CSS451: Cloud Computing

Midterm Mock Exam

curated by The Peanuts

Name . . . . . Nonprawich I. . . . . ID . . . . . 6622772422 . . . . . Section . . . . . 1 . . . . . Seat No . . . . . G36 . . . . . .

Conditions: Closed Book

Directions:

1. This exam contains 16 pages (including this one). If yours has fewer, congratulations — you have discovered the distributed truncation problem.
2. Write your name and ID clearly at the top.
3. Show your reasoning for short and long answer questions. This is not a black box, your thought process matters.
4. Answers must be written in English. Pseudocode is acceptable where explicitly permitted. YAML, Dockerfile, and Scala will not be graded.
5. This is a **closed book** exam. You may not use phones, laptops, or your neighbor's cluster of neurons.
6. For Part V (Case Study), choose **exactly one** scenario. Attempting more than one will result in zero points for the entire part. Indecision is not fault-tolerant.
7. If a question is ambiguous, state your assumptions clearly and proceed. Partial credit is awarded based on reasoning quality.

For solution, ***click here****.*

Part I: True / False (10 Points, 1 pt each)

*Write **TRUE** or **FALSE** in the blank provided. If a statement is partially correct, it is **FALSE**.*

1.1 False Spark achieves fault tolerance primarily through data replication across multiple nodes, similar to HDFS's default replication factor of 3.

It uses lineage information, a record of the sequence transformations (DAG) that produced each RDD partition

1.2 True In an RDD lineage graph, transformations are executed lazily — they are not computed until an action such as `count()` or `collect()` is called.

This allows Spark to optimize the execution DAG before running.

1.3 True A `join` operation on two RDDs of types `RDD[(K, V1)]` and `RDD[(K, V2)]` produces an output of type `RDD[(K, (V1, V2))]`.

1.4 False In Spark Streaming, computation is performed as a continuous event-driven data flow, not as small batch jobs.

Uses micro-batch (discretized stream) processing: the live stream is cut into small time-interval batches, each processed as a standard RDD batch job.

Flink/Storm/Apache Storm

1.5 True Apache Mesos uses a two-level scheduling model in which the Mesos master offers resources to framework schedulers, and the frameworks decide how to use those resources.

accept, reject

1.6 True Omega uses optimistic concurrency control, meaning multiple schedulers can read the shared cluster state simultaneously and propose resource allocations independently, with conflicts resolved at commit time.

1.7 True In Kubernetes, a Pod is the smallest deployable unit, and a single Pod can contain more than one container.

Multiple containers in the same Pod share the same network namespace (IP address) and storage volumes.

1.8 False The `etcd` component in Kubernetes is responsible for scheduling Pods onto available worker nodes.

distributed key-value store that stores cluster configuration and state
it's the "source of truth" for the cluster. Scheduling Pods onto nodes is the kube-scheduler, not etcd.

1.9 False In GFS (Google File System), the master server is directly involved in transferring chunk data from a chunkserver to the client during a read operation.

After the client queries the master for chunk metadata, the master's role is complete. The actual data transfer happens directly between the client and the chunk server

1.10 True Para-virtualization requires modifications to the guest operating system, whereas full virtualization can run an unmodified guest OS.

Part II: Fill in the Blanks (10 Points, 1 pt each)

Fill in each blank with the most appropriate word or phrase. Choose from the word bank below where indicated, or write your own answer.

Word Bank (not all words will be used):

~~*cogroup*~~ ~~*lineage*~~ ~~*DStream*~~ ~~*etcd*~~ *kubelet* *kube-proxy* *straggler* ~~*combiner*~~ ~~*chunkserver*~~ *master* ~~*optimistic*~~ *pessimistic* ~~*immutable*~~ *in-memory* ~~*partitioned*~~ *two-level* *shared-state*

2.1 In Spark, an RDD is a immutable, partitioned collection of objects that can be processed in parallel.

2.2 When a partition of an RDD is lost due to a node failure, Spark recovers it by re-executing the chain of transformations recorded in the RDD's lineage graph.

2.3 The cogroup operation in Spark groups the values of two key-value RDDs by key, returning an RDD of type `(K, (Iterable[V1], Iterable[V2]))`.

2.4 In Spark Streaming, a discretized stream is represented as a DStream (abbreviation), which is a sequence of RDDs generated at regular time intervals.

2.5 In MapReduce, a combiner is a mini-reducer that runs on the mapper output locally, reducing the volume of data transferred across the network before the shuffle phase.

