

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

Course Title: Distributed Database Systems (CSE 512)

Team: PVS

Part 4: Distributed Transaction Management

Problem Statement: Implement distributed transaction management to ensure data consistency and concurrency control in your chosen topic.

Tasks completed: ACID-compliant Implementation, Implementation on concurrency control.

In distributed databases, ensuring the ACID (Atomicity, Consistency, Isolation, Durability) properties, managing concurrency control, and coordinating transactions across distributed nodes are essential but more complex compared to centralized databases due to the distributed nature of the environment.

ACID Properties in Distributed Databases:

1. Atomicity: Ensures that transactions are either completed in full or not at all, even in a distributed setting. Achieving atomicity involves coordination mechanisms to guarantee that all operations in a transaction are executed successfully across multiple nodes or databases or are entirely rolled back in case of a failure.
2. Consistency: Maintains data consistency across distributed nodes after a transaction. Distributed systems use techniques like consensus algorithms or distributed locking to ensure data consistency despite concurrent operations happening in various locations.
3. Isolation: Ensures that concurrent transactions do not interfere with each other, providing each transaction with the illusion of executing in isolation. Isolation mechanisms, such as multi-version concurrency control (MVCC) or distributed locking, help manage concurrent access to data across distributed nodes.
4. Durability: Guarantees that once a transaction is committed, its effects persist even in the event of system failures. Replication, logging, and distributed commit protocols are used to achieve durability in distributed environments.

Concurrency Control in Distributed Databases:

- **Distributed Locking:** Mechanisms like distributed locks or timestamps are used to manage concurrent access to shared resources across distributed nodes. Techniques like two-phase locking or optimistic concurrency control help maintain consistency and prevent conflicts.

- Multi-Version Concurrency Control (MVCC): Allows for concurrent reads and writes by maintaining multiple versions of data. Each transaction gets a consistent snapshot of data, reducing contention and enhancing performance.
- Conflict Resolution: Strategies to resolve conflicts arising from concurrent updates, such as using timestamps or prioritizing transactions, are vital in distributed systems.

Distributed Coordination for Transactions:

- Two-Phase Commit (2PC): Coordinating transactions across distributed nodes involves protocols like 2PC, ensuring that all nodes either commit or abort a transaction. However, 2PC can suffer from blocking issues and isn't always suitable for large-scale distributed systems due to its blocking nature.
- Distributed Consensus Algorithms: Algorithms like Paxos, Raft, or variants of the Byzantine Fault Tolerance (BFT) consensus protocols are used for distributed coordination and agreement among nodes to ensure consistency in the presence of failures or network partitions.
- Transactional Models: Newer transactional models like Spanner's TrueTime or database systems employing distributed ledger technologies utilize innovative approaches for distributed coordination, offering strong consistency guarantees across distributed nodes.

Managing ACID properties, concurrency control, and distributed coordination in distributed databases requires sophisticated protocols, consensus algorithms, and transactional models to ensure data integrity, consistency, and reliability across distributed environments despite the challenges of network latency, node failures, and partition tolerance.

In our project, we have implemented ACID and concurrency control to manage simultaneous stock transactions, preventing conflicts and maintaining consistency across distributed nodes.

Transaction control for buying and selling of stocks:

The transaction will be rolled back if any part of the transaction fails. This ensures data integrity and consistency across the database.

```

19  try:
20      conn.start_transaction()
21      selectQuery = ("SELECT CurrentPrice from MarketData where StockSymbol = '{stockSymbol}';")
22      cursor.execute(selectQuery)
23      results = cursor.fetchall()
24
25      orderPrice = results[0][0]
26      amount = quantity * orderPrice
27      orderId = 11211
28
29      createBuyOrder = ("INSERT INTO Orders (orderId, AccountID, StockSymbol, OrderType, Quantity, OrderPrice, Amount,
30                          'OrderStatus, OrderDate')
31                          VALUES (%s, %s, %s, 'buy', %s, %s, %s, 'placed', NOW());")
32      cursor.execute(createBuyOrder, [orderId, accountId, stockSymbol, quantity, orderPrice, amount])
33      res = cursor.fetchall()
34      conn.commit()
35
36  conn.start_transaction()
37  checkSellOrderQuery = ("SELECT * FROM Orders where OrderType = 'sell' and OrderStatus = "
38                          "'placed' and StockSymbol = %s and Quantity = %s order by OrderId;")
39  cursor.execute(checkSellOrderQuery, [stockSymbol, quantity])
40  res = cursor.fetchall()
41  for r in res:
42      print(res)
43      print(res[0][1], res[0][4])
44      sellAccountID = res[0][1]
45      sellOrderId = res[0][0]
46  if res:
47      updateSellPortfolioQuery = ("UPDATE PortfolioData SET Quantity = Quantity - %s WHERE PortfolioID = (SELECT "
48                                  "'PortfolioID from Accounts WHERE AccountID = %s) and StockSymbol = %s;")
49      cursor.execute(updateSellPortfolioQuery, [quantity, sellAccountID, stockSymbol])
50
51      updateSellOrder = ("UPDATE Orders SET OrderStatus = 'fulfilled' WHERE OrderId = %s;"
52                          cursor.execute(updateSellOrder, [sellOrderId])
53
54      updateBuyPortfolioQuery = ("UPDATE PortfolioData SET Quantity = Quantity + %s WHERE PortfolioID = (SELECT "
55                                  "'PortfolioID from Accounts WHERE AccountID = %s) and StockSymbol = %s;")
56      cursor.execute(updateBuyPortfolioQuery, [quantity, accountId, stockSymbol])
57
58      updateBuyOrder = ("UPDATE Orders SET OrderStatus = 'fulfilled' WHERE OrderId = %s;"
59                          cursor.execute(updateBuyOrder, [orderId])

```

Stock Price Stimulator:

This is a stock price stimulator in which the stock prices of all the stocks in the market are updated by a percentage of their last value and inserts them into the database. This shows the real time capabilities of the database to handle continuous insertion of data in the database.

