REPORT

January 29, 2025

1 Clouds 2023

1.1 Question 1:

For each of the following class of applications, explain why they benefit from using the cloud

a) A legacy banking application that is running on a private datacenter

b) An e-commerce web application that has an annual Prime day sale

[POINTS = 2(1+1)]

- **Legacy Banking Application**: Benefits from scalability, cost efficiency, disaster recovery, security, and modernization opportunities.
- **E-commerce Web Application**: Benefits from elastic scalability, cost optimization, global reach, high availability, and rapid deployment.

Both applications leverage the cloud to address their unique challenges and improve performance, reliability, and cost-effectiveness.

2 Question 2:

The figure shows the hardware–software stack used in cloud platforms.

a) What are the main service models offered by cloud service providers?

b) For each service model, explain which parts of the stack are managed by the cloud vendor?

Application
Data
Runtime
Middleware
Operating System
Virtualization
Servers
Storage
Networking

[POINTS = 4(2+2)]

---

a) Main Service Models

1. **Infrastructure as a Service (IaaS)**
2. **Platform as a Service (PaaS)**
3. **Software as a Service (SaaS)**

b) Managed by Cloud Vendor vs. Customer

Service Model	Managed by Cloud Vendor	Managed by Customer
IaaS	Virtualization, Servers, Storage, Networking	OS, Middleware, Runtime, Data, Application
PaaS	Virtualization, Servers, Storage, Networking, OS, Middleware, Runtime	Data, Application
SaaS	Virtualization, Servers, Storage, Networking, OS, Middleware, Runtime, Data, Application	None (customer only uses the application)

Summary

- **IaaS**: Vendor manages infrastructure; customer manages OS and above.
- **PaaS**: Vendor manages infrastructure and platform; customer manages data and application.
- **SaaS**: Vendor manages everything; customer only uses the application.

3 Question 3:

You are the CTO of a web hosting company. Your job is to build a cloud infrastructure that your clients can use to run web applications. One of your clients is PMTrack, an IoT company that has installed millions of air quality sensors in several countries. PMTrack has a web application that tracks air pollution across the world. Each time air pollution crosses a particular threshold at a place, the sensor in that location does a HTTP request passing along to the webapp information about location, particulate matter concentration etc. The webapp is very lightweight (less than 1000 lines of code) and needs to process each request within 200 milliseconds.

As the CTO, answer the following questions:

a) What are all possible virtualization technologies can you use to host the web application?

b) Which one would you pick? Why?

[POINTS = 4(2+2)]

---

a) Possible Virtualization Technologies

1. **Virtual Machines (VMs)**

2. **Containers**

3. **Serverless Computing**

4. **MicroVM-based Services**

b) Recommended Technology Serverless Computing (e.g., AWS Lambda, Azure Functions)

Why?
1. **Efficient** for lightweight apps (<1000 lines of code).
2. **Low latency** (processes requests in <200ms).
3. **Auto-scaling** for millions of global sensors.
4. **Cost-effective** (pay-per-request).
5. **No infrastructure management** required.
6. **Global reach** with low-latency responses.

Summary

- **Best Choice**: Serverless Computing.
- **Reason**: Efficient, scalable, low-latency, and cost-effective for PMTrack's global IoT application.

4 Question 4:

We said that x86 is not strictly virtualizable. Why is each of the following statements true or false? Say true/false and one line explanation of why?

a. An interpretation based hypervisor will not be able to implement virtual x86 CPU

b. A trap-and-emulate hypervisor will not be able to implement virtual x86 CPU

[POINTS = 4(2+2)]

- **a) False**: Interpretation-based hypervisors can virtualize x86 but are inefficient.
- **b) True**: Trap-and-emulate hypervisors cannot fully virtualize x86 due to non-virtualizable instructions.

5 Question 5:

a) What type of parallelization is used in this code?

```
#pragma omp declare simd
double BlackBoxFunction(double x);
...
const double dx = a/(double)n;
double integral = 0.0;
```

```
#pragma omp simd reduction(+: integral)
for (int i = 0; i < n; i++) {
    const double xip12 = dx*((double)i + 0.5);
    const double dI = BlackBoxFunction(xip12)*dx;
    integral += dI;
}
```

b) What other type of parallelism could be used to accelerate same code on a CPU?

c) What's the difference between CPU and GPU in terms of parallelism?

d) Order FPGA, CPU, GPU in order of generality (A > B => A is more general)?

