Design of HDFS is such that no in-place changes (like - Inserts, Updates / Deletes) in the stored files is supported. In order to allow this capability on top of HDFS, Hive follows a standard approach provided in other data warehousing tools i.e., with the **Delta Files.**

Initially, the data of the table is stored in a set of base files. Any new entries/updates/deletes to these base files are captured in a newly created file called Delta files.

These transactions captured in the delta file are merged with the base file while reading the data. So, a latest updated view of the data will be available on reading.


HDFS Hadoop Distributed File System
New Inserts / Updates / Deletes
Base File (Table Data)
Delta File (Captures changes to the base file)
On READ (Base File is Merged with Delta File)
Updated File (with the latest transactions)

The following properties needs to be configured in Hive for the proper functioning of Transactional Tables -

```
SET hive.support.concurrency=true;
SET hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;
SET hive.enforce.bucketing=true;
SET hive.exec.dynamic.partition.mode=nostrict;
SET hive.compactor.initiator.on=true;
SET hive.compactor.worker.threads=1;
```

and
TBLPROPERIES('transactional'='true')

Note:
- These settings are only applicable for managed tables and not for the external tables, as the changes on external tables is beyond the Hive's control.
- Supports only ORC File Formats.

- In order to insert the data to tables, **insert into** need to be used as **LOAD** is not supported in ACID transactional tables.
- All transactions are auto-commit in case of transactional tables.
- A non-ACID compliant table can be converted to an ACID compliant table but reverse is not possible.

Creating a Transactional Table

```
CREATETABLE IF NOT EXISTS orders_trx1 (
order_id integer,
order_date string,
customer_id integer,
order_status string
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS ORC
TBLPROPERTIES ('transactional'='true');
```

To check if the table created is a transactional table -

```
describe formatted orders_trx1;
```

(If the following shows up in the description, then the table is a transactional table -
transactional true
transactional_properties default)

Points :

- Since there would be a large number of small files (delta files) created for transactions, it would become a bottleneck to handle the metadata for such a huge number of files. **Automatic Compaction of ACID Transaction files** by Hive improves the query performance as it merges the smaller files to a single file and thereby reducing the metadata footprint.
- **Snapshot Isolation** that is part of Hive's read semantics. When a read operation starts, hive will logically lock in the state of the warehouse and thereby not impacting the read operation.

Creating an Insert Only Transactional ACID Table

```
CREATETABLE IF NOT EXISTS orders_trx2 (
order_id integer,
order_date string,
customer_id integer,
order_status string
)
ROW FORMAT DELIMITED
```

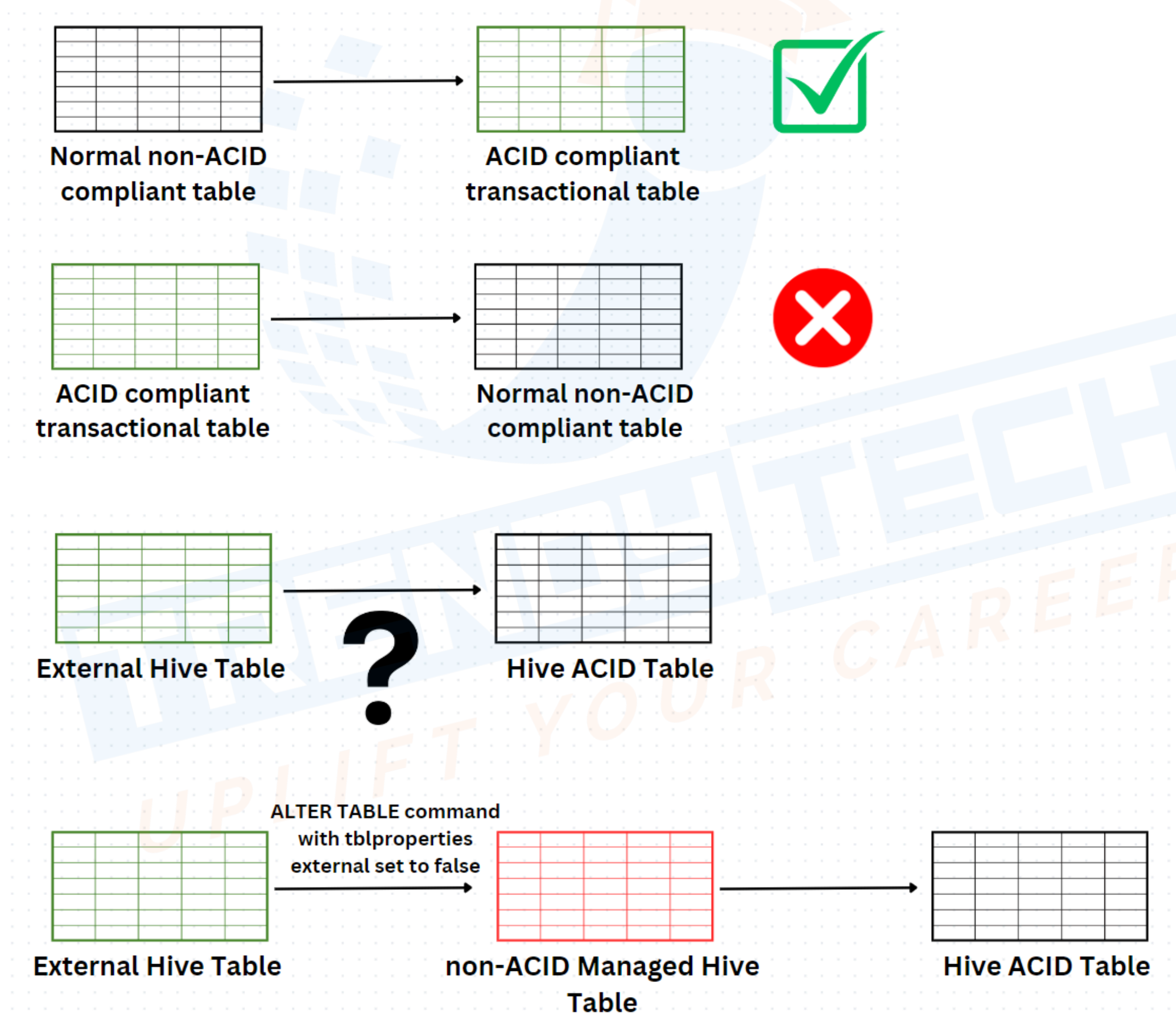
```
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
TBLPROPERTIES ('transactional'='true',
'transactional_properties'='insert_only');
```

