**Hints for Solving Lab 2**

Lab 2 focuses on applying architectural principles and trade-off analysis to distributed data management, specifically replication and consistency models. The core theme is determining the **"least worst combination of trade-offs"** for data architecture.

**I. Conceptual Foundation (The "Why")**

Before starting implementation, focus on the required depth of analysis. The goal is not just *how* to set up the database, but *why* you chose those settings, linking them to core distributed concepts.

1. **Revisit the CAP Theorem:** The trade-offs explored in this lab (Part C) relate directly to the CAP theorem (Consistency, Availability, Partition Tolerance). When you configure your system for **strong consistency** (e.g., using a high write concern like QUORUM or ALL), be prepared to observe and explain how the system sacrifices **Availability** during a network **Partition** (P) event. Conversely, when you configure for **eventual consistency** (e.g., write concern ONE), you maintain **Availability** during a **Partition** by sacrificing strong **Consistency**.
2. **Architectural Thinking:** Remember the **Second Law of Software Architecture: "Why is more important than how"**. Your report should center on justifying your observations and configurations, focusing on the architectural implications (e.g., how latency or availability is affected).
3. **The "Least Worst" Design:** Recognize that for every configuration you test (e.g., strong vs. eventual consistency, primary-backup vs. leaderless replication), there is a trade-off (an advantage and a disadvantage). Use the provided criteria to analyze which trade-offs yield the most appropriate, or **"least worst,"** solution for a hypothetical business scenario.

**II. Implementation Setup (Part A)**

The instructions offer flexibility but provide clear minimum requirements for setting up the environment.

1. **Technology Choice (The "What"):** You are encouraged to choose your own distributed NoSQL database. The recommendations—**Apache Cassandra**, **MongoDB (with Replica Sets)**, or **Redis Cluster**—are excellent choices because they natively support the replication and tunable consistency models required for Parts B and C.
2. **Cluster Size:** A **minimum of 3 nodes** is highly recommended. Using fewer nodes will make it difficult or impossible to demonstrate critical aspects like **quorum writes/reads** and **fault tolerance/partition handling**.
3. **Environment:** Use a local environment like **Docker** for easy deployment and simulation of failures (e.g., shutting down a node). Document your setup (e.g., Docker Compose file or commands) clearly in your report.
4. **Avoid Outdated Tech:** Based on past module feedback, use **modern versions** of Java or other languages/frameworks, avoiding outdated versions like Java 8.

**III. Replication & Consistency Experiments (Parts B and C)**

These parts require configuring and testing the core mechanisms of distributed data storage.

1. **Replication Factor (RF) and Write/Read Concerns (Part B.1/C.1):**
   * Set your replication factor (RF) to 2 or 3.
   * To demonstrate the trade-off between Consistency and Availability, explicitly configure different **write/read concerns** (e.g., ONE, QUORUM, ALL in Cassandra; w:1, w:majority in MongoDB).
   * **Hint:** A strong consistency configuration requires writes and reads to interact with a quorum or all nodes (e.g., QUORUM or ALL). An eventual consistency configuration requires interacting with only a subset (e.g., ONE or w:1).
2. **Simulating Failure (Part B.2/C.1):**
   * To test the CAP implications, you must **simulate a failure** (e.g., stop the primary node in a leader-follower setup or partition the network between nodes).
   * **Observation Focus:** Observe how operations behave during this simulated partition. Under **strong consistency** settings, write operations are expected to **block or fail** (sacrificing A for C). Under **eventual consistency**, writes are expected to **succeed** on available nodes (sacrificing C for A).
3. **Eventual Consistency Demonstration (Part C.2):**
   * Perform a write operation to Node A using eventual consistency settings. Immediately try to read that data from Node B.
   * The goal is to **read stale (old) data** from Node B, demonstrating the "eventual" lag. A simple loop reading the data repeatedly until the correct value appears is a good way to visualize the convergence process.

**IV. Distributed Transactions (Part D - Conceptual)**

Part D is a conceptual exercise focused on **trade-off analysis** of complex distributed transactions. No code implementation is required.