A record for each stock is inserted in the database every second.

```

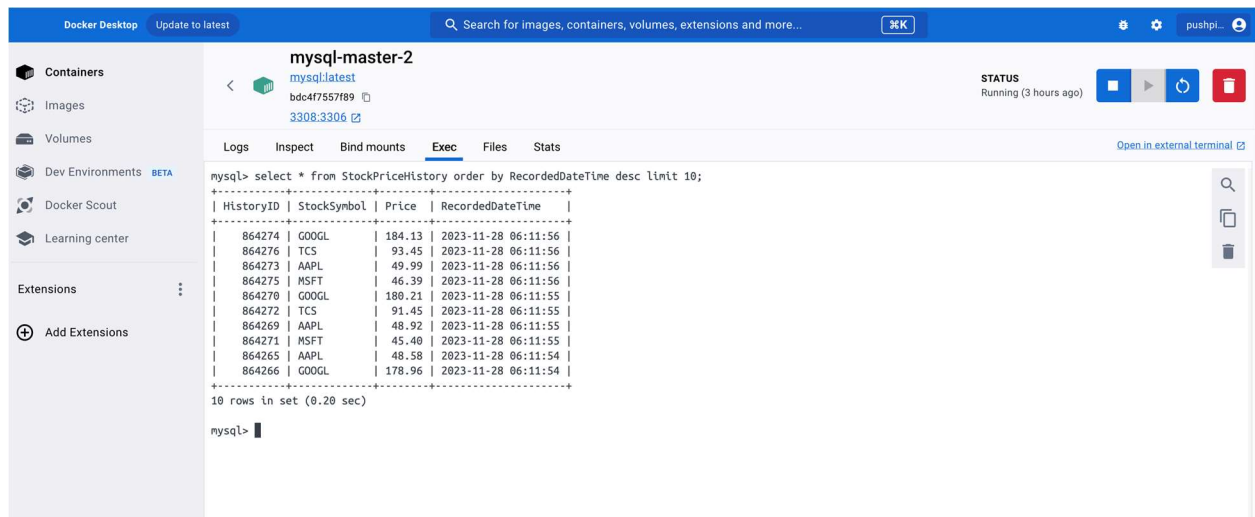
sell.py stockPriceStimulator.py
27 records = cursor.fetchall()
28 print("MarketData table data:")
29 for record in records:
30     print(record)
31 print()
32
33 aapl = records[0][2]
34 googl = records[1][2]
35 msft = records[2][2]
36 tcs = records[3][2]
37 percentage_range = (0.2, 3)
38 while(True):
39     percentage_change = random.uniform(*percentage_range)
40
41     # Randomly decide to increase or decrease
42     if random.choice([True, False]):
43         # Increase the value
44         aapl = aapl * (1 + percentage_change / 100)
45         googl = googl * (1 + percentage_change / 100)
46         msft = msft * (1 + percentage_change / 100)
47         tcs = tcs * (1 + percentage_change / 100)
48     else:
49         # Decrease the value
50         aapl = aapl * (1 - percentage_change / 100)
51         googl = googl * (1 - percentage_change / 100)
52         msft = msft * (1 - percentage_change / 100)
53         tcs = tcs * (1 - percentage_change / 100)
54
55 try:
56     conn.start_transaction()
57     insertStockPrice = ("INSERT INTO StockPriceHistory (StockSymbol,Price,RecordedDateTime) "
58                          "VALUES"
59                          "('AAPL', %s, NOW()),"
60                          "('GOOGL', %s, NOW()),"
61                          "('MSFT', %s, NOW()),"
62                          "('TCS', %s, NOW());")
63
64     cursor.execute(insertStockPrice, [aapl, googl, msft, tcs])
65     conn.commit()
66     time.sleep(1)
67     print("Stock price updated")
68

```

Output:

[illegible]

The data which is inserted by the stimulator:



The screenshot shows the Docker Desktop interface. On the left is a sidebar with navigation options: Containers, Images, Volumes, Dev Environments (BETA), Docker Scout, Learning center, Extensions, and Add Extensions. The main area displays details for a container named 'mysql-master-2', which is based on the 'mysql:latest' image and has ID 'bdc4f7557f89'. The container is running, with a status of 'Running (3 hours ago)'. Below the container details, there are tabs for Logs, Inspect, Bind mounts, Exec, Files, and Stats. The 'Exec' tab is active, showing a terminal window with a MySQL prompt. The terminal displays a SQL query: 'select * from StockPriceHistory order by RecordedDateTime desc limit 10;'. The results are shown in a table with columns: HistoryID, StockSymbol, Price, and RecordedDateTime. The table contains 10 rows of data, sorted by RecordedDateTime in descending order. The terminal also shows '10 rows in set (0.20 sec)' and a MySQL prompt 'mysql>'.

HistoryID	StockSymbol	Price	RecordedDateTime
864274	GOOGL	184.13	2023-11-28 06:11:56
864276	TCS	93.45	2023-11-28 06:11:56
864273	AAPL	49.99	2023-11-28 06:11:56
864275	MSFT	46.39	2023-11-28 06:11:56
864270	GOOGL	180.21	2023-11-28 06:11:55
864272	TCS	91.45	2023-11-28 06:11:55
864269	AAPL	48.92	2023-11-28 06:11:55
864271	MSFT	45.40	2023-11-28 06:11:55
864265	AAPL	48.58	2023-11-28 06:11:54
864266	GOOGL	178.96	2023-11-28 06:11:54