2.6 Mesos employs a two-level scheduling architecture, where the master exposes available resources to framework schedulers via *resource offers*.

2.7 Omega differs from Mesos primarily in its use of optimistic concurrency control, allowing all schedulers to operate on a full copy of cluster state simultaneously.

2.8 In Kubernetes, the etcd is a distributed key-value store that persists the entire cluster configuration and state.

2.9 The kubelet agent runs on each worker node in Kubernetes and is responsible for ensuring that the containers described in Pod specifications are running and healthy.

The *kubelet* agent runs on every worker node. It watches the API server for Pod specs assigned to its node and ensures the described containers are running and healthy, reporting node/pod status back to the control plane.

2.10 In GFS, the component that holds actual file data blocks and serves data directly to clients (after initial metadata lookup) is called the chunksever.

Part III: Multiple Choice (10 Points, 1 pt each)

*Choose the **single best** answer for each question. Circle the letter of your choice.*

3.1. Which of the following best explains why Spark is significantly faster than Hadoop MapReduce for iterative machine learning algorithms?

a) Spark stores intermediate RDDs in memory, avoiding repeated disk I/O between iterations.

b) Spark uses a more efficient sorting algorithm during the shuffle phase.

c) Spark parallelizes the reduce phase using a binary tree reduction strategy.

d) Spark compresses intermediate data using LZ4 before writing to HDFS.

3.2. Given `rdd1: RDD[(K, V1)]` and `rdd2: RDD[(K, V2)]`, what is the output type of `rdd1.join(rdd2)`?

a) `RDD[(K, (V1, V2))]`

b) `RDD[(K, V1, V2)]`

c) `RDD[((K, V1), (K, V2))]`

d) `RDD[(K, (Iterable[V1], Iterable[V2]))]`

3.3. In Spark Streaming, the fundamental processing model is best described as:

a) A continuous event-driven pipeline where each record is processed as it arrives.

b) A push-based model where data is sent directly to Spark executors by producers.

c) A series of small, deterministic batch jobs over micro-batch intervals (DStreams).

d) A persistent stateful loop that blocks until a defined window is full.

3.4. Which statement correctly distinguishes Omega from Apache Mesos?

a) Omega uses optimistic concurrency control on a shared cluster state; Mesos uses pessimistic control by locking resources via offers.

b) Mesos uses shared-state scheduling; Omega uses two-level scheduling.

c) Omega is a centralized monolithic scheduler; Mesos is fully decentralized. Omega serializes all scheduling decisions through a single master.

d) Mesos allows all frameworks to simultaneously modify the cluster state;

3.5. In Apache Mesos, the framework can _____ a resource offer from the Mesos master. Which word correctly completes the sentence, and what is the implication?

a) *Preempt* — the framework can forcibly reclaim resources from other frameworks.

b) *Accept* — the framework must use all offered resources immediately.

c) *Reject* — the framework may decline resources that do not suit its constraints,

d) *Partition* — the framework can split offers among multiple tasks. enabling other frameworks to receive them.

3.6. Which Kubernetes component acts as a load balancer for incoming traffic, routing requests to the correct Pod?

a) kube-proxy

b) kubelet

c) Scheduler

d) etcd

3.7. In HDFS, which node is responsible for managing file namespace, access control, and metadata, but does *not* store actual file data?

a) NameNode

b) DataNode

c) JobTracker

d) SecondaryNameNode

3.8. Which of the following correctly describes a DAG (Directed Acyclic Graph) in the context of Apache Spark?

a) A logical execution plan representing the sequence of transformations from input to output,

b) A graph that represents the physical storage layout of data blocks across cluster nodes. used by the Spark scheduler to optimize and stage tasks.

c) A replication topology used by the master to distribute tasks to executors.

d) A graph that models network communication between driver and worker nodes.

3.9. In the MapReduce programming model, data locality refers to:

a) Moving computation to the node where data resides, rather than moving data to computation.

b) Replicating data to all mapper nodes before processing begins.

c) Storing all output data on the same node as the master for fast retrieval.

d) Ensuring all reducers are co-located on the same rack to minimize network traffic.

