Write-Ahead Logs (WAL) in System Design & Kafka

1. What is a Write-Ahead Log (WAL)?

A **Write-Ahead Log (WAL)** is an **append-only log** used in databases, file systems, and distributed systems to guarantee **durability and recoverability**.

- Every change is written to the log **before** applying it to the main data store.
- If a crash happens, the system replays the WAL to restore consistency.

Where WAL is used

- **Databases:** PostgreSQL, MySQL InnoDB, Oracle → ACID durability.
- **File systems:** ext4, NTFS, ZFS (journaling).
- **Distributed systems:** Kafka, HDFS, Raft.
- **Caching systems:** Redis AOF.

---

2. WAL in Kafka

Kafka is essentially a **distributed WAL**. Every message produced to a topic partition is:

1. Written to the **leader broker's log** (append-only file).
2. Replicated to **follower brokers**.
3. Exposed to consumers **only after replication (High Watermark)**.

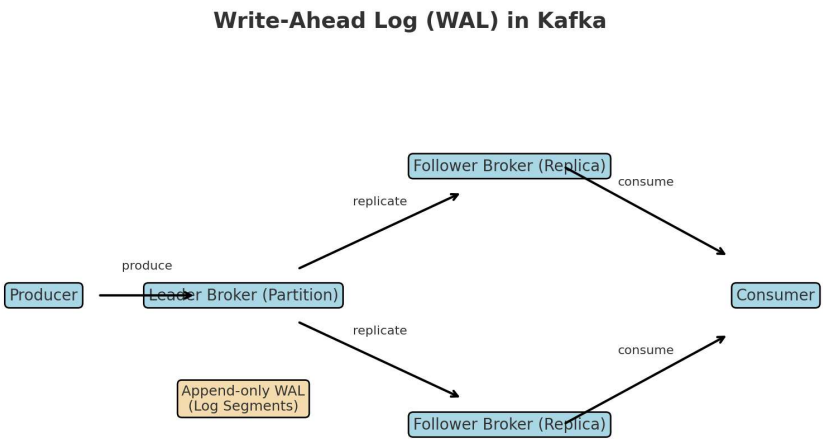
Flow

1. Producer → Leader broker.
2. Leader appends record to **WAL (log segment)** via **page cache**.

3. Record replicated to ISR followers, appended to their WAL.
4. Once all ISR replicas confirm, leader advances **High Watermark (HW)**.
5. Producer gets ack (if `acks=all`), and consumers can now read.

---

3. Kafka WAL Flow (Diagram)



Explanation:

- Producer sends records to leader broker.
- Leader writes to WAL (append-only).
- Followers replicate record into their own WALs.

- Consumers read directly from WAL using offsets.

---

4. Page Cache vs. Log Segment File

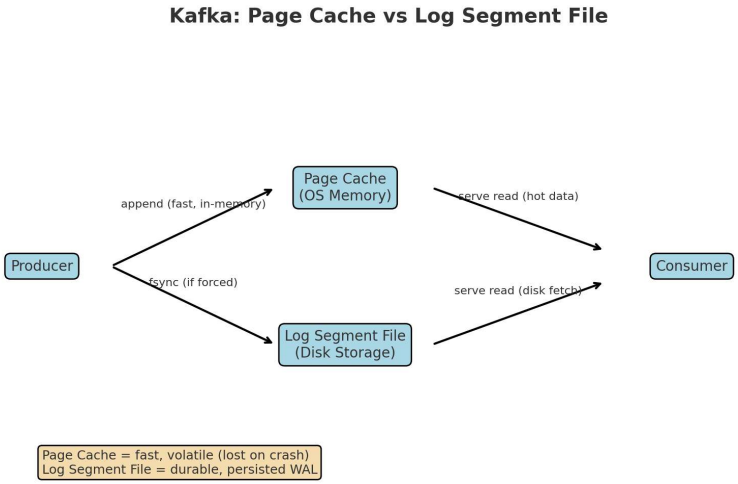
Aspect	Page Cache (OS Memory)	Log Segment File (Disk)
Location	RAM (managed by Linux kernel)	Disk (SSD/HDD)
Durability	Volatile (lost on crash/power failure)	Durable, survives crash
Performance Role	Buffers writes & serves hot reads	System of record
Who manages it	Operating System	Kafka (creates/rolls files)

Aspect	Page Cache (OS Memory)	Log Segment File (Disk)
Write Path	Appends into memory first	Flushed later by OS/fsync
Read Path	Fast if still cached	From disk if cache miss

👉 **Kafka's trick**: Use page cache for speed, replication for durability.

---

5. Page Cache vs Log Segment Flow (Diagram)



Explanation:

- Producers write to page cache first (fast).

- OS flushes to log segment file later (durable).
- Consumers can be served from page cache (hot data) or disk (cold data).