[POINTS = 4(1+1+1+1)]

a) Type of Parallelization

- **Answer**: **SIMD**
- **Reason**: `#pragma omp simd` indicates SIMD parallelism.

b) Other Parallelism for CPU

- **Answer**: **Multithreading**
- **Reason**: Use `#pragma omp parallel for` for thread-based parallelism.

c) CPU vs. GPU Parallelism

- **Answer**:
  - **CPU**: Task parallelism, low latency.
  - **GPU**: Data parallelism, high throughput.

d) Order of Generality

- **Answer**: **CPU > GPU > FPGA**
- **Reason**: CPU is most general-purpose; FPGA is least.

6 Question 6:

MapReduce has proved an extremely popular framework for distributed computation on large clusters, because it masks many of the painful parts of ganging together thousands of nodes to accomplish a task.

(a) In general high-performance computing (HPC), programmed using message passing, what is the technique used to provide fault tolerance?

(b) What limitation does MapReduce places upon the Map functions so that the framework can hide failures from the programmer?

(c) How does MapReduce handle "straggler" tasks that take longer than all of the others (e.g., perhaps they're running on a slower machine or one with buggy hardware)?

(d) MapReduce and the Google File System (GFS) were designed to work well together. GFS exposes block replica locations via an API to MapReduce. What important optimization in MapReduce is enabled by this?

[POINTS = 4 (1*4)]

a) Fault Tolerance in HPC with Message Passing

- **Answer**: **Checkpointing**

- **Explanation**: In HPC, fault tolerance is often achieved through **checkpointing**, where the state of the computation is periodically saved to stable storage. If a failure occurs, the computation can restart from the last checkpoint. #### **b) Limitation on Map Functions for Fault Tolerance**
- **Answer**: **Map functions must be stateless**

- **Explanation**: MapReduce requires **Map functions to be stateless** so that if a node fails, the framework can rerun the Map task on another node without affecting the correctness of the computation. #### **c) Handling Straggler Tasks**
- **Answer**: **Speculative Execution**

- **Explanation**: MapReduce handles straggler tasks by using **speculative execution**, where the framework redundantly executes slow-running tasks on other nodes and uses the result from the first task to complete. #### **d) Optimization Enabled by GFS Block Replica Locations**
- **Answer**: **Data Locality**

- **Explanation**: GFS exposes block replica locations to MapReduce, enabling **data locality optimization**. Map tasks are scheduled on nodes that already have the required data, reducing network overhead and improving performance.

7 Question 7:

Consider the following Spark program:

```
1. lines = spark.textFile("hdfs://...")
2. errors = lines.filter(_.startsWith("ERROR"))
3. errors.persist()
4. errors.filter(_.contains("MySQL")).count()
5. errors.filter(_.contains("HDFS")).map(_.split(\t)(3)).collect()
```

(a) List the lines in the above program during whose execution an RDD is materialized: that is, computed and saved in memory or persisted on disk for use in future transformations. (Briefly explain your answer).

(b) Consider the following scenario. The input file is partitioned into 100 pieces stored in HDFS. The program is executing line 5 on a cluster of 100 worker machines (these are different

machines than the HDFS servers). One worker machine loses power and does not restart. Explain what data (if any) must be recomputed to recover. (Briefly explain your answer.)

[POINTS = 6(3+3)]

a) Lines Where RDDs Are Materialized

- **Answer**: **Lines 4 and 5**
- **Reason**: Actions (`count()` and `collect()`) trigger RDD materialization.

b) Data Recovery After Worker Failure

- **Answer**: **Lost partition(s) of `errors` RDD**
- **Reason**: Only the lost partition(s) are recomputed from HDFS input data; others remain intact.

8 Question 8:

Consider the following database schema:

```
Suppliers(sid: integer, sname: string, address: string)
Parts(pid: integer, pname: string, color: string)
Catalog(sid: integer, pid: integer, cost: real)
```

Write the most optimal relational algebra expression you can for computing the following: "Find the sids of suppliers who supply red or green parts." (Use projection π, selection σ, join $\bowtie$ operators)

[POINTS = 4]

Objective: Find the `sids` of suppliers who supply red or green parts.

