Hive :

Create table

```
CREATE TABLE IF NOT EXISTS employee_text (
    id INT,
    name STRING,
    age INT,
    salary FLOAT
)
PARTITIONED BY (department STRING)
CLUSTERED BY (id) INTO 4 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION '/user/hive/warehouse/employee_text';
```

Load data into partition wise

```
INSERT INTO TABLE employee PARTITION (department='HR')
SELECT id, name, age, salary FROM some_source_table WHERE department = 'HR';

INSERT INTO employee_text PARTITION (department='HR') VALUES (1, 'John Doe', 30, 60000.0)

LOAD DATA INPATH '/user/hive/warehouse/employee_text/employee_data.txt' INTO TABLE
employee_text PARTITION (department='HR');
```

```
Desc formatted table_name
Show tables
Show databases
```

```
ALTER TABLE old_table_name RENAME TO new_table_name;

ALTER TABLE employee CHANGE name full_name STRING;
```

To create a new table in Hive with the same schema as an existing table without copying the data, you can use the `CREATE TABLE ... LIKE` statement. This command copies the structure of the table but not the data.

```
CREATE TABLE new_table_name LIKE existing_table_name;
```

Remove duplicate

```
SET hivevar:partition_cols = 'id, name, age, salary, department, ...'; -- List all partition columns


WITH dedup_cte AS (
    SELECT
        ${hivevar:partition_cols},
        ROW_NUMBER() OVER (PARTITION BY ${hivevar:partition_cols} ORDER BY id) AS row_num
    FROM employee
)
INSERT OVERWRITE TABLE employee
PARTITION (department)
SELECT ${hivevar:partition_cols}
FROM dedup_cte
WHERE row_num = 1;
```

Struct Example: Creating a table with a struct type

```
CREATE TABLE employee_struct (
    id INT,
    name STRING,
    contact_info STRUCT<phone:STRING, email:STRING>,
    address STRUCT<street:STRING, city:STRING, state:STRING, zip:STRING>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY ':'
MAP KEYS TERMINATED BY '='
LINES TERMINATED BY '\n';

-- Array Example: Creating a table with an array type
CREATE TABLE student_scores (
    student_id INT,
    scores ARRAY<INT>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY ':'
MAP KEYS TERMINATED BY '='
LINES TERMINATED BY ‘\n';
```

Lateral view we can use explode

```
SELECT ...
FROM table_name
LATERAL VIEW explode(array_column) exploded_table AS
column_name;
```

In Hive, the inability to directly delete or update records stems from its design and architecture, which is primarily geared towards batch processing and storing data in a distributed manner across a cluster of nodes. Here are the key reasons why Hive does not support delete and update operations like traditional relational databases:

```
SELECT

    get_json_object(json_string, '$.address.street') AS street

FROM json_data;
```

Hive seems to support IN, NOT IN, EXIST and NOT EXISTS from 0.13.

```
select count(*)
from flight a
where not exists(select b.tailnum from plane b where b.tailnum = a.tailnum);
```

Managed table

A managed table in Hive refers to a table where Hive manages both the data and the metadata. This means that when you drop a managed table, Hive will delete both the metadata (table definition) and the associated data files from HDFS (Hadoop Distributed File System).

```
CREATE TABLE my_managed_table (

   id INT,

   name STRING,

   age INT

);
```

External Table

An external table in Hive is one where Hive manages only the metadata (table definition), while the data files remain external to Hive's control.

When you drop an external table (`DROP TABLE`), Hive only deletes the metadata entry from the metastore. It does not delete the underlying data files from the location they reside in (HDFS or any other supported file system).

```
CREATE EXTERNAL TABLE my_external_table (
   id INT,
   name STRING,
   age INT
)
LOCATION ‘/path/to/data';
```

DISTRIBUTE BY Clause

The `DISTRIBUTE BY` clause is used to specify how data should be distributed among reducers during the shuffle phase of a query. It determines which reducer receives which keys based on the specified column(s).

```
CREATE TABLE table_name (

   column1 data_type,

   column2 data_type,

)DISTRIBUTE BY column_name;
```

If we have same value for the column then it will go to same reducer it is not doing any types of sorting and ordering.

