



**Term:** Fall 2023    **Subject:** Computer Science & Engineering (CSE)    **Number:** 512

**Course Title:** Distributed Database Systems (CSE 512)

**Team:** PVS

---

## Part 2: Fragmentation and Replication Techniques

**Problem Statement:** Implement fragmentation and replication techniques for optimizing the performance of the distributed database system in your chosen topic.

**Tasks Completed:** Horizontal Fragmentation, Vertical Fragmentation and Replication Setup.

**Code and Explanation:**

Main.py

```
# Modifying tables to implement Vertical and Horizontal Partitioning:
# Creating two sub-tables of Users table, UsersBasic and UsersSensitive as a part of Vertical Partitioning:
USERS_BASIC_TABLE_QUERY = ("CREATE TABLE IF NOT EXISTS UsersBasic ("
    "UserID INT AUTO_INCREMENT PRIMARY KEY,"
    "UserName VARCHAR(255) NOT NULL,"
    "FirstName VARCHAR(255) NOT NULL,"
    "LastName VARCHAR(255) NOT NULL,"
    "Email VARCHAR(255) NOT NULL UNIQUE);")
pc.vertical_partitioning(conn, USERS_BASIC_TABLE_QUERY)

USERS_SENSITIVE_TABLE_QUERY = ("CREATE TABLE IF NOT EXISTS UsersSensitive ("
    "UserID INT PRIMARY KEY,"
    "Password VARCHAR(255) NOT NULL,"
    "PhoneNumber VARCHAR(15),"
    "Address TEXT,"
    "RegistrationDate TIMESTAMP,"
    "Region VARCHAR(255) NOT NULL,"
    "LastLogin TIMESTAMP);")
pc.vertical_partitioning(conn, USERS_SENSITIVE_TABLE_QUERY)
```

```

# Altering tables to create partitions in the existing tables
# (MySQL supports creating partitions after creating the tables)

ORDERS_TABLE_PARTITION_QUERY = ("ALTER TABLE Orders "
                                "PARTITION BY LIST COLUMNS(OrderType) ("
                                "PARTITION buy VALUES IN ('buy'),"
                                "PARTITION sell VALUES IN ('sell'))");
pc.horizontal_partitioning(conn, ORDERS_TABLE_PARTITION_QUERY)

STOCK_PRICE_HISTORICAL_DATA_TABLE_PARTITION_QUERY = ("ALTER TABLE StockPriceHistory "
                                                      "PARTITION BY LIST COLUMNS(StockSymbol) ("
                                                      "PARTITION aapl VALUES IN ('AAPL'),"
                                                      "PARTITION googl VALUES IN ('GOOGL'),"
                                                      "PARTITION msft VALUES IN ('MSFT'),"
                                                      "PARTITION tcs VALUES IN ('TCS'))");
pc.horizontal_partitioning(conn, STOCK_PRICE_HISTORICAL_DATA_TABLE_PARTITION_QUERY)

```

Here, the horizontal\_partitioning function and the vertical\_partitioning function is called and the query has been passed to create the partition. Vertical partitioning typically involves splitting columns into separate tables based on certain criteria such as access patterns, sensitivity, or usage frequency. For instance, if certain columns are accessed infrequently or contain sensitive information, they might be separated into a different physical storage location or table for better security or performance reasons. The function pc.vertical\_partitioning(conn, USERS\_SENSITIVE\_TABLE\_QUERY) itself isn't a standard MySQL function or command. It is a custom function or method designed to perform vertical partitioning on the UsersSensitive table using the provided query. The code segments involve the usage of SQL statements to perform horizontal partitioning on specific tables within a database. Horizontal partitioning involves splitting a table's rows into multiple partitions (segments) based on defined criteria. The functions pc.horizontal\_partitioning(conn, ORDERS\_TABLE\_PARTITION\_QUERY) and pc.horizontal\_partitioning(conn, STOCK\_PRICE\_HISTORICAL\_DATA\_TABLE\_PARTITION\_QUERY) are custom functions or methods designed to execute these SQL queries on the provided database connection (conn). These functions could handle executing the provided partitioning queries and managing the partitioning logic for the respective tables (Orders and StockPriceHistory).