To check if the table created is on insert only transactional table -

```
describe formatted orders_trx2;
```

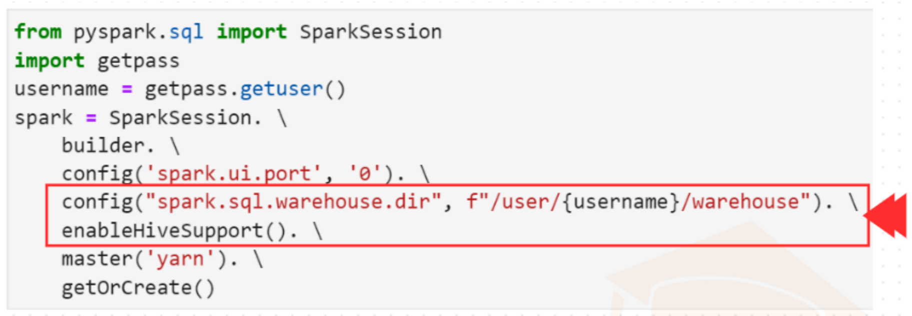
(If the following shows up in the description, then the table is an insert only transactional table -
transactional true
transactional_properties insert_only)

Note : All file formats are supported by the Insert only ACID table.


Normal non-ACID compliant table
ACID compliant transactional table
ACID compliant transactional table
Normal non-ACID compliant table
External Hive Table
?
Hive ACID Table
ALTER TABLE command with tblproperties external set to false
External Hive Table
non-ACID Managed Hive Table
Hive ACID Table

Hive-Spark Integration

Use-case : Creating a table in Hive (Leveraging the Metastore functionality) and processing the data using Spark SQL (Avoiding hive queries as it would involving MapReduce jobs that could be time consuming)



```
from pyspark.sql import SparkSession
import getpass
username = getpass.getuser()
spark = SparkSession. \
    builder. \
    config('spark.ui.port', '0'). \
    config("spark.sql.warehouse.dir", f"/user/{username}/warehouse"). \
    enableHiveSupport(). \
    master('yarn'). \
    getOrCreate()
```

Creating a Spark Session with
- Hive Support enabled
- The warehouse directory location set where the data will be stored

With the above configurations being set, Hive tables can be queried using Spark.

Hive MSCK Repair

MSCK repair is used to correct any metadata changes made at the backend (Ex : hadoop fs -cp source destination). These changes are not done through Spark or Hive and thereby the changes are not recognized by Hive or Spark.

In this scenario, MSCK repair command has to be used to repair the metadata.
spark.sql(“msck repair table <table-name>”);