CLUSTERED BY Clause

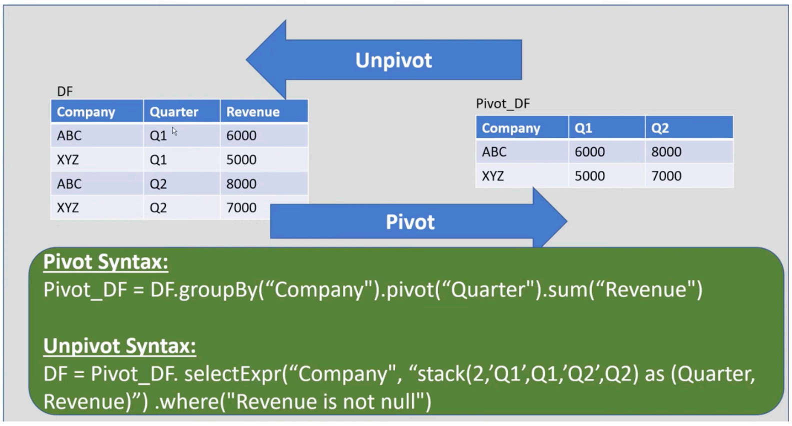
The `CLUSTERED BY` clause is used to organize the data stored in the table into buckets or clusters based on the values of one or more columns. This is also known as data clustering.

```
CREATE TABLE table_name (

    column1 data_type,

    column2 data_type,

    ...

)

CLUSTERED BY (column_name1, column_name2, ...)

INTO num_buckets BUCKETS;
```

Dirstubeted by and sort by become =clustered by it means duplicate value of the column in same reducer and in sorting way

Distribution vs. Clustering: `DISTRIBUTE BY` focuses on how data is distributed among reducers during query execution, optimizing data movement. `CLUSTERED BY` organizes data into buckets or clusters based on column values, optimizing data retrieval.

Reducers vs. Buckets: `DISTRIBUTE BY` affects how data is partitioned across reducers. `CLUSTERED BY` affects how data is stored within files (buckets) based on column values.



DF

Company	Quarter	Revenue
ABC	Q1	6000
XYZ	Q1	5000
ABC	Q2	8000
XYZ	Q2	7000

Pivot_DF

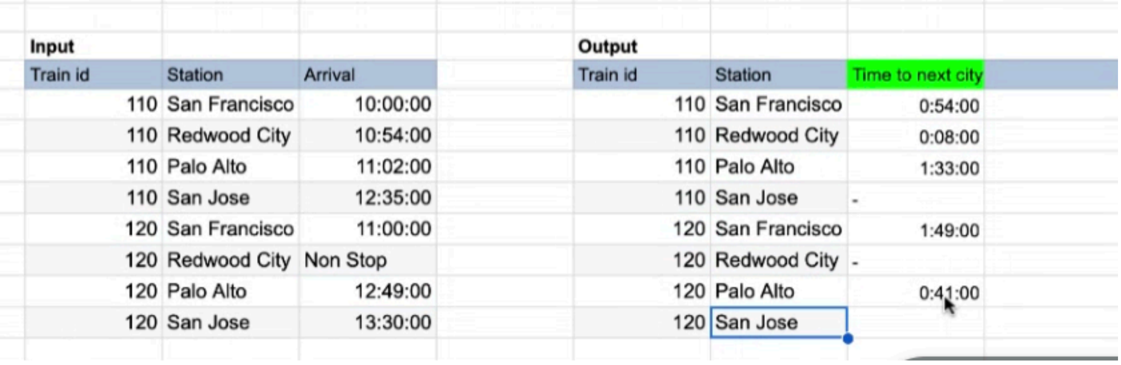
Company	Q1	Q2
ABC	6000	8000
XYZ	5000	7000

Pivot Syntax:

Pivot_DF = DF.groupBy(“Company").pivot(“Quarter").sum(“Revenue")