**partitionCreation.py:**

```

from mysql.connector import Error

def vertical_partitioning(conn, tableQuery):
    cursor = conn.cursor()

    try:
        cursor.execute(tableQuery)
        conn.commit()
        results = cursor.fetchall()
        for row in results:
            print(row)
    except Error as e:
        print(f"Error creating table: {e}")

    finally:
        cursor.close()

```

```

def horizontal_partitioning(conn, tableQuery):
    cursor = conn.cursor()

    try:
        cursor.execute(tableQuery)
        conn.commit()
        results = cursor.fetchall()
        for row in results:
            print(row)

    except Error as e:
        print(f"Error creating horizontal partitions: {e}")

    finally:
        cursor.close()

```

These functions are invoked in the main.py file to implement both the partitioning.

## partitionDataInsertion.py

```
import pandas as pd
from mysql.connector import Error

def insert_users_partitions(conn):
    cursor = conn.cursor()
    try:
        selectAllUsers = "SELECT * FROM Users;"

        cursor.execute(selectAllUsers)
        usersData = cursor.fetchall()

        insertBasicQuery = ("INSERT INTO UsersBasic (UserID, UserName, FirstName, LastName, Email)"
                             " VALUES (%s, %s, %s, %s, %s);")

        insertSensitiveQuery = ("INSERT INTO UsersSensitive ("
                                "UserID, Password, PhoneNumber, Address, RegistrationDate, Region, LastLogin)"
                                " VALUES (%s, %s, %s, %s, %s, %s, %s);")

        # Insert data into partitioned tables
        for user in usersData:
            cursor.execute(insertBasicQuery, (user[0], user[1], user[2], user[3], user[4]))
            cursor.execute(insertSensitiveQuery, (user[0], user[5], user[6], user[7], user[8], user[9], user[10]))
            conn.commit()

        print("Data inserted in Users successfully!")

    except Error as e:
        print(f"Error while inserting data in Users: {e}")

    finally:
        cursor.close()
```

The `insert_users_partitions` function orchestrates the insertion of data from the `Users` table into two distinct partitioned tables, namely `UsersBasic` and `UsersSensitive`. Leveraging a provided database connection (`conn`), the function initially retrieves all user data using a `SELECT` query from the original `Users` table. Subsequently, it performs `INSERT` operations iteratively for each user record fetched. It separates the user attributes into two different sets based on sensitivity: basic user information (`UserID`, `UserName`, `FirstName`, `LastName`, `Email`) is inserted into the `UsersBasic` partitioned table, while more sensitive data (`UserID`, `Password`, `PhoneNumber`, `Address`, `RegistrationDate`, `Region`, `LastLogin`) is inserted into the `UsersSensitive` partitioned table. These operations are carried out using respective `INSERT` queries tailored for each partitioned table. Throughout the process, the function ensures transactional integrity by committing changes after each user's data insertion. In case of any errors encountered during the process, it captures and logs the details but ensures the closure of the database cursor upon completion. Ultimately, the function serves to distribute and organize user data across appropriate partitioned tables based on their sensitivity levels, enhancing data management and access control within the database system.

## partitionDataRetrieval.py

```
def vertical_partition_data(conn, table):
    cursor = conn.cursor()

    try:
        selectQuery = f"SELECT * FROM {table} LIMIT 10;" # Limiting search result to 10 for display purposes.
        cursor.execute(selectQuery)
        records = cursor.fetchall()
        print(f"{table} table data:")
        for record in records:
            print(record)
        print()

    except Error as e:
        print(f"Error retrieving data from partitioned table {table} table: {e}")

    finally:
        cursor.close()

def horizontal_partition_data(conn, table, partition):
    cursor = conn.cursor()

    try:
        selectQuery = f"SELECT * FROM {table} PARTITION ({partition}) LIMIT 10;" # Limiting search result to 10 for display purposes.
        cursor.execute(selectQuery)
        records = cursor.fetchall()
        print(f"{table} table data:")
        for record in records:
            print(record)
        print()

    except Error as e:
        print(f"Error retrieving data from horizontal partitioned {table} table: {e}")

    finally:
        cursor.close()
```