3.10. A framework running on Mesos holds onto acquired resources without using them, preventing other frameworks from accessing those resources. This behavior is best described as:

a) Starvation

b) Thrashing

c) Resource Hoarding

d) Contention

Part IV: Short and Long Answer (20 Points)

Question 4.1 (4 Points)

Consider the following two RDDs:

```
orders: RDD[(CustomerID, OrderAmount)]
customers: RDD[(CustomerID, CustomerName)]
```

With the following data:

orders	customers
(101, 250.0)	(101, "Alice")
(102, 80.0)	(102, "Bob")
(101, 430.0)	(103, "Carol")
(104, 150.0)	

(a) Perform a `join` operation on `orders` and `customers` by `CustomerID`. Write the complete output as a set of key-value pairs. **(2 pts)**

(101, (250.0, "Alice"))
(101, (430.0, "Alice"))
(102, (80.0, "Bob"))

#

(b) Perform a `cogroup` operation on the same two RDDs. Write the complete output, clearly showing each key and its grouped iterables for both RDDs. Include keys that appear in *only one* of the RDDs. **(2 pts)**

(101, ([250.0, 430.0], ["Alice"])
(102, ([80.0], ["Bob"]))
(103, ([], ["Carol"]))
(104, ([150.0], []))

#

Question 4.2 (3 Points)

A company wishes to count the number of purchases made per product category from a large transaction log. Each line of the log contains: `<TransactionID> <Category> <Amount>`

(a) Describe (in words) what the **Map** function should output for each input record. **(1 pt)**

For each record <TransactionID> <Category> <Amount>:

Map output: (Category, 1)

The mapper extracts the Category field and emits it as the key with a value of 1 (indicating one purchase in the category). TransactionID and Amount are discarded (not needed for counting)

```
Input:  "TXN001 Electronics 199.99"
Output: ("Electronics", 1)
```

(b) Describe (in words) what the **Reduce** function should do. **(1 pt)**

The reducer receives all values for a given key (category) and sums them:

Reduce: (Category, [1, 1, 1, ...]) ⟶ (Category, count)

For each key (Category), the reducer receives an iterable of 1s (one per purchase) and reduce them by summing, producing the total purchase count per category

```
Input:  ("Electronics", [1, 1, 1, 1, 1])
Output: ("Electronics", 5)
```

(c) Could a **Combiner** be used here? Justify your answer briefly. **(1 pt)**

Yes, a combiner can be used.

The combiner applies the same summation logic as the reducer, locally aggregating the mapper's (Category, 1) outputs on each mapper node before they are sent across the network

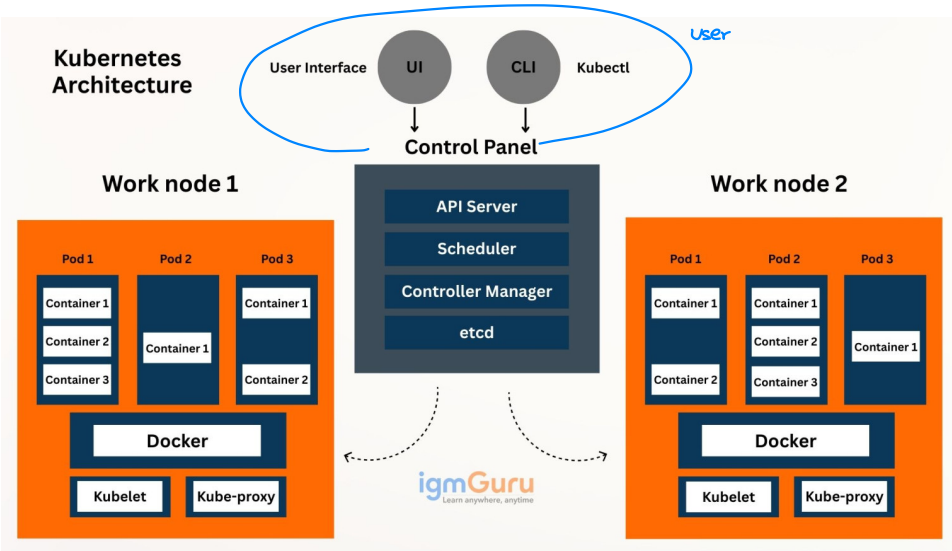
Benefit: Instead of sending 1,000 individual ("Electronics", 1) records across network, the combiner might reduce this to a single ("Electronics", 1000) per mapper, significantly reducing shuffle cost

Question 4.3 (5 Points)

Draw and label a diagram of a Kubernetes cluster that includes the following components and clearly shows how they interact:

- Control Plane: API Server, Scheduler, Controller Manager, `etcd`
- Worker Node(s): kubelet, kube-proxy, Pod(s) with container(s)
- An external user/client issuing a `kubectl` command

Indicate with **arrows** the direction of communication between components (e.g., user → API Server → Scheduler → kubelet). Brief labels on arrows (e.g., "schedules Pod", "watch loop") are encouraged.