Unpivot Syntax:

DF = Pivot_DF. selectExpr(“Company", “stack(2,’Q1’,Q1,’Q2’,Q2) as (Quarter, Revenue)”) .where("Revenue is not null")



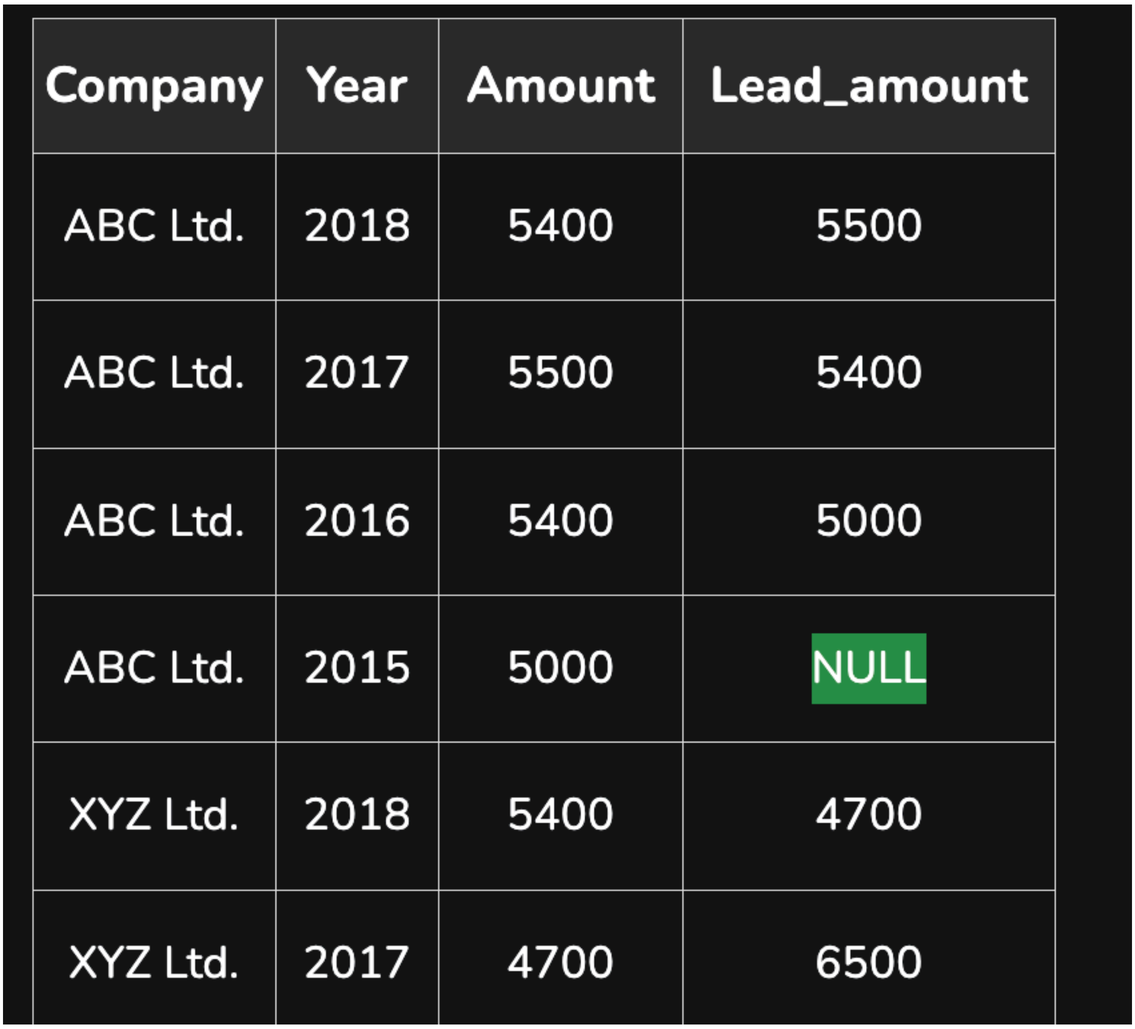
Input

Train id	Station	Arrival
110	San Francisco	10:00:00
110	Redwood City	10:54:00
110	Palo Alto	11:02:00
110	San Jose	12:35:00
120	San Francisco	11:00:00
120	Redwood City	Non Stop
120	Palo Alto	12:49:00
120	San Jose	13:30:00