Relational Algebra Expression: $\pi_{\text{sid}}\,(\text{Suppliers} \bowtie (\pi_{\text{sid}}\,(\text{Catalog} \bowtie (\sigma_{\text{color='red'} \vee \text{color='green'}}(\text{Parts})))))$

Step-by-Step Explanation:

1. **Select red or green parts**: $\sigma_{\text{color='red'} \vee \text{color='green'}}(\text{Parts})$
   - Filters the `Parts` table to include only rows where the `color` is either "red" or "green".
2. **Join with Catalog**: $\text{Catalog} \bowtie (\sigma_{\text{color='red'} \vee \text{color='green'}}(\text{Parts}))$
   - Joins the filtered `Parts` table with the `Catalog` table to get the `sid` and `pid` of suppliers who supply red or green parts.
3. **Project `sid` from the join result**: $\pi_{\text{sid}}\,(\text{Catalog} \bowtie (\sigma_{\text{color='red'} \vee \text{color='green'}}(\text{Parts})))$
   - Extracts the `sid` of suppliers who supply red or green parts.
4. **Join with Suppliers** (optional, if supplier details are needed): $\text{Suppliers} \bowtie (\pi_{\text{sid}}\,(\text{Catalog} \bowtie (\sigma_{\text{color='red'} \vee \text{color='green'}}(\text{Parts}))))$
   - Joins the result with the `Suppliers` table to include supplier details (if required).
5. **Final projection**: $\pi_{\text{sid}}\,(\text{Suppliers} \bowtie (\pi_{\text{sid}}\,(\text{Catalog} \bowtie (\sigma_{\text{color='red'} \vee \text{color='green'}}(\text{Parts})))))$
   - Projects the final `sids` of suppliers who supply red or green parts.

Final Answer: $\pi_{\text{sid}}\,(\text{Catalog} \bowtie (\sigma_{\text{color='red'} \vee \text{color='green'}}(\text{Parts})))$

This expression is optimal and directly computes the required `sids` without unnecessary joins or projections.

9 Question 9:

In the context of Spark,

a. Why does Catalyst improve the performance of SparkSQL queries?

b. Why does Tungsten improve the performance of SparkSQL queries?

c. What benefit from DeltaLake bring over the traditional data lake?

[POINTS = 6(2 + 2 + 2)]

a) Catalyst

- **Improves Performance**: Optimizes query plans via logical, physical, and cost-based optimizations.

b) Tungsten

- **Improves Performance**: Enhances execution via memory management, cache awareness, and code generation.

c) DeltaLake Benefits

- **Over Traditional Data Lakes**: Provides ACID transactions, schema enforcement, time travel, and scalability.

Summary

- **a) Catalyst**: Query optimization.
- **b) Tungsten**: Execution efficiency.
- **c) DeltaLake**: Reliability, consistency, and scalability.

10 Question 10:

For each of A, C, I and D properties of transactions,

1. Explain what each stands for
2. List which functionalities of a DBMS provides that property?

[POINTS = 6(2 + 4)]

1) Explanation of ACID

- **A (Atomicity)**: All or nothing execution.
- **C (Consistency)**: Valid state before and after.
- **I (Isolation)**: Transactions appear sequential.
- **D (Durability)**: Committed changes survive failures.

2) DBMS Functionalities

- **Atomicity**: Rollback, commit, logging.
- **Consistency**: Constraints, triggers.
- **Isolation**: Locking, isolation levels, Multiversion Concurrency Control (MVCC).
- **Durability**: Logging, replication.

Summary

- **A**: Rollback, commit, logging.
- **C**: Constraints, triggers.
- **I**: Locking, isolation levels, MVCC.
- **D**: Logging, replication.

11 Question 11:

Consider a database with objects X and Y and assume that there are two transactions T1 and T2. Transaction T1 reads object X, and then writes objects Y and X. Transaction T2 reads object X, then reads object X once more, and finally writes objects X and Y (i.e. T1: R(X), W(Y), W(X); T2: R(X), R(X), W(X), W(Y))

1. Give an example schedule with actions of transactions T1 and T2 on objects X and Y that results in a dirty read.
2. Give an example schedule with actions of transactions T1 and T2 on objects X and Y that results in overwrite uncommitted data.
3. For each of the two schedules, show that Strict 2PL disallows the schedule by listing appropriate lock acquisition and release steps.

[POINTS = 8(2+2+4)]

Hint: Use the following approach to write schedules

```
- T1: START Action1 Action2 .... ABORT
```

```
- T2:      START Action1 Action2 .... COMMIT
```

---

1) Dirty Read Schedule

```
T1: START, R(X), W(Y), W(X), COMMIT
T2:        START, R(X), R(X), W(X), W(Y), COMMIT
```

- **Dirty Read**: T2 reads uncommitted X from T1.