1. **User → API Server:** All commands (`kubectl apply`, `scale`, etc.) go through the API Server, which is the single entry point to the control plane.
2. **API Server etcd:** The API Server is the only component that reads/writes to etcd directly. etcd stores the desired state (e.g., "3 replicas of nginx").
3. **API Server → Scheduler:** When a new Pod is created with no node assigned, the Scheduler watches the API Server, selects the best node based on resource availability, and writes the node binding back to the API Server (which stores it in etcd).
4. **API Server → Controller Manager:** The Controller Manager watches the API Server for state mismatches (e.g., desired replicas = 3, current = 2) and takes corrective actions (creates new Pods), reconciling desired vs. actual state continuously.
5. **API Server → kubelet:** The kubelet on each worker node watches the API Server for Pod specs assigned to its node. It starts/stops containers via the container runtime (Docker, containerd) accordingly, and reports status back to the API Server.
6. **kube-proxy → Pods:** kube-proxy maintains iptables/IPVS rules on each node to load-balance Service traffic to the correct Pod IP:port endpoints.

Question 4.4 (4 Points)

An engineering team is considering whether to adopt Mesos or Omega as the scheduling backbone for a mixed workload cluster (batch analytics + real-time services + ML training).

(a) State **two key limitations** of Apache Mesos that motivated the design of Omega. **(2 pts)**

1. Pessimistic concurrency control (serialized offers)

   Mesos offers resources to one framework at a time. While a framework is evaluating an offer, no other framework can use or see those resources. This is effectively a "resource lock" for the duration of a scheduling decision. For frameworks with complex scheduling logic, this can introduce significant latency and overall scheduling throughput

2. Information hiding

   A framework scheduler in Mesos only sees the resources currently offered to it, not the full cluster state. It cannot see what resources other frameworks are using, cannot perform global optimizations.

(b) Explain how Omega's **optimistic concurrency control** addresses those limitations. What is the main risk or trade-off introduced by this approach? **(2 pts)**

Omega gives each scheduler a full snapshot of the entire cluster state and allows all schedulers to operating in parallel without blocking each other. Schedulers make decisions independently using their local copy and submit changes as atomic transactions to the shared state.

Eliminate 的是 2 problem over Mesos

Trade off: If multiple schedulers claim the same resource simultaneously, the shared-state store accepts only one transaction; the other are rejected and must resync and retry.

Question 4.5 (4 Points)

(a) In GFS, the master is **not** involved in one particular step of a read operation. Identify that step and explain why the architecture is designed this way. **(2 pts)**

The step: Actual data transfer (from chunkserver to client)

GFS Read Process:

1. Client send filename + byte range to the master.
2. Chunk returns the chunk handle and the locations (chunkservers) that hold replcas of that chunk.
3. Client contacts a chunkserver directly (closest replica) and reads the data

The master is not involved in the step?

If the master was involved in every data transfer, it would become a bottleneck — a single master handling terabytes or petabytes of data reads across thousands of concurrent clients would saturate quickly. By keeping the master out of the data path (master only handles metadata, not data), GFS scales the read throughput horizontally across all chunkservers while keeping the master load manageable.

(b) HDFS uses a **NameNode** and multiple **DataNodes**. What is the key single point of failure risk in HDFS, and what mechanism does HDFS use to mitigate it? **(2 pts)**

HDFS has a single NameNode that stores the entire file system namespace in memory
If the NameNode crashes and is unrecoverable, all metadata is lost. The cluster cannot locate any data blocks even through the DataNodes still hold the raw data. This is the classic single point of failure problem.

Mitigation mechanism:

- Secondary NameNode
- Standby NameNode

Part V: Case Study

(20 Points)

INSTRUCTION: Choose **exactly ONE** scenario from the five options below. Attempting more than one will result in **zero points** for this entire part.

Circle your chosen scenario number before you begin writing.

Your answer will be graded on: *correctness of concepts* (10 pts), *justification and reasoning* (6 pts), and *clarity of presentation* (4 pts).

Scenario A — Kubernetes Deployment Under Failure

Context: A fintech startup deploys a payment processing microservice on a Kubernetes cluster consisting of 1 Control Plane node and 4 Worker nodes. The payment service runs as a Deployment with `replicas: 3`.
The following sequence of events occurs:

1. A developer runs `kubectl scale deployment payment-service --replicas=5`.
2. Worker Node 2, which hosted 2 running Pods, suddenly crashes.
3. A new Worker Node 5 joins the cluster.