Output

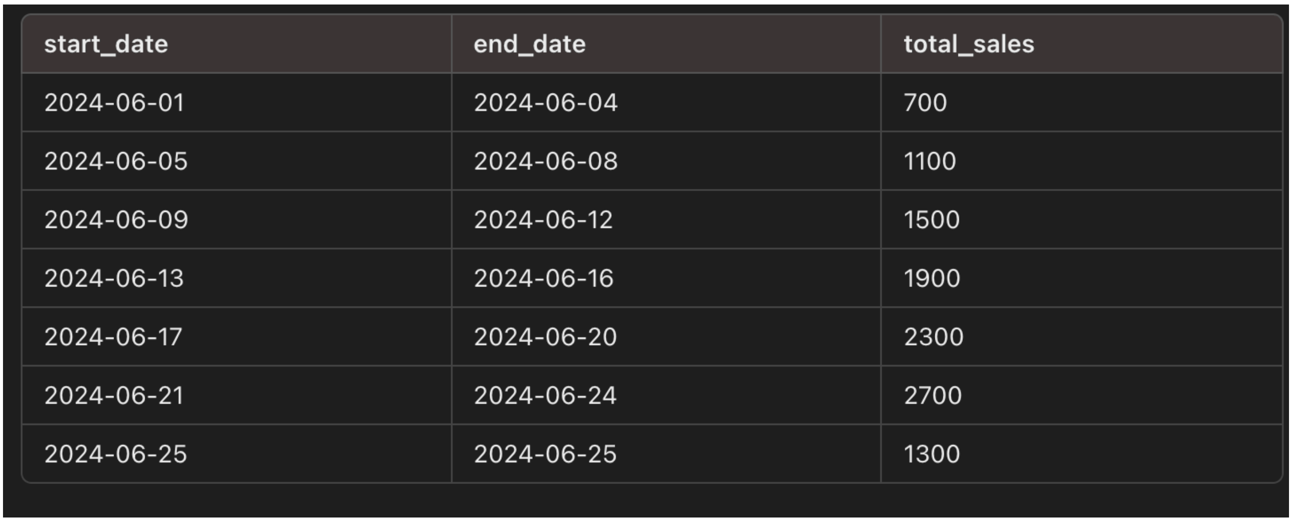
Train id	Station	Time to next city
110	San Francisco	0:54:00
110	Redwood City	0:08:00
110	Palo Alto	1:33:00
110	San Jose	-
120	San Francisco	1:49:00
120	Redwood City	-
120	Palo Alto	0:41:00
120	San Jose	

```sql
WITH next_station
     AS (SELECT train_id,
                station,
                arrival,
                Lead(arrival)
                  OVER (
                    partition BY train_id
                    ORDER BY arrival) AS Next_Arrival
         FROM   train_schedule)
SELECT train_id,
       station,
       arrival,
       CASE
         WHEN next_arrival IS NOT NULL THEN Sec_to_time(Timestampdiff(second,
                                                        arrival,
                                                        next_arrival))

         ELSE '-'
       END AS Time_to_next_city
FROM   next_station
ORDER  BY train_id,
          arrival;
```



