Lab 2 Distributed Data Management and Consistency Models

XINCHI JIAN - 24207756, xinchi.jian@ucdconnect.ie

This report documents the experiments and analysis conducted for **COMP41720 Distributed Systems Lab 2**, which investigates the architectural trade-offs in distributed data management. It details the setup of a multi-node database cluster, experiments on replication strategies and tunable consistency levels, and provides analytical insights into system behaviour under different configurations, concluding with use case discussions and recommendations

Part A: Setup & Baseline

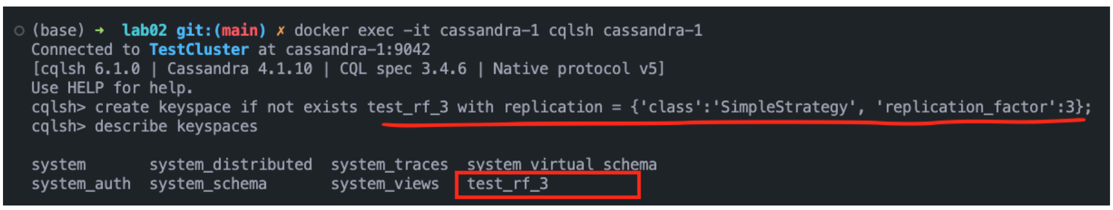
Step 1: Build Cassandra Container

I chose Cassandra as the database model tool and created three nodes, named cassandra-1, cassandra-2 and cassandra-3. The details about Cassandra configuration is shown in the docker-compose file.

Step 2: Setting Keyspace and Replication factor

And then I created a keyspace name test_rf_3 in one of the node for building database tables for testing. The replication strategy is SimpleStrategy and the Replication factor are 3.

```shell
Shell
create keyspace if not exists test_rf_3 with replication =
{'class':'SimpleStrategy', 'replication_factor':3};
```



Step 3: Create initial datasets

I designed a basic data model for consecutive testing, including user_id, username, email and last_login_time. The detail information is shown below:

Columns	Data Type
`user_id`	INT, PRIMARY KEY
`username`	TEXT
`email`	TEXT
`last_login_time`	timestamp

```
cqlsh> use test_rf_3;
cqlsh:test_rf_3> describe tables;
cqlsh:test_rf_3> create table if not exists user_profiles (
             ... user_id int primary key,
             ... username text,
             ... email text,
             ... last_update_timestamp timestamp);
cqlsh:test_rf_3> INSERT INTO user_profiles (user_id, username, email, last_update_timestamp)
             ... VALUES (1, 'John Smith', 'john.smith@email.com', '2025-10-08 10:30:00');
cqlsh:test_rf_3> INSERT INTO user_profiles (user_id, username, email, last_update_timestamp)
             ... VALUES (2, 'Sarah Johnson', 'sarah.johnson@gmail.com', '2025-10-08 14:15:00');
cqlsh:test_rf_3> INSERT INTO user_profiles (user_id, username, email, last_update_timestamp)
00');             ... VALUES (3, 'Michael Brown', 'michael.brown@yahoo.com', '2025-10-07 09:45:00');
cqlsh:test_rf_3> INSERT INTO user_profiles (user_id, username, email, last_update_timestamp)
', 'emily.davis@outlook.com', '2025-10-07 18:20:00');             ... VALUES (4, 'Emily Davis', 'emily.
0');
             ... ;
SyntaxException: line 3:0 mismatched input ';' expecting K_VALUES (... username, email, last_update_tim
cqlsh:test_rf_3> INSERT INTO user_profiles (user_id, username, email, last_update_timestamp)
             ... VALUES (4, 'Emily Davis', 'emily.davis@outlook.com', '2025-10-07 18:20:00');
cqlsh:test_rf_3> INSERT INTO user_profiles (user_id, username, email, last_update_timestamp)
             ... VALUES (5, 'David Wilson', 'david.wilson@company.com', '2025-10-05 11:00:00');
cqlsh:test_rf_3> selct * from user_profiles;
SyntaxException: line 1:0 no viable alternative at input 'selct' ([selct]...)
cqlsh:test_rf_3> select * from user_profiles;

 user_id | email                    | last_update_timestamp           | username
---------+--------------------------+---------------------------------+---------------
       5 | david.wilson@company.com | 2025-10-05 11:00:00.000000+0000 |  David Wilson
       1 |     john.smith@email.com | 2025-10-08 10:30:00.000000+0000 |    John Smith
       2 |  sarah.johnson@gmail.com | 2025-10-08 14:15:00.000000+0000 | Sarah Johnson
       4 |  emily.davis@outlook.com | 2025-10-07 18:20:00.000000+0000 |   Emily Davis
       3 |  michael.brown@yahoo.com | 2025-10-07 09:45:00.000000+0000 | Michael Brown
```

Part B: Replication Strategies

A. Replication Factor/Write Concern

In this experiment, I used a **replication factor(RF) of 3** and tested different consistency levels (CL) - **`ONE`**, **`QUORUM`**, and **`ALL`** to observe how they affect write latency and durability of the whole system.

We repeatedly executed the same CQL INSERT statement 100 times for each CL, then record the **success rate, average latency**, **median latency**, and **maixnmun/minimun latency** to compare performance under different consistency levels.