---

6. Durability Controls in Kafka

`acks` (Producer setting)

- `acks=0` → Fire and forget (no WAL guarantee).
- `acks=1` → Leader WAL append only.
- `acks=all` → Leader waits for **all ISR replicas** to append WAL before ack.

`replication.factor`

- Data is safe as long as ≥ 2 replicas have it.
- Typically set to **3** in production.

`min.insync.replicas`

- Minimum replicas that must ack before producer ack.
- e.g. RF=3, min.insync.replicas=2 → tolerate 1 failure safely.

`log.flush.interval.ms / messages`

- Controls explicit fsync frequency.
- Usually left unset → rely on OS flush.

`unclean.leader.election.enable`

- Must be `false` to prevent out-of-sync replicas being elected → avoids data loss.

---

7. Key Questions & Answers

❓ Q1: Does `acks=all` wait for disk flush?

Answer: No. `acks=all` waits for all ISR replicas to **append to their WAL (page cache + log file)**, not for fsync to disk.

❓ Q2: Is page cache replicated?

Answer: No. Page cache is **local OS memory**. What is replicated is the **record itself**, sent to followers who append to their own page cache + log file.

❓ Q3: Why does replication (Step 3) matter if WAL append already happened in Step 2?

Answer: Because with `acks=all`, the ack to the producer happens **only after all ISR replicas have appended**. Replication ensures quorum durability — if the leader crashes, a follower has the record.

❓ Q4: So ack=ALL happens when all replicas have the record in page cache, not after disk fsync?

Answer: Correct ✅. Kafka prioritizes performance. It assumes quorum replication across brokers (each with their own WAL in page cache) is sufficient for durability. Disk flush happens later via OS daemons or segment roll/fsync.

❓ Q5: What happens if an entire cluster loses power before page cache flush?

Answer: That's the edge case where Kafka durability can fail:

- If all replicas lose unflushed page cache at the same time, committed data is lost.
- Mitigation: deploy replicas across racks/AZs to avoid correlated failures.

---

8. Recovery: Checkpointed Offsets

When a broker restarts after crash:

- Kafka uses **checkpoint files** (e.g., `recovery-point-offset-checkpoint`, `leader-epoch-checkpoint`) to find safe log boundaries.
- It truncates unflushed tails, then **re-syncs from the leader**.
- Ensures replicas converge to the same WAL contents.

---

9. Summary

- WAL ensures **sequential, durable logging**.
- In Kafka, the **log itself is the database**.

- Durability = quorum replication + High Watermark, not fsync.
- Page cache = speed, log segment = durability, replication = safety.

👉 For production:

```
replication.factor=3
min.insync.replicas=2
acks=all
unclean.leader.election.enable=false
enable.idempotence=true
```

Kafka Broker Recovery After Crash with Unflushed Page Cache

❓ Question

Once a broker node fails and it has some data in page cache which couldn't be written in log segment, then how does the broker catch up once it comes back?

✅ Answer

When a Kafka broker crashes, any data that existed only in its **page cache** (not yet flushed to disk as a log segment write) is lost on that broker.

However, **Kafka's durability model doesn't rely on a single broker's flush**. Instead, it relies on **quorum replication** and **High Watermark (HW)** to guarantee committed records are safe.
When the broker restarts, it repairs and syncs its log with the current leader.

🔹 Step-by-Step Recovery

1. Boot & Log Scan

- Broker reads **checkpoint files**:
  - `recovery-point-offset-checkpoint` → last known flushed offset.
  - `leader-epoch-checkpoint` → maps leader epochs to start offsets.
- Active segment is opened, and `.index` / `.timeindex` files rebuilt if needed.

2. Tail Validation & Truncation

- Scan from recovery point to Log End Offset (LEO).
- Validate record batch CRCs.
- Truncate any **partial or corrupt tail records** left from crash.

3. Compare with Leader (Find Divergence)

- Current partition leader provides its **leader epoch history**.
- Restarted broker aligns epochs with leader.
- If it has any **uncommitted tail**, it is truncated to the leader's log boundary.

4. Catch-up via Replication

- Broker issues **Fetch requests** to leader from its new LEO.
- Leader streams missing records.
- Broker appends them to its log (through page cache, flushed later).

5. Rejoin ISR

- Once caught up and replica lag ≤ threshold (`replica.lag.time.max.ms`), the broker is marked **in-sync** again.
- From then, it can serve clients and even become leader again.

🔹 ASCII Timeline

- Crash:
  - Broker B loses page cache (data not flushed)
  - Other ISR replicas still have data → partition safe
- Restart:
  - B reads checkpoints
  - B validates tail & truncates invalid batches
  - B aligns epochs with leader → truncates divergence
  - B fetches missing records from leader
  - B catches up → rejoins ISR

Kafka Consumers and Follower Nodes

❓ Question

How does a consumer get to know that there is a record ready for consumption if it is connected to a follower node?