2) Overwrite Uncommitted Data Schedule

```
T1: START, R(X), W(Y), W(X), COMMIT
T2:        START, R(X), R(X), W(X), W(Y), COMMIT
```

- **Overwrite**: T1 overwrites X while T2 is active.

3) Strict 2PL Prevents Both

- **Dirty Read**: T1 holds locks until commit, blocking T2 from reading uncommitted X.
- **Overwrite**: T1 holds locks until commit, blocking T2 from writing X.

Summary

1. **Dirty Read**: T2 reads uncommitted X.
2. **Overwrite**: T1 overwrites X before T2 commits.
3. **Strict 2PL**: Locks held until commit prevent both issues.

Note: T2 starts a little later to show interleaving..

12 Question 12:

Assume 2 programs P1 and P2 accessing a file stored on NFS. P1 first updates it and then P2 reads it.

1. We want P2 to see P1's latest update. This might not happen with NFS? Why?
2. AFS does aggressive prefetching. How this is different from caching?
3. Does prefetching in AFS solve the problem mentioned in question #1? Explain why it does or does not solve the problem?

[POINTS = 3(1+1+1)]

1) NFS Issue

- **Why P2 Might Not See P1's Update**: NFS caching delays propagation of updates, causing P2 to read stale data.

2) Prefetching vs. Caching

- **Caching**: Reactive (stores recently accessed data).
- **Prefetching**: Proactive (fetches data before it's requested).

3) Does AFS Prefetching Solve #1?

- **Answer**: **No**.
- **Reason**: Prefetching improves performance but doesn't ensure cache consistency.

Summary

1. **NFS**: Cache consistency delays cause stale reads.
2. **Prefetching**: Proactive data fetching.
3. **AFS Prefetching**: Doesn't solve consistency issues.

13 Question 13:

Three computers at EURECOM (A, B, and C) communicate using a protocol that implements the idea of Lamport clocks (they include their clock time stamp in messages). For reference, if you need a reminder, recall that the three rules of Lamport's algorithm are:

1. At process i, increment logical clock L_i before each event
2. To send message m at process i, apply rule 1 and then include the current local time in the message, i.e., send(m, L_i)
3. To receive a message (m, t) at process j, set $L_j = max(L_j, t)$ and then apply rule 1 before time-stamping the receive event.

At the beginning of time, all three computers begin with their logical clock set to zero. Later, the following sequence of events occurs:

- A sends message M1 to B: "hi".
- After sending M1, A sends message M2 to C: "hi".
- After receiving M1, B sends message M3 to C: "A told me hi".
- After receiving M3 first and then M2, C sends message M4 to A: "B is boring".

(a) Indicate the time included with the messages as they are sent at each step. [POINTS = 4]

Fill in the time values below

- 1. Send$(M1,)$
- 2. Send$(M2,)$
- 3. Send$(M3,)$

- 4. Send($M4, $)

Timestamps for Messages

1. **Send(M1, 1)**
   - A increments its clock to **1** before sending M1.
2. **Send(M2, 2)**
   - A increments its clock to **2** before sending M2.
3. **Send(M3, 3)**
   - B receives M1 (timestamp 1), updates its clock to **1**, increments to **2** for the receive event.
   - B increments to **3** before sending M3.
4. **Send(M4, 6)**
   - C receives M3 (timestamp 3), updates its clock to **3**, increments to **4**.
   - C receives M2 (timestamp 2), updates to **4**, increments to **5**.
   - C increments to **6** before sending M4.

13.0.1 Final Answer

1. Send($M1, \boxed{1}$)
2. Send($M2, \boxed{2}$)
3. Send($M3, \boxed{3}$)
4. Send($M4, \boxed{6}$)

Explanation

- **M1**: A sends with timestamp **1** after incrementing its clock.
- **M2**: A sends with timestamp **2** after another increment.
- **M3**: B increments twice (after receiving M1 and before sending M3).
- **M4**: C increments three times (after receiving M3, M2, and before sending M4).

Lamport clocks ensure causal order by incrementing on events and synchronizing via message timestamps.

