```sql
-- Create a database
CREATE DATABASE IF NOT EXISTS student_db;

-- Use the database
USE student_db;

-- Create a table for storing student information
CREATE TABLE IF NOT EXISTS student (
   student_id INT,
   name STRING,
   age INT,
   major STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

-- Insert data into the table
INSERT INTO TABLE student VALUES
   (1, 'Alice', 20, 'Computer Science'),
   (2, 'Bob', 22, 'Mathematics'),
   (3, 'Charlie', 21, 'Physics');

-- Alternatively, use a LOAD command to add data from an external file
LOAD DATA LOCAL INPATH '/path/to/student_data.csv' INTO TABLE student;

SELECT * FROM student;

-- Find students majoring in Computer Science
SELECT * FROM student WHERE major = 'Computer Science';
```

```
-- Find students older than 21
SELECT * FROM student WHERE age > 21;
```

Update (U): Modifying data in the table

Hive doesn't support direct updates to tables in traditional ways (like SQL's `UPDATE`), but you can achieve it using an **INSERT OVERWRITE** approach:

```
-- Update the major of a student (e.g., Bob to 'Statistics')
INSERT OVERWRITE TABLE student
SELECT
  student_id,
  name,
  age,
  CASE
    WHEN name = 'Bob' THEN 'Statistics'
    ELSE major
  END AS major
FROM student;
```

Delete (D): Deleting data from the table

Hive doesn't support `DELETE` in the traditional sense. Instead, you can filter out the data you want to keep and overwrite the table:

```
-- Delete a student (e.g., remove Charlie)
INSERT OVERWRITE TABLE student
SELECT * FROM student WHERE name != 'Charlie';


DROP TABLE IF EXISTS student;
```

Create (C): Using Collection Types in Table Creation

```
CREATE TABLE student_with_collections (
   student_id INT,
   name STRING,
   age INT,
   major STRING,
   grades ARRAY<INT>,              -- An array of grades
   contact_info MAP<STRING, STRING>, -- A map for contact details (e.g., phone and email)
   address STRUCT<city:STRING, zip:INT> -- A struct for address details
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY '|'
MAP KEYS TERMINATED BY ':'
STORED AS TEXTFILE;
```

```
-- Insert data into the table
INSERT INTO TABLE student_with_collections VALUES
   (1, 'Alice', 20, 'Computer Science', ARRAY(85, 90, 95), MAP('phone', '1234567890', 'email',
'alice@example.com'), NAMED_STRUCT('city', 'New York', 'zip', 10001)),
   (2, 'Bob', 22, 'Mathematics', ARRAY(78, 88, 92), MAP('phone', '9876543210', 'email',
'bob@example.com'), NAMED_STRUCT('city', 'Los Angeles', 'zip', 90001)),
   (3, 'Charlie', 21, 'Physics', ARRAY(80, 85, 90), MAP('phone', '5551234567', 'email',
'charlie@example.com'), NAMED_STRUCT('city', 'Chicago', 'zip', 60601));
```

Querying Data from Collection Types

```
SELECT * FROM student_with_collections;
```

```
-- Retrieve the first grade of each student
SELECT name, grades[0] AS first_grade FROM student_with_collections;

-- Filter students with a grade above 90
SELECT * FROM student_with_collections WHERE ARRAY_CONTAINS(grades, 90);


-- Retrieve the city of each student
SELECT name, address.city AS city FROM student_with_collections;

-- Filter students based on ZIP code
SELECT * FROM student_with_collections WHERE address.zip = 90001;


INSERT OVERWRITE TABLE student_with_collections
SELECT
  student_id,
  name,
  age,
  major,
  CASE
    WHEN name = 'Alice' THEN ARRAY(88, 92, 96)
    ELSE grades
  END AS grades,
  contact_info,
  address
FROM student_with_collections;
```

```sql
INSERT OVERWRITE TABLE student_with_collections

SELECT * FROM student_with_collections WHERE name != 'Charlie';




-- Explode grades into multiple rows for analysis

SELECT name, grade

FROM student_with_collections LATERAL VIEW EXPLODE(grades) grade_table AS grade;



-- Retrieve all keys and values in the contact_info map

SELECT name, contact_key, contact_value

FROM student_with_collections

LATERAL VIEW EXPLODE(contact_info) map_table AS contact_key, contact_value;


-- Filter students based on city name

SELECT * FROM student_with_collections WHERE address.city = 'New York';
```