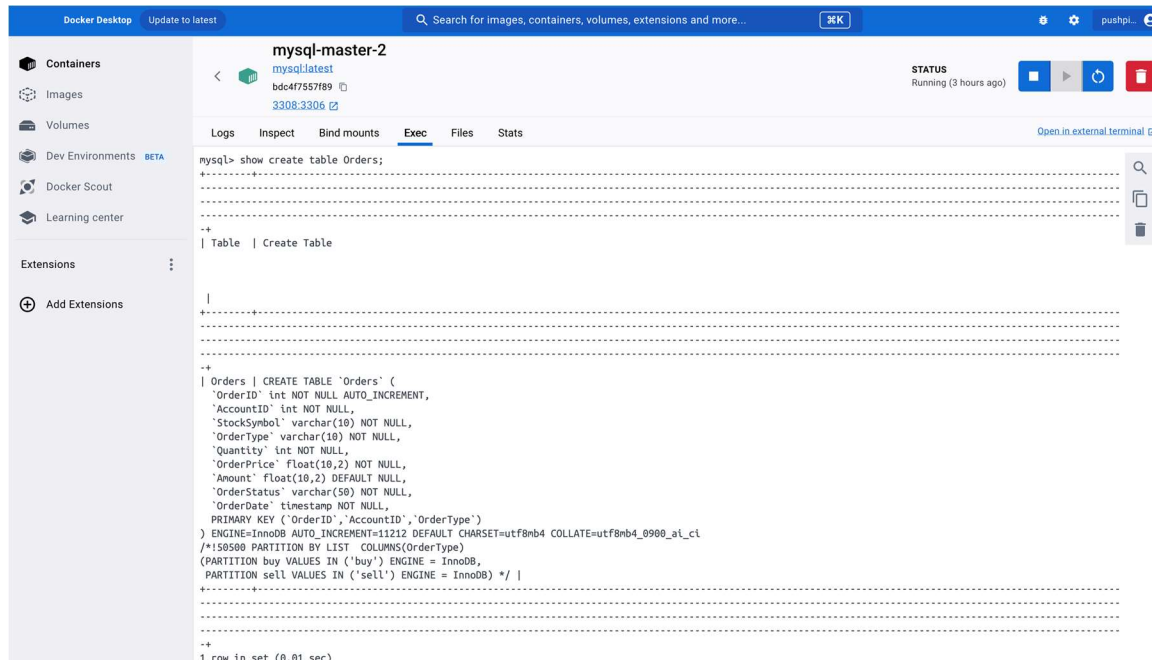
Partition status in mysql database –

StockPriceHistory Table:

The screenshot shows the Docker Desktop interface for a container named 'mysql-master-1'. The 'Exec' tab is selected, showing a terminal window with the command 'mysql> show create table StockPriceHistory;'. The output displays the table's structure and partitioning details.

```
mysql> show create table StockPriceHistory;
+-----+-----+
| Table | Create Table |
+-----+-----+
| StockPriceHistory | CREATE TABLE `StockPriceHistory` (
  `HlstoryID` int NOT NULL AUTO_INCREMENT,
  `StockSymbol` varchar(10) NOT NULL,
  `Price` float(10,2) DEFAULT NULL,
  `RecordedDateTime` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`HlstoryID`,`StockSymbol`)
) ENGINE=InnoDB AUTO_INCREMENT=864121 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
/*150500 PARTITION BY LIST COLUMNS(StockSymbol)
(PARTITION aapl VALUES IN ('AAPL') ENGINE = InnoDB,
PARTITION googl VALUES IN ('GOOGL') ENGINE = InnoDB,
PARTITION msft VALUES IN ('MSFT') ENGINE = InnoDB,
PARTITION tcs VALUES IN ('TCS') ENGINE = InnoDB) */
+-----+-----+
1 row in set (0.01 sec)
```