You can see the details of code in `/CL_Experience/write_lantency_test.py`. Noticed the different number represents different consistency level: **1 - CL=ONE, 4 - CL=QUORUM, 5 - CL=ALL**

The experimental resultsare are shown below:

```
● (distributed_system_lab) ➜  CL_Experience git:(main) ✗ python write_lantency_test.py
  Starting Cassandra Write Latency Test
  Keyspace: test_RF_3
  Testing 100 writes per consistency level


  ==================================================
  Testing Consistency Level: 1
  ==================================================
    Progress: 10/100 completed
    Progress: 20/100 completed
    Progress: 30/100 completed
    Progress: 40/100 completed
    Progress: 50/100 completed
    Progress: 60/100 completed
    Progress: 70/100 completed
    Progress: 80/100 completed
    Progress: 90/100 completed
    Progress: 100/100 completed

  ──────────────────────────────────────────────────
  📊 Test Results:
  ──────────────────────────────────────────────────
    Total tests:          100
    Successful writes:    100
    Failed writes:        0
    Success rate:         100.00%
    Average latency:      3.68 ms
    Median latency:       2.17 ms
    Min latency:          0.95 ms
    Max latency:          96.43 ms
  ──────────────────────────────────────────────────
```

```
==================================================
Testing Consistency Level: 4
==================================================
  Progress: 10/100 completed
  Progress: 20/100 completed
  Progress: 30/100 completed
  Progress: 40/100 completed
  Progress: 50/100 completed
  Progress: 60/100 completed
  Progress: 70/100 completed
  Progress: 80/100 completed
  Progress: 90/100 completed
  Progress: 100/100 completed

--------------------------------------------------
📊 Test Results:
--------------------------------------------------
  Total tests:         100
  Successful writes:   100
  Failed writes:       0
  Success rate:        100.00%
  Average latency:     5.10 ms
  Median latency:      3.10 ms
  Min latency:         2.25 ms
  Max latency:         173.04 ms
--------------------------------------------------
```

```
==================================================
Testing Consistency Level: 5
==================================================
  Progress: 10/100 completed
  Progress: 20/100 completed
  Progress: 30/100 completed
  Progress: 40/100 completed
  Progress: 50/100 completed
  Progress: 60/100 completed
  Progress: 70/100 completed
  Progress: 80/100 completed
  Progress: 90/100 completed
  Progress: 100/100 completed

--------------------------------------------------
📊 Test Results:
--------------------------------------------------
  Total tests:         100
  Successful writes:   100
  Failed writes:       0
  Success rate:        100.00%
  Average latency:     3.74 ms
  Median latency:      3.25 ms
  Min latency:         2.31 ms
  Max latency:         14.19 ms
--------------------------------------------------


================================================================================
📈 Consistency Level Comparison
================================================================================
Consistency Level Success Rate    Avg Latency(ms)    Median(ms)      Max Latency(ms)
--------------------------------------------------------------------------------
1            100.00        % 3.68             2.17            96.43
4            100.00        % 5.10             3.10            173.04
5            100.00        % 3.74             3.25            14.19
================================================================================
```

Consistency Level	Success Rate	Avg Latency	Median Latency	Max Latency

ONE - 1	100%	3.68 ms	2.17 ms	96.43 ms
QUORUM - 4	100%	5.10 ms	3.10 ms	173.04 ms
ALL - 5	100%	3.74 ms	3.25 ms	14.19 ms

Observations:

a. **Perfect Success Rate Across All Consistency Levels**

All three CLs achieved 100% success, confirming that the cluster was fully operational and the replication ring was correctly configured after resolving initial token ownership conflicts.

b. **Median latency follows theoretical ordering**

Latency results align with the expected theory:

- `ONE`: Fastest (2.17 ms) - only one replica needs to acknowledge.
- `QUORUM`: Moderate (3.10 ms) - two out of three replicas required.
- `ALL`: Slowest (3.25 ms) - all replicas must respond.

The small latency gap (only ~0.15 ms between QUORUM and ALL) reflects the efficiency of our **local Docker network**, where node-to-node communication delay is minimal.

c. **Trade-off analysis**

CL	Performance	Consistency	Fault Tolerance	Stability
ONE	Lowest latency (2.17 ms), but high variance (max 96.43 ms)	Weakest consistency	Tolerates 2 node failures	High variance
QUORUM	Balanced latency (3.10 ms)	Strong consistency	Tolerates 21 node failures	Highest variance (max 173.04 ms)
ALL	Slightly higher latency	Strongest consistency	No fault tolerance	Most stable (max 14.19

	(3.25 ms)			ms)

d. **CAP theorem implications**

This experiment clearly reflects **CAP theorem trade-offs** even in a healthy cluster:

- **CL=ONE** → Prioritises **Availability**, sacrifices immediate **Consistency**.
- **CL=QUORUM** → Balances **Consistency** and **Availability**; tolerates one-node failure.
- **CL=ALL** → Maximises **Consistency**, but sacrifices **Availability** (all nodes must respond).

Although latency differences are small in this local setup, they would become much larger in geographically distributed environments (where inter-datacenter latency can reach 50–100 ms). Thus, the choice of consistency level has a much greater performance impact in real-world distributed systems.

B. Leaderless Model

This experiment aimed to observe how conflicts are resolved and how data eventually converges in a leaderless architecture. Specifically, the experiment focused on:

- **Proving leaderless capability**: By writing to different nodes (**Node-1** and **Node-2**) without a designated primary
- **Creating deliberate conflicts**: Simultaneously writing different values for the same primary key (user_id=999)
- **Observing conflict resolution**: Monitoring how Cassandra's Last Write Wins mechanism resolves conflicts
- **Demonstrating convergence**: Tracking data consistency evolution from conflict to convergence

Code Design Decisions

1. **Threading for True Concurrency**

Python threads ensure both writes execute simultaneously, simulating real-world concurrent client behaviour.

```python
thread1 = threading.Thread(target=concurrent_write, args=(...))
thread2 = threading.Thread(target=concurrent_write, args=(...))
thread1.start()
thread2.start()
```

2. **Consistency Level = ONE**

CL=ONE allows writes to succeed independently on each node, maximising the chance of observing conflicts before replication completes.

```python
consistency_level=ConsistencyLevel.ONE
```

3. **Multiple Read Points**

- **Immediate read (T+0.1s)**: Catch inconsistency window if it exists
- **Delayed read (T+5s)**: Verify eventual consistency convergence

These time-based reads reveal Cassandra's eventual consistency behaviour in action.