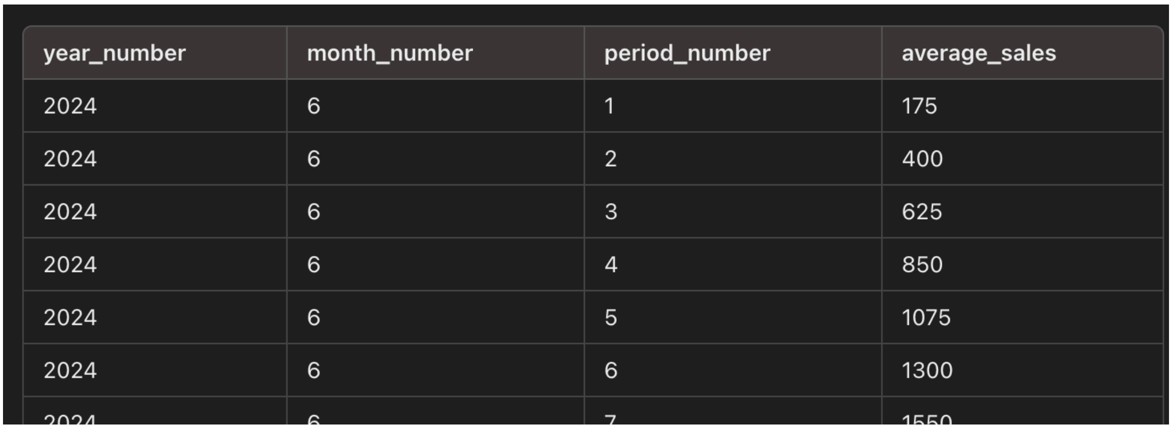
Company	Year	Amount	Lead_amount
ABC Ltd.	2018	5400	5500
ABC Ltd.	2017	5500	5400
ABC Ltd.	2016	5400	5000
ABC Ltd.	2015	5000	NULL
XYZ Ltd.	2018	5400	4700
XYZ Ltd.	2017	4700	6500

```sql
SELECT Min(visited_on) AS start_date,
       Max(visited_on) AS end_date,
       Sum(sales)      AS total_sales
FROM   (SELECT visited_on,
               sales,
               Row_number()
                 OVER (
                   ORDER BY visited_on) - 1 AS row_num
        FROM   sales_data) AS numbered_sales
GROUP  BY Floor(row_num / 4)
ORDER  BY Min(visited_on);
```



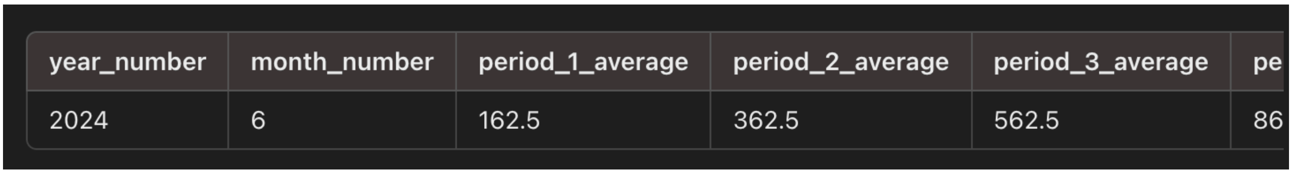
start_date	end_date	total_sales
2024-06-01	2024-06-04	700
2024-06-05	2024-06-08	1100
2024-06-09	2024-06-12	1500
2024-06-13	2024-06-16	1900
2024-06-17	2024-06-20	2300
2024-06-21	2024-06-24	2700
2024-06-25	2024-06-25	1300

```sql
WITH period_data
     AS (SELECT sales,
                date,
                ( Cast((Extract(day FROM date) - 1) / 4 AS INTEGER)
                  + 1 ) AS period_number
         FROM   sales_data
         WHERE  date BETWEEN '2024-06-01' AND '2024-06-30')
SELECT Extract(year FROM date)  AS year_number,
       Extract(month FROM date) AS month_number,
       period_number,
       Avg(sales)               AS average_sales
FROM   period_data
GROUP  BY Extract(year FROM date),
          Extract(month FROM date),
          period_number
ORDER  BY year_number,
          month_number,
          period_number;
```