(b) Maintaining all clock states from the previous question, three ADDITIONAL messages are sent:

- After receiving M4, A sends message M5 to B: "C is kind of random!"
- After receiving M5, B sends message M6 to A: "C is boring"
- A receives message M6

After all of these messages have been sent and received, what time does each computer think it is? [POINTS = 4]

---

initially: A=0, B=0, C=0

- send(M1):
- send(M2):
- send(M3):
- send(M4):
- send(M5):
- send(M6):
- recv(M6):

Finally - A = - B = - C =

Step-by-Step Calculation:

Initial State:

- $L_A = 0$, $L_B = 0$, $L_C = 0$

Timestamps for Initial Messages (from part a):

1. **Send(M1)**: $L_A = 1 \rightarrow$ **Send(M1, 1)**
2. **Send(M2)**: $L_A = 2 \rightarrow$ **Send(M2, 2)**
3. **Send(M3)**: $L_B = 3 \rightarrow$ **Send(M3, 3)**
4. **Send(M4)**: $L_C = 6 \rightarrow$ **Send(M4, 6)**

Maintaining Clock States after Initial Messages:

- $L_A = 2$
- $L_B = 3$
- $L_C = 6$

Additional Messages:

5. **A sends M5 to B**:
   - After receiving $M4$, A's clock is $L_A = \max(L_A, 6) = 6$.
   - Increment L_A: $L_A = 7$.
   - **Send(M5, 7)**.

6. **B receives M5 and sends M6 to A**:
   - $M5$ has timestamp $t = 7$.
   - B updates its clock: $L_B = \max(L_B, 7) = 7$.
   - Increment L_B: $L_B = 8$.
   - **Send(M6, 8)**.
7. **A receives M6**:
   - $M6$ has timestamp $t = 8$.
   - A updates its clock: $L_A = \max(L_A, 8) = 8$.
   - Increment L_A: $L_A = 9$.

Final Clock States:

- $L_A = 9$
- $L_B = 8$
- $L_C = 6$

Summary of Timestamps for Additional Messages:

1. **Send(M5)**: $L_A = 7$
2. **Send(M6)**: $L_B = 8$
3. **Recv(M6)**: $L_A = 9$

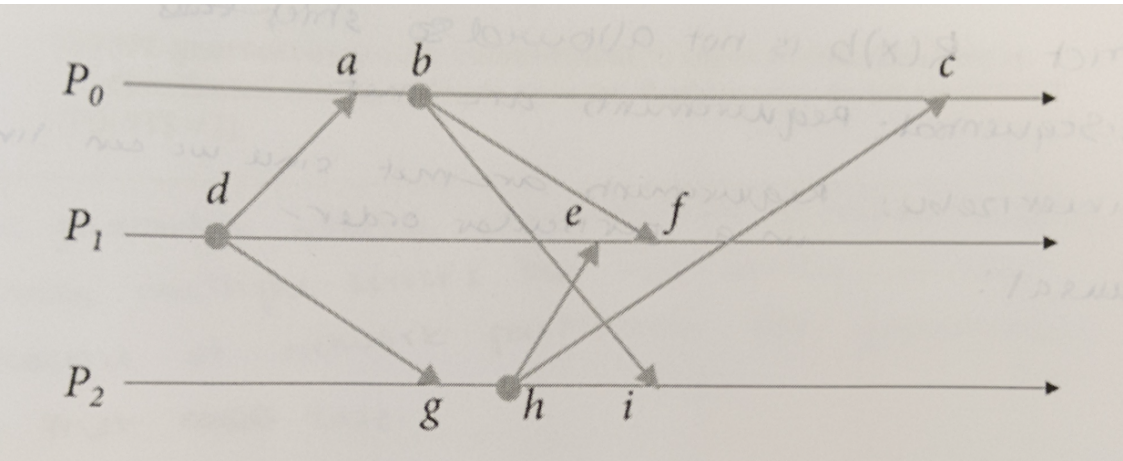
Final Answer:

- **A = 9**
- **B = 8**
- **C = 6**

14 Question 14:

The diagram shows three multicasts to a group of three processes. Sending and receiving are distinct events. But assume that the sending of each multicast message is a single event. So event d of P_1 sending a message to P_0 and P_1 count as one event.

(a) Assign vector timestamps to each event using a sequencing of (P_0, P_1, P_2). The first event on each process gets a sequence number of 1. (Ex: d would have a timestamp of $< 0, 1, 0 >$. Write the timestamp for other events.) [POINTS = 4]



Assigning Vector Timestamps

Step 1: Initialization Each process starts with the vector clock: $\langle 0, 0, 0 \rangle$.