## OrdersTable:



The screenshot shows the Docker Desktop interface with a container named 'mysql-master-2' running. The terminal window displays the output of the 'show create table Orders;' command, showing the table structure and its partitioning details.

```
mysql> show create table Orders;
+-----+
| Table | Create Table
+-----+
| Orders | CREATE TABLE `Orders` (
  `OrderID` int NOT NULL AUTO_INCREMENT,
  `AccountID` int NOT NULL,
  `StockSymbol` varchar(10) NOT NULL,
  `OrderType` varchar(10) NOT NULL,
  `Quantity` int NOT NULL,
  `OrderPrice` float(10,2) NOT NULL,
  `Amount` float(10,2) DEFAULT NULL,
  `OrderStatus` varchar(50) NOT NULL,
  `OrderDate` timestamp NOT NULL,
  PRIMARY KEY (`OrderID`,`AccountID`,`OrderType`)
) ENGINE=InnoDB AUTO_INCREMENT=11212 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
/*150500 PARTITION BY LIST COLUMNS(OrderType)
(PARTITION buy VALUES IN ('buy') ENGINE = InnoDB,
 PARTITION sell VALUES IN ('sell') ENGINE = InnoDB) */ |
+-----+
1 row in set (0.01 sec)
```

## Data retrieval from Users vertical partition – UsersBasic:

```
UsersBasic table data:
(1, 'darrellrussell9', 'Darrell', 'Russell', 'darrell.russell@yahoo.com')
(2, 'briancruz50', 'Brian', 'Cruz', 'brian.cruz@gmail.com')
(3, 'debbiehunt81', 'Debbie', 'Hunt', 'debbie.hunt@yahoo.com')
(4, 'veronicagray88', 'Veronica', 'Gray', 'veronica.gray@hotmail.com')
(5, 'christopherjohnson69', 'Christopher', 'Johnson', 'christopher.johnson@gmail.com')
(6, 'deborahfernandez46', 'Deborah', 'Fernandez', 'deborah.fernandez@hotmail.com')
(7, 'kaylapratt93', 'Kayla', 'Pratt', 'kayla.pratt@hotmail.com')
(8, 'dianejohnson13', 'Diane', 'Johnson', 'diane.johnson@gmail.com')
(9, 'emilyramirez5', 'Emily', 'Ramirez', 'emily.ramirez@yahoo.com')
(10, 'lisagreene40', 'Lisa', 'Greene', 'lisa.greene@hotmail.com')
```

## Data retrieval from Orders horizontal partition – sell:

```
Orders table data:
(2, 1, 'G006L', 'sell', 8, 339.67, 2717.36, 'fulfilled', datetime.datetime(2023, 11, 6, 14, 38, 4))
(4, 1, 'G006L', 'sell', 53, 205.92, 10913.76, 'fulfilled', datetime.datetime(2023, 11, 7, 13, 44, 48))
(6, 1, 'TCS', 'sell', 33, 218.07, 7196.31, 'fulfilled', datetime.datetime(2023, 11, 15, 12, 53, 44))
(9, 1, 'MSFT', 'sell', 32, 13.35, 427.2, 'fulfilled', datetime.datetime(2023, 11, 13, 14, 17, 9))
(11, 2, 'AAPL', 'sell', 27, 122.01, 3294.27, 'fulfilled', datetime.datetime(2023, 11, 6, 10, 20, 48))
(13, 2, 'TCS', 'sell', 57, 82.29, 4690.53, 'fulfilled', datetime.datetime(2023, 11, 6, 12, 8, 35))
(17, 3, 'AAPL', 'sell', 31, 101.24, 3138.44, 'fulfilled', datetime.datetime(2023, 11, 10, 14, 39, 10))
(21, 4, 'MSFT', 'sell', 22, 17.05, 375.1, 'fulfilled', datetime.datetime(2023, 11, 8, 13, 35, 12))
(25, 5, 'AAPL', 'sell', 25, 107.79, 2694.75, 'fulfilled', datetime.datetime(2023, 11, 9, 10, 43, 53))
(27, 5, 'MSFT', 'sell', 26, 49.26, 1280.76, 'fulfilled', datetime.datetime(2023, 11, 15, 10, 28, 30))
```