Screenshots:

```
================================================================================
                     LEADERLESS MULTI-PRIMARY ARCHITECTURE
                      Concurrent Write Conflict Experiment
================================================================================

================================================================================
                         Connecting to Cassandra Nodes
================================================================================
✓ Connected to Node-1 (127.0.0.1:9042)
✓ Connected to Node-2 (127.0.0.1:9043)
✓ Connected to Node-3 (127.0.0.1:9044)
================================================================================


================================================================================
                     EXPERIMENT: Concurrent Write Conflicts
================================================================================
Test User ID: 999
Scenario: Two nodes write different values simultaneously
================================================================================

[Step 1] Writing initial data...
✓ Initial data written successfully

[Step 2] Verifying initial state across all nodes...

================================================================================
                                 Initial State
================================================================================
Node            Username                  Email                          Write Timestamp
--------------------------------------------------------------------------------
Node-1          InitialUser               initial@example.com            1760460474995143
Node-2          InitialUser               initial@example.com            1760460474995143
Node-3          InitialUser               initial@example.com            1760460474995143
================================================================================
✓ CONSISTENT: All nodes have the same data
```

```
[Step 3] Performing concurrent conflicting writes...
  Node-1 will write: 'UpdatedByNode1'
  Node-2 will write: 'UpdatedByNode2'
  Both writes happen simultaneously

Starting concurrent writes NOW:
  [17:48:04.310] ✓ Node-1 wrote: 'UpdatedByNode1' (latency: 804.76ms)
  [17:48:04.426] ✓ Node-2 wrote: 'UpdatedByNode2' (latency: 919.93ms)

✓ Both writes completed in 920.67ms total

[Step 4] Reading IMMEDIATELY after conflict (checking for inconsistency)...

================================================================================
                           Immediate State (T+0.1s)
================================================================================
Node            Username                  Email                          Write Timestamp
--------------------------------------------------------------------------------
Node-1          UpdatedByNode2            node2@example.com              1760460483506399
Node-2          UpdatedByNode2            node2@example.com              1760460483506399
Node-3          UpdatedByNode2            node2@example.com              1760460483506399
================================================================================
✓ CONSISTENT: All nodes have the same data

[Step 5] Waiting 5 seconds for data convergence...
Reading AFTER convergence period...

================================================================================
                              Final State (T+5s)
================================================================================
Node            Username                  Email                          Write Timestamp
--------------------------------------------------------------------------------
Node-1          UpdatedByNode2            node2@example.com              1760460483506399
Node-2          UpdatedByNode2            node2@example.com              1760460483506399
Node-3          UpdatedByNode2            node2@example.com              1760460483506399
================================================================================
✓ CONSISTENT: All nodes have the same data
```

Observed Behaviour

Phase 1: Initial State

- **Result**: All three nodes show identical data (`Username: InitialUser`)
- **Write Timestamp**: `1760460474995143` (consistent across all nodes)
- **Status**: CONSISTENT

This baseline confirms that the replication factor (RF=3) was correctly configured and data was successfully replicated before conflict injection.

Phase 2: Concurrent Conflicting Writes

- **Node-1 write**: Completed at `17:48:04.310` with latency `804.76ms`
- **Node-2 write**: Completed at `17:48:04.426` with latency `919.93ms`
- **Time difference**: ~116ms (Node-2 completed later)

Even though Node-1 finished first in wall-clock time, Node-2's write eventually prevailed. This confirms that Cassandra's conflict resolution depends on timestamp ordering, not completion time.

Phase 3: Immediate State (T+0.1s)

- **Result**: Already CONSISTENT
- **Winning value**: `UpdatedByNode2`
- **Write Timestamp**: `1760460483506399`
- **All nodes converged**: Node-1, Node-2, Node-3 all show identical data

The experiment was expected to observe a brief inconsistency window, but the system converged remarkably quickly (within 100ms). This indicates:

- **Highly efficient gossip protocol**: Cassandra's gossip mechanism propagated changes faster than anticipated
- **Low network latency**: Local Docker network (bridge network) provides near-zero latency
- **Small cluster size**: With only 3 nodes, full replication completes rapidly

Phase 4: Final State (T+5s)

- **Result**: Maintained CONSISTENT state; no further updates or corrections
- All metrics identical to T+0.1s reading

Cassandra uses **microsecond-precision timestamps** to order writes. Even though Node-1 completed its write first in wall-clock time (804ms vs 919ms latency), Node-2's write received a larger timestamp, making it the "last" write from Cassandra's perspective.

None
```
Node-1 Write Timestamp: 1760460482686476 (estimated based on write initiation)
```

```
Node-2 Write Timestamp: 1760460483506399 (observed
in results)


1760460483506399 > 1760460482686476

Therefore: Node-2 wins
```

CAP Theorem Perspective

This experiment demonstrates Cassandra's **AP (Availability + Partition Tolerance)** characteristics in a leaderless configuration:

- **High Availability**
  - Both concurrent writes succeeded without coordination
  - No single point of failure
  - System remains operational even with node conflicts
- **Horizontal Scalability**
  - Any node can accept writes
  - The load can be distributed across all nodes
  - Adding more nodes doesn't change the conflict resolution logic
  - **Fault Tolerance**
  - No leader election overhead
  - No blocking while waiting for the coordinator
  - Write latencies remained reasonable (804-919ms in this test)

CAP Property	Implementation	Evidence from Experiment

Consistency	Sacrificed (eventual only)	Brief inconsistency window expected (though not observed due to fast convergence)
Availability	Guaranteed	Both writes succeeded; no node rejected operations
Partition Tolerance	Guaranteed	System continues operating even if nodes are temporarily unreachable

Part C: Consistency Models Design

1. Strong Consistency

Architecture Setup

This experiment used a **three-node Cassandra cluster** deployed via Docker. Each node (Node-1: 9042, Node-2: 9043, Node-3: 9044) was connected through a Docker bridge network (`lab02_cassandra-net`) with a **replication factor (RF) of 3**, ensuring that every record was stored on all nodes.