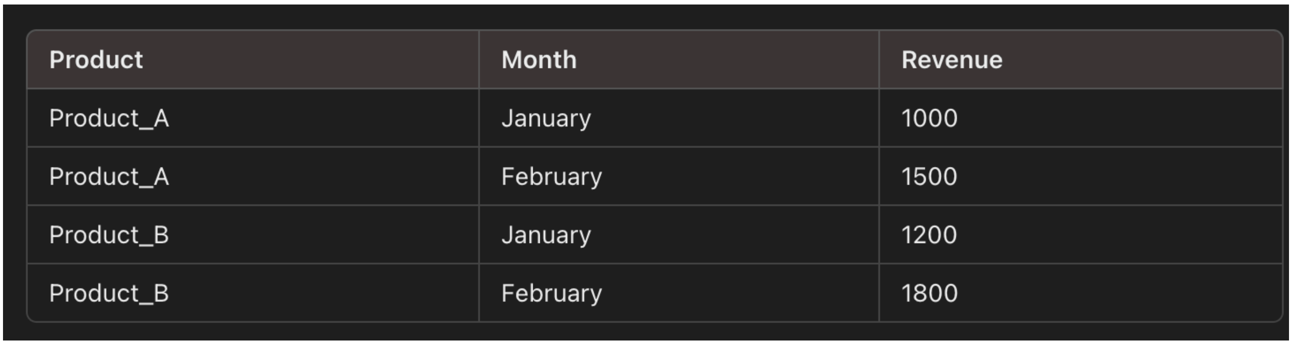
year_number	month_number	period_number	average_sales
2024	6	1	175
2024	6	2	400
2024	6	3	625
2024	6	4	850
2024	6	5	1075
2024	6	6	1300
2024	6	7	1550

```
WITH period_data AS
(
       SELECT sales,
              date,
              (Extract(day FROM date) - 1) / 4 + 1 AS period_number
       FROM   sales_data
       WHERE                date BETWEEN '2024-06-01' AND    '2024-06-30' )
SELECT   Extract(year FROM  date)                    AS year_number,
         Extract(month FROM date)                    AS month_number,
         Avg(sales) filter (WHERE period_number = 1) AS period_1_average,
         avg(sales) filter (WHERE period_number = 2) AS period_2_average,
         avg(sales) filter (WHERE period_number = 3) AS period_3_average,
         avg(sales) filter (WHERE period_number = 4) AS period_4_average,
         avg(sales) filter (WHERE period_number = 5) AS period_5_average,
         avg(sales) filter (WHERE period_number = 6) AS period_6_average,
         avg(sales) filter (WHERE period_number = 7) AS period_7_average
FROM     period_data
GROUP BY extract(year FROM  date),
         extract(month FROM date)
ORDER BY year_number,
         month_number;
```

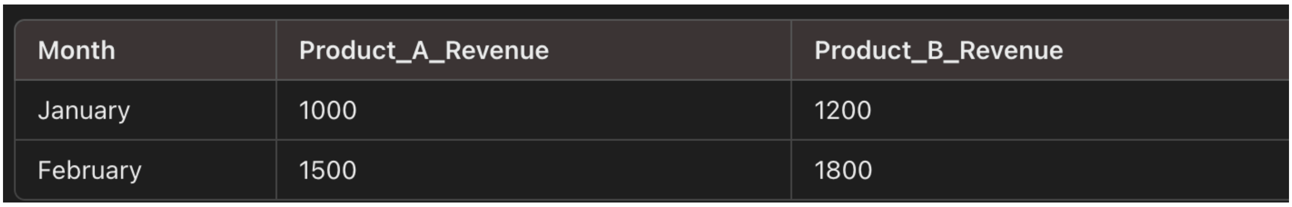


year_number	month_number	period_1_average	period_2_average	period_3_average	pe
2024	6	162.5	362.5	562.5	86

```
SELECT month,
       [product_a] AS Product_A_Revenue,
       [product_b] AS Product_B_Revenue
FROM   (SELECT product,
               month,
               revenue
        FROM   sales) AS SourceTable
       PIVOT ( Sum(revenue)
             FOR product IN ([Product_A],
                             [Product_B]) ) AS pivottable;
```



Product	Month	Revenue
Product_A	January	1000
Product_A	February	1500
Product_B	January	1200
Product_B	February	1800



Month	Product_A_Revenue	Product_B_Revenue
January	1000	1200
February	1500	1800