Data retrieval from Orders horizontal partition – googl:

```
StockPriceHistory table data:
(2, 'GOOGL', 219.82, datetime.datetime(2023, 11, 6, 9, 0))
(6, 'GOOGL', 221.34, datetime.datetime(2023, 11, 6, 9, 0, 1))
(10, 'GOOGL', 223.39, datetime.datetime(2023, 11, 6, 9, 0, 2))
(14, 'GOOGL', 224.12, datetime.datetime(2023, 11, 6, 9, 0, 3))
(18, 'GOOGL', 223.34, datetime.datetime(2023, 11, 6, 9, 0, 4))
(22, 'GOOGL', 227.3, datetime.datetime(2023, 11, 6, 9, 0, 5))
(26, 'GOOGL', 230.42, datetime.datetime(2023, 11, 6, 9, 0, 6))
(30, 'GOOGL', 232.46, datetime.datetime(2023, 11, 6, 9, 0, 7))
(34, 'GOOGL', 236.7, datetime.datetime(2023, 11, 6, 9, 0, 8))
(38, 'GOOGL', 240.48, datetime.datetime(2023, 11, 6, 9, 0, 9))
```

## Replication:

In MySQL distributed databases, replication is a mechanism used to create and maintain copies of data across multiple database servers. It's a key feature for improving fault tolerance, scalability, and data availability. MySQL offers various replication methods to synchronize data among nodes in a distributed environment:

Types of Replications in MySQL Distributed Databases:

### Master-Slave Replication:

- Involves a single master database server and one or more slave servers.
- The master server logs changes to its data, and these changes are propagated to the slave servers, keeping them synchronized.
- Slaves can be used for read operations, backups, analytics, etc., without impacting the master's performance.
- Asynchronous replication: Transactions committed on the master are replicated to slaves asynchronously.

### Master-Master Replication:

- Allows multiple servers to act as both master and slave simultaneously.
- Any change made on one master is replicated to other masters, ensuring that all nodes have the same data.
- Conflict resolution mechanisms are crucial to manage conflicting writes on different masters.

### Group Replication:

- Forms a group of servers that work together in a fault-tolerant and distributed way.
- All members of the group can accept reads and writes (similar to multi-master replication).
- It ensures consistency by replicating transactions across the group and handling failover automatically.

In our project, we've set up a master-slave replication system. The code snippet below demonstrates the configuration for replicating data, using the identifiers 'mysql-master-1' for the master and 'mysql-master-2' for the slave.

## docker-compose.yml

```
version: '3'

services:
  db1:
    image: mysql:latest
    container_name: mysql-master-1
    restart: unless-stopped
    tty: true
    ports:
      - "3307:3306"
    expose:
      - '3306'
    environment:
      MYSQL_DATABASE: pvs_stock_trading
      MYSQL_USER: master
      MYSQL_PASSWORD: master
      MYSQL_ROOT_PASSWORD: root
      SERVICE_TAGS: dev
      SERVICE_NAME: mysql

  db2:
    image: mysql:latest
    container_name: mysql-master-2
    restart: unless-stopped
    tty: true
    ports:
      - "3308:3306"
    expose:
      - '3306'
    environment:
      MYSQL_DATABASE: pvs_stock_trading
      MYSQL_USER: replica
      MYSQL_PASSWORD: replica
      MYSQL_ROOT_PASSWORD: root
      SERVICE_TAGS: dev
      SERVICE_NAME: mysql
```

This code snippet represents a Docker Compose configuration file (`docker-compose.yml`) used to set up two MySQL database instances (`db1` and `db2`) as part of a master-slave replication system.

Description:

`db1` (mysql-master-1):

- Represents the master MySQL database instance (`mysql-master-1`).
- Configuration includes specifying the MySQL version via the `image` tag (`mysql:latest`).



- Defines port mappings where MySQL runs on port `3306` inside the container and is exposed externally on port `3307`.
- Sets environment variables for the master database, such as database name, user (`master`), and passwords for root and the specified user.

`db2` (mysql-master-2):