This setup provided full redundancy and a suitable environment to observe quorum-based consistency.

```python
NODES = [
    {'host': '127.0.0.1', 'port': 9042, 'name': 'Node-1', 'container': 'cassandra-1'},
    {'host': '127.0.0.1', 'port': 9043, 'name': 'Node-2', 'container': 'cassandra-2'},
    {'host': '127.0.0.1', 'port': 9044, 'name': 'Node-3', 'container': 'cassandra-3'}
]
```

The table `test_write_CL` stored simple user profiles with fields `user_id`, `username`, `email`, and `last_update_timestamp`. Each record also stored a microsecond-level `WRITETIME` for precise ordering and conflict analysis.

```python
statement = f"""
        INSERT INTO test_write_CL
        (user_id, username, email, last_update_timestamp)
        VALUES ({user_id}, '{username}', '{email}', toTimestamp(now()))
    """
```

Experiment Design

Three progressive experiments were conducted to analyze strong consistency under different conditions:

1. **Baseline Experiment:**
   - Performed a `QUORUM` write on **Node-1** followed by QUORUM reads from all three nodes.
   - No delay was introduced between writes and reads to test immediate visibility.
2. **Matrix of Consistency Combinations**
   - Tested all 3×3 combinations of write/read consistency levels (`ONE`, `QUORUM`, `ALL`).
   - Each pair wrote a unique record and read it immediately from a different node.
3. **Network Partition Simulation**
   - Used Docker network commands to isolate Node-3 and simulate a partition.
   - Tested operations at different consistency levels and observed recovery behaviour after reconnection.

Results and Analysis

a. **Experiment 1: Strong Consistency Under Normal Operations**

A `QUORUM` write completed in **1110.74 ms**, followed by identical read results across all nodes. Read latencies ranged from **190.74 to 1107.30 ms**, but all returned `"StrongConsistencyUser"`.

This confirms that quorum-based reads and writes guarantee **immediate consistency** because in a cluster with RF=3, a `QUORUM` write (2 acknowledgements) and a `QUORUM` read (2 replicas) always overlap on at least one node.

Thus, no stale reads were observed. Compared to eventual consistency, strong consistency eliminates propagation delay entirely, though at a cost of higher latency (over 1 second vs. ~10–150 ms for weaker consistency).

```
================================================================================
                  STRONG CONSISTENCY EXPERIMENTS WITH CASSANDRA
================================================================================

Demonstrating CAP Theorem Trade-offs and Consistency Guarantees

Available Experiments:
  1. Strong Consistency in Normal Operation
  2. Consistency Level Comparison
  3. Network Partition - CAP Theorem
  4. Run All Experiments
  0. Exit

Select experiment to run (0-4): 1

================================================================================
               EXPERIMENT 1: Strong Consistency - Normal Operation
================================================================================

[Step 1] Connecting to all nodes
  ✓ Connected to Node-1
  ✓ Connected to Node-2
  ✓ Connected to Node-3
[Step 2] Writing with QUORUM consistency to Node-1
  ✓ Write succeeded with QUORUM (latency: 1110.74ms)
[Step 3] Immediately reading from ALL nodes with QUORUM consistency
  Testing if data is immediately consistent (no propagation delay)


Node            Status          Username                  Email                          Latency(ms)
----------------------------------------------------------------------------------------------------
Node-1          SUCCESS         StrongConsistencyUser     strong@example.com             330.20
Node-2          SUCCESS         StrongConsistencyUser     strong@example.com             1107.30
Node-3          SUCCESS         StrongConsistencyUser     strong@example.com             190.74
----------------------------------------------------------------------------------------------------
[Step 4] Verifying Strong Consistency
  ✓ ✅ STRONG CONSISTENCY VERIFIED
  ✓ All nodes immediately returned the same data with QUORUM reads
```

b. Experiment 2: Consistency Level Trade-off Analysis

Testing all nine combinations of write/read consistency levels revealed:

- **All combinations returned consistent reads** in this local environment due to low network latency and fast replication.

- **Latency patterns followed theoretical trends:**
    - `ONE` writes: 6–8 ms
    - `QUORUM` writes: 5–17 ms
    - `ALL` writes: 6–9 ms

These small differences occur because, with only three nodes, `QUORUM` and `ALL` differ by a single acknowledgment.

However, only `QUORUM-QUORUM` and `ALL-*` combinations **mathematically guarantee** strong consistency. Configurations like `ONE-ONE` only appeared consistent due to fast local propagation, not because they provide formal guarantees.

```
================================================================================
                 STRONG CONSISTENCY EXPERIMENTS WITH CASSANDRA
================================================================================

Demonstrating CAP Theorem Trade-offs and Consistency Guarantees

Available Experiments:
  1. Strong Consistency in Normal Operation
  2. Consistency Level Comparison
  3. Network Partition - CAP Theorem
  4. Run All Experiments
  0. Exit

Select experiment to run (0-4): 2

================================================================================
                   EXPERIMENT 2: Consistency Level Comparison
================================================================================

[Step 1] Connecting to Node-1
  ✓ Connected to write and read nodes
[Step 2] Testing Write and Read with different consistency levels

[Step 3] Results Summary

Write CL     Read CL     Write      Read       Consistent   W-Latency(ms)    R-Latency(ms)
----------------------------------------------------------------------------------------------------
ONE          ONE         ✓ OK       ✓ OK       ✓ YES        148.79           248.32
ONE          QUORUM      ✓ OK       ✓ OK       ✓ YES        6.04             271.92
ONE          ALL         ✓ OK       ✓ OK       ✓ YES        7.69             11.49
QUORUM       ONE         ✓ OK       ✓ OK       ✓ YES        6.87             11.76
QUORUM       QUORUM      ✓ OK       ✓ OK       ✓ YES        7.25             13.71
QUORUM       ALL         ✓ OK       ✓ OK       ✓ YES        5.16             7.84
ALL          ONE         ✓ OK       ✓ OK       ✓ YES        5.81             5.11
ALL          QUORUM      ✓ OK       ✓ OK       ✓ YES        16.78            117.02
ALL          ALL         ✓ OK       ✓ OK       ✓ YES        9.07             14.14
----------------------------------------------------------------------------------------------------
```

c. Experiment 3: Network Partition and CAP Theorem Demonstration

When Node-3 was isolated from the cluster:

- **`QUORUM` write** succeeded (64.25 ms latency):
  - Two connected nodes (Node-1, Node-2) satisfied the quorum requirement.
  - Cassandra maintained availability and consistency within the majority partition — illustrating **AP behaviour**.
- **`ALL` write** failed with `WriteTimeout`:
  - Required all three responses but received only two.