Step 2: Event Descriptions and Timestamps

1. **Event** d (on P_1):
   - P_1 sends a message to P_0 and P_2.
   - Increment P_1's clock: $\langle 0, 1, 0 \rangle$.
   - Timestamp for $d = \langle 0, 1, 0 \rangle$.
2. **Event** a (on P_0):
   - P_0 receives the message from d.
   - Update P_0's clock: $\max(\langle 0, 0, 0 \rangle, \langle 0, 1, 0 \rangle) = \langle 0, 1, 0 \rangle$.
   - Increment P_0's clock: $\langle 1, 1, 0 \rangle$.
   - Timestamp for $a = \langle 1, 1, 0 \rangle$.
3. **Event** b (on P_0):
   - P_0 sends a message to both P_1 and P_2.
   - Increment P_0's clock: $\langle 2, 1, 0 \rangle$.
   - Timestamp for $b = \langle 2, 1, 0 \rangle$.
4. **Event** g (on P_2):
   - P_2 receives the message from d.
   - Update P_2's clock: $\max(\langle 0, 0, 0 \rangle, \langle 0, 1, 0 \rangle) = \langle 0, 1, 0 \rangle$.

- Increment P_2's clock: $\langle 0, 1, 1 \rangle$.
- Timestamp for $g = \langle 0, 1, 1 \rangle$.

5. **Event** h (on P_2):
   - P_2 sends a message to P_0.
   - Increment P_2's clock: $\langle 0, 1, 2 \rangle$.
   - Timestamp for $h = \langle 0, 1, 2 \rangle$.
6. **Event** e (on P_1):
   - P_1 receives the message from h.
   - Update P_1's clock: $\max(\langle 0, 1, 0 \rangle, \langle 0, 1, 2 \rangle) = \langle 0, 1, 2 \rangle$.
   - Increment P_1's clock: $\langle 0, 2, 2 \rangle$.
   - Timestamp for $e = \langle 0, 2, 2 \rangle$.
7. **Event** f (on P_1):
   - P_1 receives the message from b.
   - Update P_1's clock: $\max(\langle 0, 2, 2 \rangle, \langle 2, 1, 0 \rangle) = \langle 2, 2, 2 \rangle$.
   - Increment P_1's clock: $\langle 2, 3, 2 \rangle$.
   - Timestamp for $f = \langle 2, 3, 2 \rangle$.
8. **Event** c (on P_0):
   - P_0 receives the message from h.
   - Update P_0's clock: $\max(\langle 2, 1, 0 \rangle, \langle 0, 1, 2 \rangle) = \langle 2, 1, 2 \rangle$.
   - Increment P_0's clock: $\langle 3, 1, 2 \rangle$.
   - Timestamp for $c = \langle 3, 1, 2 \rangle$.
9. **Event** i (on P_2):
   - P_2 receives the message from b.
   - Update P_2's clock: $\max(\langle 0, 1, 2 \rangle, \langle 2, 1, 0 \rangle) = \langle 2, 1, 2 \rangle$.
   - Increment P_2's clock: $\langle 2, 1, 3 \rangle$.
   - Timestamp for $i = \langle 2, 1, 3 \rangle$.

Final Vector Timestamps

- $a : \langle 1, 1, 0 \rangle$ (receive from d)
- $b : \langle 2, 1, 0 \rangle$ (send to P_1 and P_2)

- $c : \langle 3, 1, 2 \rangle$ (receive from h)
- $d : \langle 0, 1, 0 \rangle$ (send to P_0 and P_2)
- $e : \langle 0, 2, 2 \rangle$ (receive from h)
- $f : \langle 2, 3, 2 \rangle$ (receive from b)
- $g : \langle 0, 1, 1 \rangle$ (receive from d)
- $h : \langle 0, 1, 2 \rangle$ (send to P_0 and P_1)
- $i : \langle 2, 1, 3 \rangle$ (receive from b)

(b) Based on the vector clock values, which events are "concurrent" with event b and why? [POINTS = 4]

Definition of Concurrent Events: Two events are **concurrent** if their vector clocks are **incomparable**. This means that neither clock is less than or greater than the other in element-wise comparison.

- **Event b:** $\langle 2, 1, 0 \rangle$
  We need to compare b's vector clock with other events.