Your Task:

(a) For each event (i–iii), identify **which Kubernetes component(s)** detect the change, make the scheduling decision, and take corrective action. Explain the role of each component in that specific event.

(b) Draw a simplified diagram of the cluster state **after all three events**, showing which Pods run on which nodes (you may use boxes and labels).

(c) The Control Plane node also fails shortly after. What is the impact on the *currently running* Pods? Can the cluster still serve traffic? Explain your answer in terms of Kubernetes architecture.

Scenario B — Mesos vs. Omega: Cluster Redesign

Context: You manage a shared cluster used by four teams:

- **Team Alpha:** Latency-critical web services (must respond within 50ms)
- **Team Beta:** Batch analytics jobs (Spark, runtime hours to days)
- **Team Gamma:** GPU-based ML training (long-running, resource-intensive)
- **Team Delta:** CI/CD pipelines (short-lived, bursty, unpredictable)

Current problems:

- Static quotas lead to resource waste when teams are idle.
- Batch jobs starve latency-critical services during peak hours.
- The centralized scheduler is a bottleneck — scheduling latency is increasing.

Your Task:

(a) Choose **either Mesos or Omega** as the new scheduling backbone. Justify your choice based on the workload characteristics of the four teams. Explicitly argue why the other system would be *less suitable*.

(b) Explain concretely how your chosen system addresses: (i) resource waste from static quotas, (ii) starvation of high-priority services, and (iii) scheduler scalability/bottleneck.

(c) Identify **one remaining limitation** of your chosen system that this cluster design does *not* solve, and propose a mitigation strategy.

We'll use Omega

The four workloads present diverse characteristics :

- Team Alpha (web services) requires low latency resource allocation, they cannot wait for serialized resource offer cycles
- Team Beta (batch Spark) runs long jobs that benefit from seeing the full cluster state to make optimal placement decisions.
- Team Gamma (GPU ML) requires gang scheduling (all GPUs must be allocated together for training runs) A feature Mesos cannot support well due to information hiding.
- Team Delta (CI/CD) is bursty and unpredictable, requiring rapid, flexible scheduling.

Why Omega is more suitable: Omega gives each team's scheduler **full visibility into the cluster state** and allows all schedulers to **operate in parallel**. This means: Alpha's scheduler can instantly check GPU availability for Gamma and avoid resource conflicts. Gamma can plan gang-scheduled allocations with full knowledge of current usage. Delta's bursty needs are met without waiting for serialized offer cycles.

Why Mesos is less suitable: Mesos's pessimistic, serialized offer model creates scheduling latency. Gamma's gang scheduling is nearly impossible under Mesos — it would need to receive offers for all required GPUs simultaneously, which Mesos cannot guarantee. Alpha's latency-critical services would suffer from Mesos's offer serialization delays.

b)

(i) Resource waste from static quotas:
Omega has no static quotas. Each scheduler sees the full cluster and can claim any available resource. Idle resources from one team become immediately visible to other schedulers and can be allocated dynamically. A common priority scale ensures high-priority jobs (Alpha's services) preempt lower-priority ones when resources are scarce.

(ii) Starvation of high-priority services:
Omega resolves conflicts using a **priority system**. If Alpha (priority: high) and Beta (priority: low) both want the same resource and conflict at commit time, Alpha's transaction wins. Beta retries with updated state and avoids the claimed resource. This prevents indefinite starvation of high-priority services.

(iii) Scheduler bottleneck / scalability:
Because all schedulers run in parallel and independently (no central lock), scheduling throughput scales with the number of schedulers, not bottlenecked at a single master. Each team's scheduler can make decisions simultaneously without blocking others.

(c) Remaining limitation and mitigation

Limitation: High conflict rate under peak load
When all four teams have bursty demand simultaneously, many schedulers may compete for the same resources (especially the few GPU nodes for Gamma). This leads to many failed transactions and retry cycles, temporarily slowing effective scheduling and potentially re-introducing latency.

Mitigation:
Implement **incremental scheduling** (schedule small batches of tasks per transaction rather than entire jobs at once) to reduce transaction size and thus conflict probability. Additionally, apply **resource partitioning hints** (soft affinity rules) so each scheduler prefers different subsets of the cluster, reducing overlap in claims.