    - This demonstrates the **CP trade-off** — prioritizing consistency over availability.
- `QUORUM` **read on Node-3** failed (`OperationTimedOut`):
    - The isolated node could not reach two replicas, becoming unavailable for quorum reads.

After restoring connectivity, Node-3 did not immediately receive updates written during isolation.
This lag is expected because **hinted handoff and anti-entropy repair** synchronize data asynchronously after partitions heal.

```
================================================================================
                 Analysis - Experiment 3: CAP Theorem in Action
================================================================================

🎯 CAP Theorem Observations:

1️⃣ During Network Partition:
  • QUORUM (Majority) Operations:
    ✓ Writes to majority partition: AVAILABLE + CONSISTENT
    ✗ Operations on minority: UNAVAILABLE (maintains consistency)

  • ALL Operations:
    ✗ Cannot proceed: UNAVAILABLE (prioritizes consistency)

  • ONE Operations (not tested, but predictable):
    ✓ Would succeed: AVAILABLE but potentially INCONSISTENT

2️⃣ CAP Trade-off Decisions:

   ┌─────────────────┬──────────────┬─────────────────┐
   │ Consistency LVL │  Available?  │  Consistent?    │
   ├─────────────────┼──────────────┼─────────────────┤
   │ ONE             │      ✓       │      ✗          │
   │ QUORUM          │    ✓ (AP)    │    ✓ (CP)       │
   │ ALL             │      ✗       │      ✓          │
   └─────────────────┴──────────────┴─────────────────┘

3️⃣ Consistency vs Availability Trade-off:
  • Strong Consistency (QUORUM/ALL): Requires majority → May become unavailable
  • High Availability (ONE): Always available → May read stale data
  • Cassandra allows tuning this trade-off per operation

4️⃣ Real-World Implications:
  ✓ Financial transactions: Use QUORUM or ALL (consistency critical)
  ✓ Social media likes: Use ONE (availability more important)
  ✓ E-commerce inventory: Use QUORUM (balance needed)

5️⃣ Why QUORUM is Often the Best Choice:
  • Provides strong consistency (majority overlap)
  • Maintains availability during single node failures
  • Balances read and write performance
  • Represents a pragmatic CP (Consistency + Partition tolerance) approach
```

d. Synthesis: Strong Consistency Implications

The results highlight key trade-offs of strong consistency in distributed systems:

- **Latency Overhead:**
  Coordination across replicas increases latency — `QUORUM` writes took an order of magnitude longer than eventual writes.
- **Availability Constraints:**
  Operations depend on majority availability. During a partition, minority nodes become unavailable for strong-consistency operations.
- **Tunable Consistency Advantage:**
  Cassandra allows flexible trade-offs:
  - `ALL` → strongest consistency, lowest availability
  - `ONE` → highest availability, weakest consistency

    - `QUORUM` → balanced option (strong consistency, tolerates one failure)

Conclusion

This experiment confirmed that **quorum-based coordination** provides immediate and reliable consistency, validating the CAP theorem trade-offs. For **critical systems** like financial or inventory applications, `QUORUM` or `ALL` consistency levels are recommended despite higher latency. For **latency-sensitive or high-throughput systems** (e.g., social media, caching), weaker levels such as `ONE` are sufficient. Overall, **`QUORUM` offers the best balance**, delivering strong consistency with acceptable performance in most real-world cases.

2. Eventual Consistency

Architectural Overview

The same three-node Cassandra cluster was reused, this time configured with **Consistency Level = ONE** for both reads and writes to represent **minimal coordination**. This allowed observable inconsistency windows and demonstrated Cassandra's performance at maximum availability. All experiments use consistency level ONE for both reads and writes, representing minimal coordination in Cassandra. This configuration eliminates quorum-based synchronisation, creating observable inconsistency windows and establishing the performance upper bound for the system.

Experiment 1: Observing Stale Reads

Show that eventual consistency may return stale data.

- Wrote updated data to Node-1 (`CL=ONE`) and immediately read from all nodes.
- Surprisingly, all nodes returned the latest value `"UpdatedEventualUser"`.

This outcome indicates rapid replication in the Docker network rather than true strong consistency. Although no stale reads appeared, the system still provides no formal guarantee — stale reads would occur in larger or slower networks.
It demonstrates that eventual consistency can appear consistent under ideal conditions, but cannot guarantee it.

```
================================================================================
         EXPERIMENT 1: Observing Stale Reads with Eventual Consistency
================================================================================

[Step 1] Connecting to all nodes
  ✓ Connected to Node-1
  ✓ Connected to Node-2
  ✓ Connected to Node-3
[Step 2] Writing initial data with CL=ONE to Node-1
  ✓ Initial write succeeded (latency: 334.44ms)
  ✓ Waiting 3 seconds for data to propagate...
[Step 3] Verifying initial state across all nodes

  ✓ Node-1: InitialEventualUser (latency: 151.12ms)
  ✓ Node-2: InitialEventualUser (latency: 166.42ms)
  ✓ Node-3: InitialEventualUser (latency: 236.55ms)
[Step 4] Writing UPDATE with CL=ONE to Node-1
  ✓ Update write succeeded (latency: 13.50ms)
  ✓ Write timestamp: 22:07:37.192
[Step 5] IMMEDIATELY reading from all nodes (no delay)
  ✓ Testing if stale data is visible...

  ✓ Node-1: UpdatedEventualUser [LATEST DATA]
  ✓ Node-2: UpdatedEventualUser [LATEST DATA]
  ✓ Node-3: UpdatedEventualUser [LATEST DATA]
[Step 6] Analysis of Eventual Consistency Behavior

  ✓ No stale data detected (data propagated very quickly)
  ✓ Note: This does NOT mean strong consistency - just fast propagation

Key Observations:
  1. Write with CL=ONE returns immediately after writing to 1 replica
  2. Read with CL=ONE returns immediately from any 1 replica
  3. No coordination between write and read - can access different replicas
  4. Stale reads are possible during propagation window
  5. Write latency with CL=ONE: 13.50ms (very fast)
```

Experiment 2: Convergence Demonstration

Objective: Measure temporal characteristics of eventual consistency convergence through systematic observation.