✅ Answer

In Apache Kafka, **consumers never consume directly from follower nodes**.
Consumers always fetch from the **leader replica** of a partition. Followers exist only for **replication** and **fault tolerance**.

If a consumer mistakenly contacts a follower for data:

- The follower responds with a `NotLeaderForPartition` error.
- The consumer client updates its **metadata cache** (partition → leader mapping) by

querying the cluster.

- The consumer retries its fetch from the correct **leader broker**.

🔹 How Consumers Know Records Are Ready

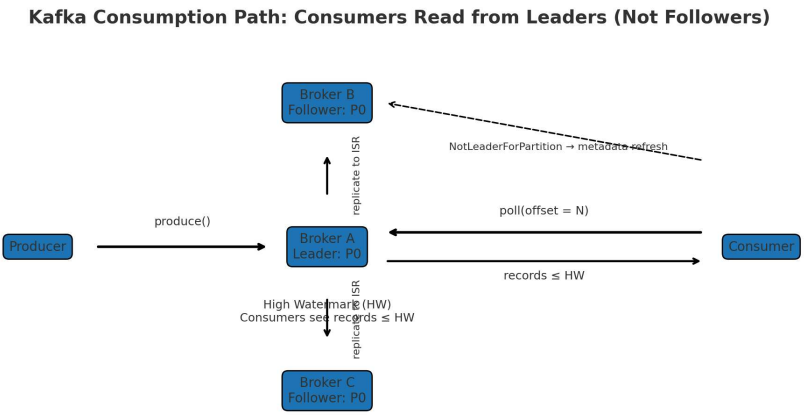
- Consumers use a **poll()** loop to request data from the leader broker.
- The leader tracks the **High Watermark (HW)** = highest offset replicated to all in-sync replicas.
- Only records up to the **HW** are exposed to consumers, ensuring they never read data that could be lost in a leader crash.

🔹 Flow

1. **Producer → Leader**
    - Producer sends records to the partition leader.
    - Leader appends them to its WAL (log segment).

2. **Leader → Followers (Replication)**
    - ISR (in-sync replicas) followers pull records from the leader and append to their own WALs.
3. **Consumer → Leader**
    - Consumer polls the leader for records starting from its last committed offset.
    - Leader returns records ≤ **High Watermark (HW)**.
4. **Consumer accidentally contacting a follower**
    - Follower replies with `NotLeaderForPartition`.
    - Consumer refreshes metadata and connects to the correct leader.

🔹 Diagram



- Producers write to the **Leader**.
- Followers replicate data into their own WALs.
- Consumers **poll only from the Leader**.
- If they mistakenly hit a follower, they are redirected.
- Leader serves records up to the **HW** (safe, replicated data).

🔹 Important Consumer Properties

`max.poll.records`

- **Definition:** Maximum number of records returned by a single `poll()` call.
- **Purpose:** Controls batch size for processing.
- **Tuning:**
  - Higher → fewer network round trips, but more processing latency per batch.
  - Lower → faster responsiveness, but more frequent polls.
- **Example:**

```
max.poll.records=500
```

→ Each poll will return up to 500 records at most.

max.poll.interval.ms Definition: Maximum delay between two consecutive poll() calls before the broker considers the consumer dead and triggers a rebalance.

Default: 300000 ms (5 minutes).

Purpose: Ensures that a consumer that is “stuck” processing records without polling is removed from the group.

Tuning:

Must be set high enough to allow your longest batch processing to finish.

If exceeded, the consumer will lose its partitions and cause group rebalance.

Example:

properties Copy Edit max.poll.interval.ms=600000 → Consumer can take up to 10 minutes to process a batch before re-polling.

Relation Between the Two max.poll.records controls how much work is handed to the consumer at once.

max.poll.interval.ms sets how long the consumer is allowed to process that work before re-polling.

⚖️ Together they balance throughput vs. stability:

Too high records + too low interval → risk of rebalance.

Too low records → under-utilization of consumers.

🔹 Key Points Consumers always fetch from leaders, not followers (except in special cases like KIP-392 Read Replicas).

High Watermark (HW) ensures consumers see only committed data.

Followers exist only for replication and failover.

Redirection via NotLeaderForPartition keeps consumers aligned with leaders.

Consumer tuning requires careful setting of max.poll.records and max.poll.interval.ms to balance throughput and avoid unnecessary rebalances.

📌 Final Takeaway Consumers never read directly from followers. They poll leaders, who expose only replicated records (≤ HW). If a consumer mistakenly connects to a follower, Kafka redirects it to the leader. Consumer configs like max.poll.records and max.poll.interval.ms govern batch size and processing timeout, ensuring efficient and stable consumption.