Scenario C — Spark Streaming Pipeline Design

Context: A logistics company wants to build a real-time shipment tracking system. Sensors on delivery trucks emit GPS coordinates every 5 seconds. The data is ingested into Apache Kafka and must be processed to:

1. Detect trucks that have been stationary for more than 10 minutes (possible breakdowns).
2. Compute the average speed per truck over a sliding window of 15 minutes.
3. Alert a monitoring dashboard within 30 seconds of anomaly detection.

The engineering team proposes using **Spark Streaming**.
Your Task:

(a) Describe the Spark Streaming processing model (DStream, micro-batch), and explain why it is suitable or unsuitable for the 30-second latency requirement in requirement (iii). Would standard Spark Streaming or Structured Streaming be more appropriate? Justify.

(b) For requirement (ii), describe how you would configure a **windowed operation** in Spark Streaming. Define the window duration, slide interval, and what aggregation would be performed. Draw the window timeline for the first 30 minutes of data.

(c) If a Spark executor node crashes mid-window, how does Spark ensure the window computation is eventually correct? Reference the fault tolerance mechanism.

(a) Spark Streaming model and latency suitability

Spark Streaming model (DStream / micro-batch):
Spark Streaming chops the live input stream into discrete time intervals (micro-batches), typically 0.5–10 seconds. Each micro-batch is processed as a standard Spark RDD job. Results are produced at the end of each batch interval. This is called a **Discretized Stream (DStream)**.
Suitability for 30-second latency requirement:
Standard Spark Streaming is suitable for a 30-second SLA. With a batch interval of, say, 5–10 seconds, anomalies detected within one batch would be reported with at most 10 seconds of processing latency — well within 30 seconds.
However, for the lowest possible latency and more sophisticated stateful operations, **Structured Streaming** (Spark 2.x+) is more appropriate because:

- It supports *continuous processing mode* (down to ~1ms latency).
- It has native watermarking for handling late data in windows.
- It offers a higher-level, SQL-friendly API with stronger exactly-once guarantees.

Recommendation: Use **Structured Streaming** for production; standard Spark Streaming is acceptable for this use case given the 30-second requirement.

(b) Windowed operation configuration

For 15-minute sliding average speed per truck:

- **Window duration:** 15 minutes (the time range over which speed is averaged)
- **Slide interval:** 5 minutes (how often the window is computed / moves forward)
- **Aggregation:** For each truck ID: average speed = sum of speeds in window / count of readings

Pseudocode concept:

```
speedStream
  .map(record => (truckID, speed))
  .reduceByKeyAndWindow(
    (a, b) => a + b,        // sum speeds
    windowDuration = 15min,
    slideDuration  = 5min
  )
  .mapValues(totalSpeed => totalSpeed / countInWindow)
```

Window timeline (first 30 minutes, slide = 5 min):

Window	Covers
W1	t=0 to t=15 min
W2	t=5 to t=20 min
W3	t=10 to t=25 min
W4	t=15 to t=30 min

Each window overlaps with the previous one by 10 minutes (window 15 − slide 5 = 10 min overlap).

(c) Fault recovery mid-window

Spark's fault tolerance mechanism: Lineage + Checkpointing

1. Each DStream batch's RDD has a **lineage graph** tracking which input data produced it (from Kafka, in this case).
2. If an executor crashes mid-window, the lost RDD partitions are recomputed by re-reading the relevant Kafka offsets (Spark + Kafka integration supports offset tracking).
3. For stateful operations (like sliding windows that accumulate state across batches), **checkpointing** is configured: Spark periodically saves the window state to HDFS. On recovery, Spark restores from the last checkpoint and re-processes only the batches since the checkpoint.

Result: The window computation is eventually correct because the lineage guarantees that all data within the window interval is fully accounted for after recovery. The 30-second latency SLA may be momentarily violated during recovery but resumes normally afterward.

Scenario D — RDD Lineage and Fault Recovery

Context: A data science team builds the following Spark pipeline to analyze customer purchase data:

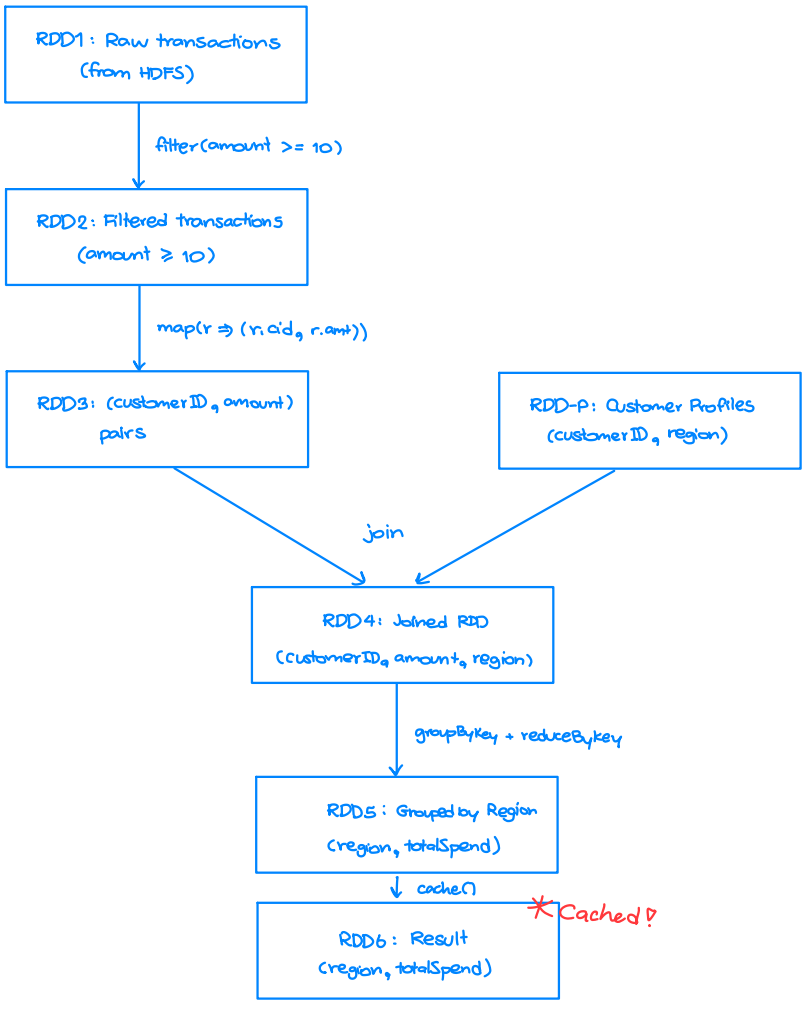
1. Load raw transaction records from HDFS into an RDD.
2. Filter out records with amount < 10 (noise).
3. Map each record to `(customerID, amount)` pairs.
4. Join with a customer profile RDD `(customerID, region)`.
5. Group by region and compute total spend per region.
6. Cache the result for downstream reporting queries.

Your Task:

(a) Draw the complete **RDD lineage graph** (DAG) for this pipeline. Label each node with the RDD name and each edge with the transformation applied. Clearly mark which RDD is cached.

(b) Suppose step 4's joined RDD loses 2 of its 8 partitions due to a worker failure. Explain step-by-step how Spark recovers *only those 2 partitions*. Which prior transformations must be re-executed? Which can be skipped?

(c) The team notices the pipeline is slow due to the join operation causing a large shuffle. Propose **two** optimizations they could apply to reduce shuffle cost, and explain the principle behind each.



(b) Recovery of 2 lost partitions from RDD4 (joined)

Assume 8 partitions total; 2 are lost from RDD4.
Step-by-step recovery:

1. Spark detects the lost partitions in RDD4 (joined) through lineage tracking.
2. Spark identifies that RDD4 depends on RDD3 (mapped) and RDD-P (profiles) via a `join` transformation.
3. Spark checks whether the parent partitions (RDD3, RDD-P) that produced the lost RDD4 partitions are still available or need recomputation.
4. If RDD3's corresponding partitions are also lost (e.g., same node failed): Spark traces back further to RDD2 (filtered) and RDD1 (raw data on HDFS). HDFS data is always available (replicated), so re-reading is possible.
5. Spark re-executes **only the chain for the 2 missing partitions**: `read from HDFS` → `filter` → `map` → `join` for the specific data shards that produced those 2 partitions.
6. The other 6 partitions of RDD4 are **not recomputed** — only the 2 lost ones.
7. After RDD4's 2 partitions are restored, the downstream RDD5 and RDD6 are also recomputed for those partitions if needed.

Key point: Lineage enables *fine-grained, partition-level recovery* — no full recomputation is needed.

Scenario E — Multi-Framework Cluster: Mesos in Practice

Context: A media company runs the following workloads on a single Mesos cluster (200 nodes):

Framework	Type	Characteristic
Hadoop MapReduce	Batch	Long-running, fault-tolerant
Apache Storm	Real-time streaming	Short tasks, low latency
Spark	Iterative ML	Memory-intensive, iterative

During peak hours, resource contention spikes. The ops team observes:

- Storm tasks are delayed because Hadoop is holding large resource slots.
- Spark is hoarding memory even between iterations.
- Mesos master scheduling latency has risen to 2 seconds per offer cycle.