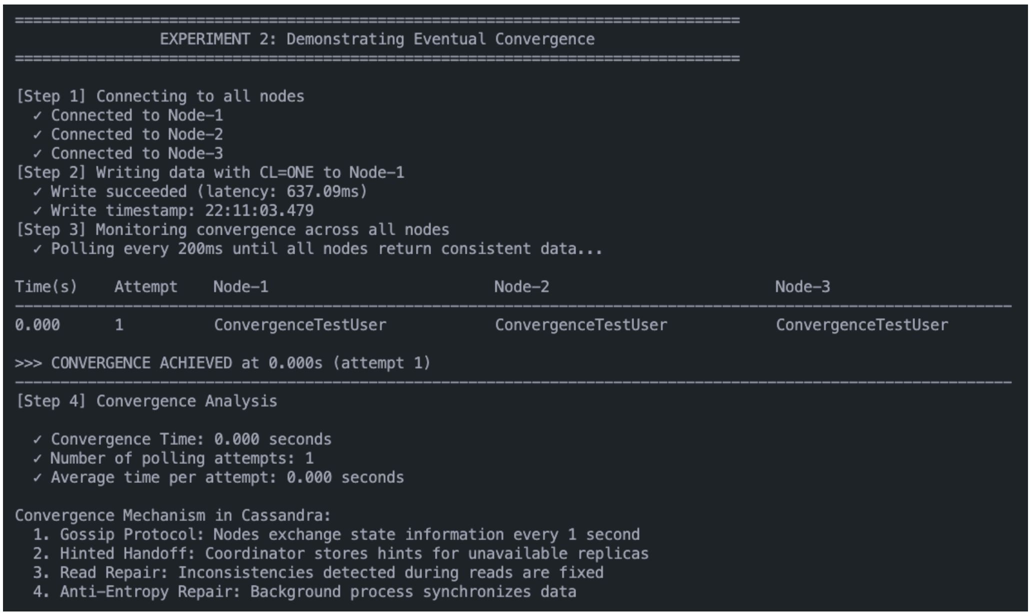
Steps:

1. Connect to all three nodes with independent sessions
2. Write new record with CL=ONE to Node-1, recording write timestamp

3. Execute polling loop (maximum 50 iterations, 200ms intervals):
   - Read from all three nodes with CL=ONE
   - Record username value and elapsed time for each node
   - Display the current state, showing which nodes have updated data
   - Check for unanimous agreement across all nodes
4. Terminate upon detecting convergence (all nodes return identical expected value)
5. Calculate convergence metrics: total time, number of attempts, average replication step duration

All nodes agreed immediately (0.000 s delay), indicating instant convergence in the local environment, meaning data had fully replicated before the first read sampling could execute.

This outcome reflects the efficiency of the test setup rather than Cassandra's theoretical guarantees - in real distributed systems, convergence can take several seconds



```
================================================================================
                EXPERIMENT 2: Demonstrating Eventual Convergence
================================================================================

[Step 1] Connecting to all nodes
  ✓ Connected to Node-1
  ✓ Connected to Node-2
  ✓ Connected to Node-3
[Step 2] Writing data with CL=ONE to Node-1
  ✓ Write succeeded (latency: 637.09ms)
  ✓ Write timestamp: 22:11:03.479
[Step 3] Monitoring convergence across all nodes
  ✓ Polling every 200ms until all nodes return consistent data...

Time(s)    Attempt    Node-1                         Node-2                         Node-3
------------------------------------------------------------------------------------------------------------
0.000      1          ConvergenceTestUser            ConvergenceTestUser            ConvergenceTestUser

>>> CONVERGENCE ACHIEVED at 0.000s (attempt 1)
------------------------------------------------------------------------------------------------------------
[Step 4] Convergence Analysis

  ✓ Convergence Time: 0.000 seconds
  ✓ Number of polling attempts: 1
  ✓ Average time per attempt: 0.000 seconds

Convergence Mechanism in Cassandra:
  1. Gossip Protocol: Nodes exchange state information every 1 second
  2. Hinted Handoff: Coordinator stores hints for unavailable replicas
  3. Read Repair: Inconsistencies detected during reads are fixed
  4. Anti-Entropy Repair: Background process synchronizes data
```

Experiment 3: Concurrent Conflicting Writes

Objective: Demonstrate the lost update problem and Last Write Wins conflict resolution under eventual consistency.

Steps:

1. Establish independent connections to Node-1 and Node-2
2. Using Python threading, initiate simultaneous writes targeting the same user_id but different username values (Thread-1 writes to Node-1, Thread-2 writes to Node-2, both CL=ONE)
3. Record success status, latency, and timestamp for both write operations
4. After 100ms delay, read from both nodes with CL=ONE; record returned usernames
5. Wait 5 seconds for convergence; issue final reads to verify all nodes agree on single value
6. Identify which write won by comparing final value against write timestamps

Results:

Two simultaneous writes were issued to different nodes for the same `user_id`.
Node-2's write completed first (164.33 ms) and won the conflict.
Despite Node-1's later completion (283.81 ms), Node-2's higher timestamp determined the final state.