Comparison with Other Events:

1. **Event a:** $\langle 1, 1, 0 \rangle$
   - Compare $b = \langle 2, 1, 0 \rangle$ and $a = \langle 1, 1, 0 \rangle$:
     - $b > a$ because $2 > 1$ and all other elements are $\geq$.
   - **Not concurrent**.
2. **Event c:** $\langle 3, 1, 2 \rangle$
   - Compare $b = \langle 2, 1, 0 \rangle$ and $c = \langle 3, 1, 2 \rangle$:
     - $c > b$ because $3 > 2$ and $2 > 0$.
   - **Not concurrent**.
3. **Event d:** $\langle 0, 1, 0 \rangle$
   - Compare $b = \langle 2, 1, 0 \rangle$ and $d = \langle 0, 1, 0 \rangle$:
     - $b > d$ because $2 > 0$.
   - **Not concurrent**.
4. **Event e:** $\langle 0, 2, 2 \rangle$
   - Compare $b = \langle 2, 1, 0 \rangle$ and $e = \langle 0, 2, 2 \rangle$:
     - Neither $b > e$ nor $e > b$, since $b[0] > e[0]$ but $e[1] > b[1]$.
   - **Concurrent**.
5. **Event f:** $\langle 2, 3, 2 \rangle$
   - Compare $b = \langle 2, 1, 0 \rangle$ and $f = \langle 2, 3, 2 \rangle$:

- $f > b$ because $3 > 1$ and $2 > 0$.

- **Not concurrent**.

6. **Event** g**:** $\langle 0, 1, 1 \rangle$
   - Compare $b = \langle 2, 1, 0 \rangle$ and $g = \langle 0, 1, 1 \rangle$:
     - Neither $b > g$ nor $g > b$, since $b[0] > g[0]$ but $g[2] > b[2]$.

   - **Concurrent**.
7. **Event** h**:** $\langle 0, 1, 2 \rangle$
   - Compare $b = \langle 2, 1, 0 \rangle$ and $h = \langle 0, 1, 2 \rangle$:
     - Neither $b > h$ nor $h > b$, since $b[0] > h[0]$ but $h[2] > b[2]$.

   - **Concurrent**.
8. **Event** i**:** $\langle 2, 1, 3 \rangle$
   - Compare $b = \langle 2, 1, 0 \rangle$ and $i = \langle 2, 1, 3 \rangle$:
     - $i > b$ because $3 > 0$.

   - **Not concurrent**.

Final Answer: Concurrent Events with b The events concurrent with **event** b ($\langle 2, 1, 0 \rangle$) are:
- **Event** e**:** $\langle 0, 2, 2 \rangle$
- **Event** g**:** $\langle 0, 1, 1 \rangle$
- **Event** h**:** $\langle 0, 1, 2 \rangle$

These events are concurrent because their vector clocks are incomparable with b.

15 Question 15:

Consider the following execution timeline. Consider strict, sequential, linearizable, and causal consistency models. For each model, explain why its requirements are met or not met.

[POINTS = 4]

P1:	W(x)a		W(x)c		
P2:		R(x)a	W(x)b		
P3:		R(x)a		R(x)c	R(x)b
P4:		R(x)a		R(x)c	R(x)b

Analysis of Consistency Models We analyze whether the given execution satisfies the requirements for **strict**, **sequential**, **linearizable**, and **causal consistency**.

1. Strict Consistency

- **Definition**: All reads return the value of the most recent write, and time is globally synchronized.
- **Analysis**:

- $P2$ reads $R(x)a$ after $W(x)a$, which is correct.
- $P3$ and $P4$ read $R(x)a$, $R(x)c$, and $R(x)b$, but the writes $W(x)c$ and $W(x)b$ appear out of order. Reads do not align with the latest writes.

- **Conclusion**: **Not satisfied** because the reads do not reflect the global order of writes.

2. Sequential Consistency

- **Definition**: All processes see the same global order of operations, though the order may not match real-time execution.
- **Analysis**:
  - A valid sequential order can be:
    $W(x)a \rightarrow R(x)a \rightarrow W(x)c \rightarrow R(x)c \rightarrow W(x)b \rightarrow R(x)b$.
  - $P3$ and $P4$ observe the same order for their reads ($R(x)a \rightarrow R(x)c \rightarrow R(x)b$).
  - The reads and writes can be serialized in a consistent order visible to all processes.
- **Conclusion**: **Satisfied** because a valid global order exists, even if it doesn't match real-time execution.

3. Linearizability

- **Definition**: Operations must appear instantaneous, preserving real-time order of events.
- **Analysis**:
  - $W(x)a$ occurs before $R(x)a$, which is correct.
  - $W(x)c$ occurs after $R(x)a$, but $R(x)b$ observes $W(x)b$ despite $W(x)c$ being more recent.
  - Reads do not reflect real-time causality for all processes.
- **Conclusion**: **Not satisfied** because reads do not consistently reflect real-time order.