Your Task:

(a) Identify which of the three problems above is caused by a **framework behavior** and which is caused by **Mesos architecture limitations**. Explain your reasoning for each.

(b) Mesos supports fine-grained and coarse-grained resource sharing modes. Explain the difference, and argue which mode would better address the Storm latency problem. What trade-off does your choice introduce?

(c) Suppose the team considers migrating from Mesos to Omega to solve the scheduler latency problem. Explain how Omega's architecture would reduce scheduling latency in this scenario. Would it introduce any new risks? Be specific.

(a) Framework behavior vs. Mesos architecture limitations

Problem 1: Storm tasks delayed because Hadoop holds large slots
Cause: Framework behavior (Hadoop) + partially Mesos architecture.
Hadoop (MapReduce) is coarse-grained: it tends to request large resource slots and hold them for the full job duration. This is Hadoop's scheduling behavior. However, Mesos's resource offer model does not force frameworks to release resources promptly — it *can* kill tasks but prefers to let frameworks manage their own lifetimes. Both are contributing factors; the root cause is Hadoop's coarse-grained usage pattern.
Problem 2: Spark hoarding memory between iterations
Cause: Framework behavior (Spark).
Spark caches RDDs in memory across iterations by design (this is its core feature). When Spark caches between iterations, it retains memory allocations even when not actively computing. This is a Spark-specific behavior, not a Mesos architecture issue. Mesos could address this via revocation (killing Spark tasks), but Spark is designed to hold memory as long as possible for performance.
Problem 3: Mesos master scheduling latency at 2 seconds per offer cycle
Cause: Mesos architecture limitation.
This is a direct consequence of Mesos's **pessimistic, serialized offer model**. The master processes one offer cycle at a time. As the number of frameworks and slaves grows, the offer cycle takes longer. This is a fundamental scalability bottleneck of the Mesos architecture — not fixable purely by framework behavior changes.

(b) Fine-grained vs. coarse-grained sharing for Storm latency

Definitions:

- **Coarse-grained:** Resources (e.g., entire machines) are allocated to one framework for the duration of a job. Simple, low overhead, but poor utilization.
- **Fine-grained:** Resources are allocated at the *task level* within a job. When a task finishes, its resources are returned to Mesos immediately for reuse.

For the Storm latency problem:
Fine-grained sharing would better address Storm's latency issue. If Hadoop is running in fine-grained mode, its resources are returned to Mesos as soon as each individual map/reduce task finishes. Storm can then immediately pick up these freed resources for new tasks, reducing the delay caused by Hadoop holding large static allocations.
Trade-off introduced:
Fine-grained sharing increases **scheduling overhead** — Mesos must process far more offer/response cycles (one per task completion rather than one per job). For Hadoop with millions of tasks, this can overload the Mesos master and ironically worsen the scheduling latency problem (Problem 3 above). There is thus a fundamental tension between fine-grained sharing and master scalability.

(c) Migrating to Omega to solve scheduling latency

How Omega reduces scheduling latency:
In Mesos, all framework scheduling decisions are serialized through the master's offer cycle. Under Omega's **shared-state architecture**:

1. Each of the three frameworks (Hadoop, Storm, Spark) gets its own independent scheduler with a full snapshot of cluster state.
2. All three schedulers run **simultaneously in parallel** — no waiting for each other's offer cycle.
3. Scheduling decisions are committed as atomic transactions; conflicts trigger retries.

This eliminates the single-scheduler bottleneck. Effective scheduling throughput scales with the number of parallel schedulers rather than being limited by a single master's processing speed.
New risks introduced:

1. **Conflict and retry overhead:** Under high contention (e.g., all three frameworks bursting simultaneously), many transactions will conflict and be rejected. Each rejection triggers a resync-and-retry cycle, which could re-introduce latency in extreme cases. The paper notes this is acceptable for typical workloads but may be problematic for very high-churn clusters.
2. **State synchronization overhead:** Each scheduler must keep its local copy of cluster state up to date. In a fast-changing cluster (many short-lived Storm tasks completing rapidly), the state can become stale quickly, increasing the probability of stale-read conflicts.
3. **Implementation complexity:** Each framework must be rewritten to use Omega's shared-state API and implement its own conflict resolution logic. This is a significant engineering investment compared to Mesos's simpler resource offer interface.