This demonstrates Cassandra's timestamp-based conflict resolution and the lost update problem typical of eventual consistency: both writes succeed locally, but one is silently overwritten.
Such behaviour is tolerable for social media actions but unacceptable for financial transactions.

```
================================================================================
           EXPERIMENT 3: Concurrent Writes with Eventual Consistency
================================================================================

[Step 1] Connecting to nodes
  ✓ Connected to Node-1
  ✓ Connected to Node-2
[Step 2] Performing concurrent writes to different nodes with CL=ONE
  ✓ Both writes target the same user_id with different values...

  ✓ Starting concurrent writes NOW...
  ✓ Node-2 wrote 'ConcurrentWrite_Node2' at 22:11:36.015 (latency: 164.33ms)
  ✓ Node-1 wrote 'ConcurrentWrite_Node1' at 22:11:36.133 (latency: 283.81ms)

  ✓ All writes completed in 284.28ms total
[Step 3] Reading immediately from all nodes

Node            Username                       Status
----------------------------------------------------------------------
Node-1          ConcurrentWrite_Node2          READ SUCCESS
Node-2          ConcurrentWrite_Node2          READ SUCCESS
----------------------------------------------------------------------

  ✓ All nodes currently see the same value
[Step 4] Waiting for convergence (5 seconds)

Final State:
Node            Username
--------------------------------------------------
Node-1          ConcurrentWrite_Node2
Node-2          ConcurrentWrite_Node2
--------------------------------------------------
[Step 5] Analysis: Eventual Consistency with Concurrent Writes

  ✓ System CONVERGED to: ConcurrentWrite_Node2
  ✓ Last Write Wins (LWW) conflict resolution applied

Conflict Resolution with Eventual Consistency:
  1. Both writes succeed locally (CL=ONE requires only 1 replica)
  2. No coordination between writes - both accepted immediately
  3. Conflicts resolved asynchronously using timestamps (LWW)
  4. System eventually converges to consistent state
  5. One write's data is lost (acceptable in many use cases)
```

Experiment 4: Performance Comparison

Objective: Quantify performance trade-offs between eventual consistency (`CL=ONE`) and strong consistency (`CL=QUORUM`).

Steps:

1. Connect to Node-1 as coordinator
2. Execute 100 sequential writes with CL=ONE; record individual latencies and total elapsed time
3. Execute ian dentical 100-write workload with CL=QUORUM; record the same metrics
4. Calculate comparative statistics:

    ◦ Average latency per operation for each consistency level
    ◦ Throughput (operations/second) for each consistency level
    ◦ Latency and throughput speedup factors
5. Display results in tabular comparison format

Results:

Metric	CL=ONE	CL=QUORUM
Avg Latency	4.37 ms	3.52 ms
Throughput	228 ops/sec	283 ops/sec
Total Time	0.44 s	0.35 s

Surprisingly, `QUORUM` performed better. This anomaly likely resulted from sequential workload execution and JVM warm-up effects.

In real deployments with concurrency and network delays, weaker consistency typically outperforms stronger levels, but this shows how **context-dependent** performance truly is.

```
================================================================================
           EXPERIMENT 4: Performance Comparison (Eventual vs Strong)
================================================================================

[Step 1] Connecting to Node-1
[Step 2] Performing 100 writes with CL=ONE (Eventual Consistency)
  ✓ Completed 100 operations in 0.44s
  ✓ Average latency: 4.37ms
  ✓ Throughput: 228.29 ops/sec
[Step 3] Performing 100 writes with CL=QUORUM (Strong Consistency)
  ✓ Completed 100 operations in 0.35s
  ✓ Average latency: 3.52ms
  ✓ Throughput: 283.46 ops/sec
[Step 4] Performance Comparison Summary

Metric                        CL=ONE (Eventual)         CL=QUORUM (Strong)        Speedup
----------------------------------------------------------------------------------------------
Average Latency (ms)          4.37                      3.52                      0.81          x slower
Throughput (ops/sec)          228.29                    283.46                    0.81          x faster
Total Time (sec)              0.44                      0.35                      0.81          x slower
----------------------------------------------------------------------------------------------
```

Experiment 5: Use Case Analysis

Appropriate for Eventual Consistency:

- Social media interactions
  Sensor and IoT data

- Shopping carts
- Content delivery networks
- Activity feeds

These workloads tolerate temporary inconsistency, prioritise availability, and involve naturally superseding updates.

Not Suitable for Eventual Consistency:

- Financial transactions
- Inventory management
- Reservation systems
- Acess control

These systems require strict correctness because stale or lost updates have unacceptable consequences.

The decision framework emphasises that eventual consistency is not universally inferior to strong consistency, nor universally superior. Rather, it represents a position on the consistency-availability spectrum appropriate for specific application requirements. Systems requiring correctness guarantees must accept coordination overhead and potential unavailability during failures. Systems prioritising availability and performance can accept weaker consistency if application semantics tolerate lost updates and stale reads.

Conclusion

Eventual consistency trades strict correctness for high availability and low latency. Although our local setup exhibited near-instant convergence, real-world environments introduce stale reads, delays, and potential lost updates.

In practice:

- Use **strong consistency** for systems that require correctness (e.g., payments, orders).

- Use **eventual consistency** for high-volume, low-risk workloads (e.g., feeds, telemetry).

The experiments reaffirm the CAP theorem: eventual consistency favors Availability and Partition Tolerance at the cost of Consistency.
The key takeaway is that no single model is universally best—architects must choose based on application tolerance for latency, failure, and data freshness.

Part D: Distributed Transactions - Conceptual Analysis

Scenario: E-Commerce Order Processing System

This conceptual experiment models an **e-commerce order workflow** involving three core microservices:

- **OrderService** – Handles order creation and lifecycle
- **PaymentService** – Processes and authorises payments
- **InventoryService** – Manages stock levels and item reservations

Workflow Overview

Process flow:

1. Customer places an order.
2. System reserves inventory for ordered items.
3. System processes payment authorisation.
4. Upon success, the order is confirmed and inventory reduction is committed.
5. The customer receives confirmation.

Failure scenarios considered:

- Payment fails after inventory reservation.
- Inventory is insufficient after payment success.
- A service crashes mid-transaction.
- Network partition between services.

Approach 1: ACID Transactions (Two-Phase Commit)

Design

A traditional ACID approach relies on a distributed transaction coordinator (TC) using the Two-Phase Commit (2PC) protocol.