1. **Review Challenges:** Recall that highly distributed systems generally abandon traditional ACID transactions because they become problematic due to network latency, decoupling goals, and the difficulty of coordination across services.
2. **Saga Pattern Analysis:** The challenge asks you to describe how a workflow (e.g., placing an e-commerce order) is managed using **Sagas** (Orchestrated or Choreographed).
   * **Orchestration** involves a central service directing the flow and managing state, which improves control but increases coupling and adds latency.
   * **Choreography** involves services reacting independently to events, which is more decoupled but significantly complicates error handling (requiring compensating messages spread across multiple services).
3. **Trade-off Focus:** The hint here is to **analyze the trade-offs** of Sagas versus ACID in terms of **consistency, complexity, fault tolerance, and performance**. Sagas generally favor availability and performance but sacrifice strong transactional consistency for eventual consistency.

**V. Report and Documentation**

The report is weighted equally with the implementation's correctness, emphasizing the architectural insight.

* **Analysis Depth:** Ensure the analysis section for each experiment clearly links your observations (e.g., latency, downtime, data state) back to the theoretical concepts: **CAP Theorem**, consistency models, and the **architectural implications** (fault tolerance, performance, complexity).
* **Documentation:** Provide clear setup details (e.g., Docker commands) and use **screenshots or console output** as evidence to support your observations.
* **Justification:** Justify **why** you would choose a specific replication/consistency setting for a given business case (e.g., why eventual consistency is desirable for social media likes or sensor data, but strong consistency is critical for inventory or payment processing).

**Suggested Reading**

**Core Distributed Systems Concepts**

**1. CAP Theorem & Trade-offs:**

* **Brewer's Conjecture and the CAP Theorem (2002)** - The original paper by Seth Gilbert and Nancy Lynch that formally proved Brewer's conjecture. A must-read for understanding the theoretical foundation.
* **"Please stop calling databases CP or AP"** - Martin Kleppmann's blog post that provides a more nuanced modern understanding of CAP.
* **"A Plain English Introduction to CAP Theorem"** - Kaushik Sathupadi's blog post that gives intuitive explanations with examples.

**2. Consistency Models:**

* **"Consistency in Non-Transactional Distributed Storage Systems"** - A comprehensive survey paper covering various consistency models.
* **"Eventually Consistent"** - Werner Vogels' (Amazon CTO) classic article explaining why eventual consistency is chosen in practice.
* **Chapter 5 of "Designing Data-Intensive Applications"** by Martin Kleppmann - This is arguably the **best resource** for this entire lab.

**Database-Specific Documentation**

**For Apache Cassandra:**

* **"Cassandra Documentation: Architecture"** - Focus on the sections about replication, consistency levels, and tunable consistency.
* **"Read Repair and Hinted Handoff"** - Understand how Cassandra handles consistency behind the scenes.
* **"Datastax Java Driver Documentation"** - Even if using Python, the concepts of consistency levels, load balancing, and retry policies are well-explained.

**For MongoDB:**

* **"MongoDB Replication"** - Official documentation on replica sets.
* **"Read Concern and Write Concern"** - Crucial for your consistency experiments.
* **"MongoDB and the CAP Theorem"** - MongoDB's official stance on CAP.

**For Redis Cluster:**

* **"Redis Cluster Specification"** - Understand how Redis handles partitioning and failure detection.
* **"Redis Consistency and Durability"** - Documentation on Redis persistence and replication.

**Practical Implementation Guides**

**Docker & Containerization:**

* **"Docker Compose for Multi-Container Applications"** - Official Docker documentation.
* **"Networking in Compose"** - Important for simulating network partitions.

**Client Libraries:**

* **"Cassandra Python Driver Documentation"** - Official driver docs with examples.
* **"PyMongo Documentation"** - If choosing MongoDB with Python.

**Advanced Topics**

**Distributed Transactions & Saga Pattern:**

* **"Saga Pattern"** - [Microservices.io](https://microservices.io/) pattern description.
* **"Applying the Saga Pattern"** - Caitie McCaffrey's talk that's become a classic reference.
* **"Life Beyond Distributed Transactions"** - An older but influential paper by Pat Helland.

**Network Partitions & Failure Handling:**

* **"The Network is Reliable"** - Peter Bailis' article on network reliability assumptions.
* **"Jay Kreps' 'I Can't Believe It's Not Coordination!'"** - Discussion on avoiding coordination where possible.

**Recommended Books**

1. **"Designing Data-Intensive Applications" by Martin Kleppmann** - This should be your **primary reference**. Chapters 5 (Replication), 6 (Partitioning), 7 (Transactions), and 9 (Consistency) are directly relevant.
2. **"Database Internals" by Alex Petrov** - More technical, but excellent for understanding how databases implement these concepts.
3. **"Understanding Distributed Systems" by Roberto Vitillo** - Practical approach with concrete examples.