4. Causal Consistency

- **Definition**: Writes that are causally related must be observed in the same order by all processes. Independent writes can be seen in different orders.
- **Analysis**:
  - $P2$'s $R(x)a$ occurs after $W(x)a$, and $W(x)b$ occurs after $R(x)a$. This is causally consistent.
  - $P3$ and $P4$ see $R(x)a \rightarrow R(x)c \rightarrow R(x)b$, which is causally consistent because there are no causal dependencies violated.
- **Conclusion**: **Satisfied** because all causally related operations are observed in the same order by all processes.

Final Summary

Model	Requirements Met?	Reason
Strict	**No**	Reads do not reflect the most recent write globally.
Sequential	**Yes**	A valid global order exists that is consistent across all processes.
Linearizable	**No**	Reads do not consistently reflect real-time ordering of writes.
Causal	**Yes**	All causally related writes are observed in the same order by all processes.

16 Question 16:

A core concept in ACID transactions is Consistency. Another student in your class, Todd, says that data is defined to be consistent according to transactional serializability if and only if every copy of a replicated data set reports the same value when you read them all during a transaction. Explain why you agree or disagree.

[POINTS = 2]

I **disagree** with Todd's statement.

- **Reason**: Consistency in ACID transactions is not about every copy of a replicated data set reporting the same value at all times. Instead, **consistency** refers to the property that a database remains in a valid state (satisfying all constraints and rules) before and after a transaction.
- **Key points**:
  - **Transactional serializability** ensures that the outcome of transactions is equivalent to executing them sequentially in some order, not necessarily that all replicas must always have identical values during a transaction.
  - Replication consistency (where all replicas show the same value) is a property of distributed systems and not necessarily tied to ACID transactional serializability. This is typically handled by **synchronization protocols** like strong consistency or eventual consistency.

Thus, Todd confuses **replication consistency** with the broader definition of **transactional consistency** in ACID.

17 Question 17:

A sharded database uses two-phase commit to ensure that either all shard servers commit their part of each transaction, or none of them do. A database client executing a transaction sends the transaction's puts and gets to the shard servers, and then uses a transaction coordinator (TC) to execute two-phase commit for the transaction.

(a) The TC sends a PREPARE message to each participant (each shard server that is involved in the transaction). Each participant replies with YES. As all participants answer YES, the TC decides to send a COMMIT message. Unfortunately, TC sends a COMMIT message to one of a transaction's multiple participants, but the TC crashes before sending any more messages. What should the TC do after it reboots?

[POINTS = 2]

After the Transaction Coordinator (TC) reboots, it must **recover the transaction state** and ensure the transaction is correctly completed according to the **Two-Phase Commit (2PC) protocol**.

Steps the TC should take:

1. **Consult the Transaction Log:**
   - The TC should check its transaction log to determine the state of the transaction before the crash.
   - Since it decided to send a `COMMIT` message before crashing, the `COMMIT` decision should have been logged.
2. **Resend the `COMMIT` message:**
   - Once the TC confirms from the log that the decision was to commit, it should resend the `COMMIT` message to all participants that did not receive it before the crash.
   - Participants who already received the `COMMIT` message will ensure the transaction has been committed on their side (idempotency of commit ensures this step is safe).
3. **Await Acknowledgments:**
   - The TC should wait for acknowledgments from all participants to confirm that the commit has been applied across all shard servers.

Why This Is Correct: The 2PC protocol ensures atomicity by logging the decision (`PREPARE` or `COMMIT`) before taking action. After a crash, the TC uses the log to recover and complete any incomplete transactions. Re-sending the `COMMIT` ensures that all participants complete the transaction, even if the crash interrupted communication.

Paxos

1. **Paxos vs. 2PC failure handling:**
   Paxos tolerates coordinator crashes; 2PC may block.
2. **3PC vs. 2PC:**
   3PC prevents blocking with a pre-commit phase.
3. **Safety:**
   Both ensure safety, but Paxos handles crashes better.
4. **Message complexity:**
   Paxos requires more rounds than 2PC.
5. **Network partitions:**
   Paxos works in the majority partition; 2PC may block.
6. **2PC recovery:**
   Participants may query each other; manual intervention may be needed.
7. **Paxos vs. 2PC use case:**
   Paxos is ideal for highly available systems.
8. **Leader election in Paxos:**
   A leader drives proposals; crashes trigger re-election.
9. **Multi-Paxos:**
   Reuses a leader to reduce message rounds.
10. **Byzantine faults:**
    Neither Paxos nor 2PC handles them; PBFT is needed.