Partitioning in Hive

Partitioning organizes data into separate directories based on a column (or multiple columns). This helps improve query performance by scanning only the relevant partitions.

Creating a Partitioned Table

We’ll partition the `student` table by the `major` column

```
CREATE TABLE student_partitioned (
   student_id INT,
   name STRING,
   age INT
)
PARTITIONED BY (major STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

Inserting Data into the Partitioned Table

You must specify the partition value when inserting data.

```
-- Insert data into the "Computer Science" partition
INSERT INTO TABLE student_partitioned PARTITION (major='Computer Science')
VALUES (1, 'Alice', 20);

-- Insert data into the "Mathematics" partition
INSERT INTO TABLE student_partitioned PARTITION (major='Mathematics')
VALUES (2, 'Bob', 22);

-- Insert data into the "Physics" partition
INSERT INTO TABLE student_partitioned PARTITION (major='Physics')
VALUES (3, 'Charlie', 21);
-- Show all partitions in the table
SHOW PARTITIONS student_partitioned;
```

```
-- Fetch all students from the "Computer Science" partition

SELECT * FROM student_partitioned WHERE major = 'Computer Science';


LOAD DATA LOCAL INPATH '/path/to/computer_science_students.csv'

INTO TABLE student_partitioned PARTITION (major='Computer Science');
```

Bucketing in Hive

Bucketing divides data into smaller, fixed-size files based on a column's hash function, often improving query performance for operations like joins.

Creating a Bucketed Table

We’ll bucket the `student` table by the `student_id` column into 4 buckets.

```
CREATE TABLE student_bucketed (
   student_id INT,
   name STRING,
   age INT,
   major STRING
)
CLUSTERED BY (student_id) INTO 4 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

SET hive.enforce.bucketing = true;
```

```
-- Insert data into the bucketed table
INSERT INTO TABLE student_bucketed
VALUES
   (1, 'Alice', 20, 'Computer Science'),
   (2, 'Bob', 22, 'Mathematics'),
   (3, 'Charlie', 21, 'Physics'),
   (4, 'David', 23, 'Statistics'),
   (5, 'Eve', 19, 'Biology');
```

```
-- Fetch students based on student_id
SELECT * FROM student_bucketed WHERE student_id = 1;
```

You can combine **partitioning** and **bucketing** for better data organization and query optimization.

```
CREATE TABLE student_partitioned_bucketed (
    student_id INT,
    name STRING,
    age INT
)
PARTITIONED BY (major STRING)
CLUSTERED BY (student_id) INTO 3 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

```
-- Insert data into the "Computer Science" partition and bucket by student_id

INSERT INTO TABLE student_partitioned_bucketed PARTITION (major='Computer Science')

VALUES (1, 'Alice', 20), (4, 'David', 23);


-- Fetch students from a specific partition

SELECT * FROM student_partitioned_bucketed WHERE major = 'Computer Science';


-- Fetch a specific student within a partition

SELECT * FROM student_partitioned_bucketed WHERE major = 'Computer Science' AND student_id = 1;


-- Show partitions in the table

SHOW PARTITIONS student_partitioned_bucketed;


-- To check the files in a specific partition (use shell command on the Hadoop file system)

hadoop fs -ls /path/to/hive/warehouse/student_partitioned_bucketed/major=Computer Science/
```

🎬 **Partitioning** improves query performance by dividing data into directories based on column values.
🎬 **Bucketing** splits data into smaller files within partitions, enabling faster joins and better data distribution.