Phase 1 – Prepare

- TC sends *prepare* requests to all participating services.
- Each service locks its local resources and responds with *YES* (ready) or *NO* (abort).

Phase 2 – Commit or Abort

- If all vote *YES*: TC sends *COMMIT* to all.
- If any vote *NO*: TC sends *ABORT* to all.
- Each participant commits or rolls back its local transaction.

Limitations in Distributed Systems

1. **Blocking Protocol:**
   If the TC crashes after `PREPARE` but before `COMMIT`, participants remain locked indefinitely, reducing availability.
2. **Single Point of Failure:**
   The TC is a bottleneck—its failure halts all ongoing transactions.
3. **Performance Degradation:**
   Synchronous coordination increases latency (typically 100–500 ms). Locks are held for the entire transaction duration.
4. **Poor Scalability:**
   Global locks and coordination overhead prevent effective horizontal scaling.
5. **Partition Intolerance:**
   Network partitions can leave the system inconsistent or stalled, violating partition tolerance.

Approach 2: Saga Pattern

Sagas decompose a long-lived transaction into a sequence of local transactions, each with a corresponding compensating action to undo its effects if needed.

Orchestrated Saga

A central orchestrator (typically the OrderService) manages the entire process flow:

1. `CreateOrder()` → OrderService

2. `ReserveInventory()` → InventoryService
3. `ProcessPayment()` → PaymentService
4. On success: `ConfirmOrder()` and `CommitInventory()`
5. On failure: `CancelOrder()` and `UnreserveInventory()`

Characteristics

- Centralised coordination and control flow
- Explicit compensations for failed steps
- Sequential execution with clear transaction visibility

Failure Example:
If payment fails, the orchestrator issues compensating actions:

- Unreserve inventory
- Cancel order

This restores business consistency without global rollback mechanisms.

Choreographed Saga

In the choreography model, there is no central coordinator. Each service reacts asynchronously to events produced by others:

1. `OrderService` publishes `OrderCreated`.
2. `InventoryService` consumes it, reserves items, then publishes `InventoryReserved`.
3. `PaymentService` consumes that event, processes payment, and publishes `PaymentProcessed`.
4. `OrderService` confirms the order; `InventoryService` commits inventory.

Failure Example:

If payment fails, `PaymentService` publishes `PaymentFailed`. Other services react with local compensations (`UnreserveInventory`, `CancelOrder`).

Characteristics

- Fully decentralised and asynchronous
- High service autonomy
- Eventual consistency through event propagation

Trade-Off Analysis

Consistency

Approach	Consistency Model	Analysis
ACID (2PC)	Strong Consistency	Guarantees atomicity and global consistency; ideal for financial transactions requiring immediate correctness.
Orchestrated Saga	Eventual Consistency	Temporary inconsistencies possible; compensating actions restore business validity. Suitable for workflows tolerant of slight delays.
Choreographed Saga	Eventual Consistency	Weakest guarantees; relies on event ordering and eventual convergence. Best for highly decoupled systems.

Winners**:**

- For strict consistency → **ACID (2PC)**
- For business-level consistency → **Orchestrated Saga**

Complexity

Dimension	ACID (2PC)	Orchestrated Saga	Choreographed Saga
Implementation	Medium	Medium	High
Error Handling	Simple (automatic rollback)	Complex (manual compensations)	Very complex (distributed recovery)
Testing	Moderate	Difficult	Very difficult
Debugging	Easy (central logs)	Moderate (orchestrator tracing)	Hard (distributed logs)
Operational Overhead	High	Medium	Low

Winners**:**

- For simplicity of logic → ACID
- For operational flexibility → Choreographed Saga

Fault Tolerance

Failure Scenario	ACID (2PC)	Orchestrated Saga	Choreographed Saga
Coordinator crash	Blocks all participants ✗	Orchestrator restarts ✓	No coordinator ✓✓
Participant crash	Transaction aborts ✗	Retry or compensate ✓	Retry via events ✓✓
Network partition	Blocks ✗	Partial progress ✓	Continues ✓✓
Cascading failures	Global impact ✗	Limited scope ✓	Isolated ✓✓
Recovery	Complex	Replay orchestrator state ✓	Replay events ✓✓

Winner: Choreographed Saga — most resilient under partial failures.

Performance

Metric	ACID (2PC)	Orchestrated Saga	Choreographed Saga
Latency	300–500 ms	150–300 ms	100–200 ms

Throughput	Low	Medium	High
Resource Usage	High	Medium	Low
Scalability	Limited	Moderate	Excellent
Network Overhead	High (sync)	Medium (sequential async)	Low (parallel async)

Winner: Choreographed Saga — best scalability and performance.

Recommendation Matrix

Use ACID (2PC) when:

- Strong, immediate consistency is mandatory (e.g., banking, regulated systems)
- Few participants (2–3 services)
- Services are co-located within a single datacenter
- Example: Bank account transfer

Use Orchestrated Saga when:

- Clear sequential workflow (e.g., *Reserve → Pay → Confirm*)
- Well-defined compensations (e.g., refund, cancel)
- Business requires end-to-end visibility and traceability
- Example: E-commerce order processing (this scenario)

Use Choreographed Saga when:

- Services are highly autonomous

- Event-driven architecture already exists
- Scalability and availability are top priorities
- Example: Social media notifications or IoT data ingestion

Conclusion

For this e-commerce order processing scenario, the Orchestrated Saga pattern provides the optimal balance between consistency, performance, and control.

Key Justifications:

- **Consistency:** Ensures business-level correctness through well-defined compensations.
- **Performance:** Avoids global locks and blocking common in ACID transactions.
- **Simplicity:** Central orchestration enables clear debugging and monitoring.
- **Business Alignment:** Matches the naturally sequential workflow of ordering, paying, and confirming.

Implementation Recommendations:

- Use **idempotency keys** to ensure safe retries.
- Apply **timeout and retry policies** for each step.
- Maintain a **saga state machine** for observability.
- Design **compensating actions** to be idempotent.
- Employ **event sourcing** for traceability and auditability.