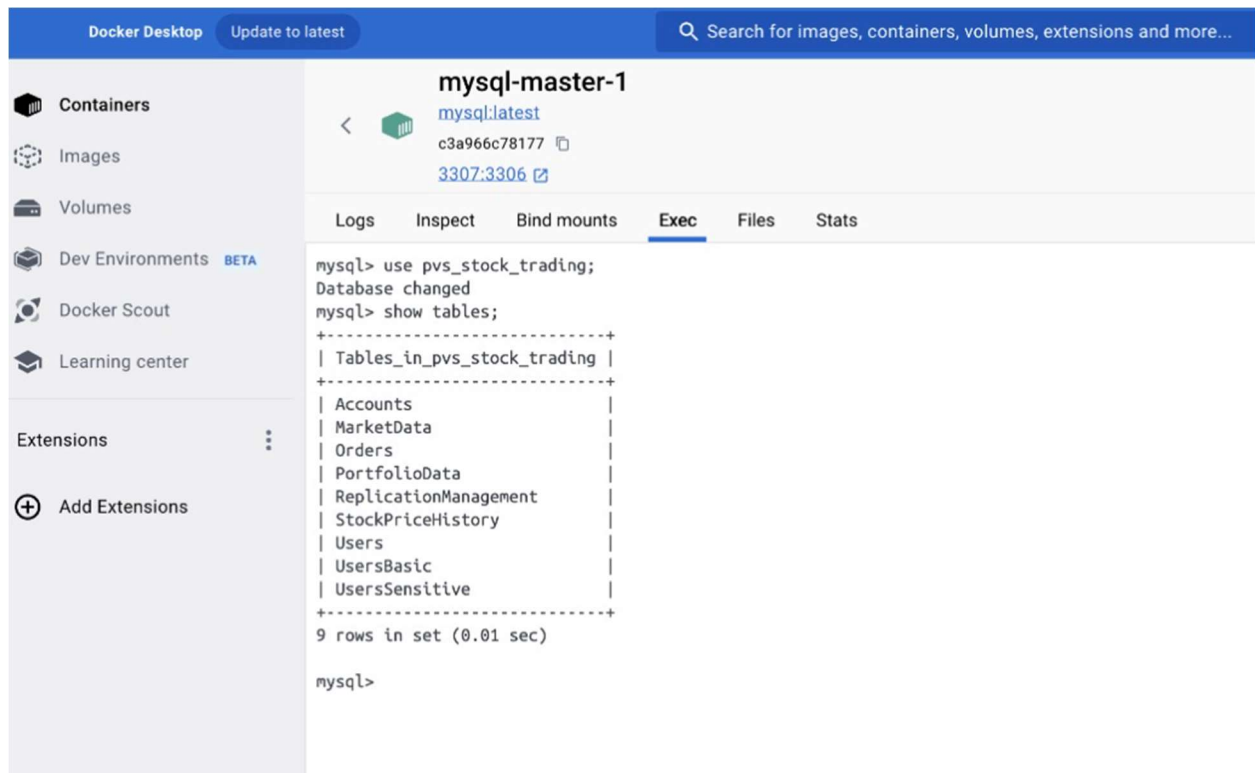
- Represents the slave MySQL database instance (`mysql-master-2`).
- Configuration mirrors `db1` but uses a different port (`3308:3306`) to avoid port conflicts.
- Environment variables are set for the slave database, including the database name (`pvs\_stock\_trading`), user (`replica`), and corresponding passwords.

How it Works:

- The `image: mysql:latest` indicates the use of the latest MySQL Docker image for both instances (`db1` and `db2`).
- Port mappings (`3307:3306` and `3308:3306`) enable external access to the databases on different ports.
- Each instance is configured with environment variables (`MYSQL\_DATABASE`, `MYSQL\_USER`, `MYSQL\_PASSWORD`, `MYSQL\_ROOT\_PASSWORD`) necessary for initializing and accessing the databases.

This setup facilitates the creation of two separate MySQL instances using Docker, one acting as the master (`db1`) and the other as the slave (`db2`), forming a basic master-slave replication setup.

Master-1:



Master-2:

Docker Desktop

Update to latest

Search for images, containers, volumes, extensions and more...

Containers

Images

Volumes

Dev Environments BETA

Docker Scout

Learning center

Extensions

Add Extensions

mysql-master-2

mysql:latest

bdc4f7557f89

3308:3306

Logs

Inspect

Bind mounts

Exec

Files

Stats

mysql> use pvs\_stock\_trading;

Database changed

mysql> show tables;

+-----+-----+

| Tables\_in\_pvs\_stock\_trading |

+-----+-----+

| Accounts |

| MarketData |

| Orders |

| PortfolioData |

| ReplicationManagement |

| StockPriceHistory |

| Users |

| UsersBasic |

| UsersSensitive |

+-----+-----+

9 rows in set (0.01 sec)

mysql>