

what is fencing token and which problem it prevents in distributed systems. Brief answer

A **fencing token** is a **monotonically increasing number** given to a client when it acquires a lock or leadership in a distributed system.

What problem it prevents:

It prevents the **split-brain / stale client problem**, where an old or slow client continues to act as the leader after losing its lock (e.g., due to network delays or timeouts).

How:

Each operation includes the fencing token, and the system only accepts requests with the **highest token**, rejecting outdated ones.



what are the techniques to prevent cache stampede?

Cache stampede (thundering herd) happens when many requests miss the cache at once and hit the backend.

Common techniques to prevent it (brief):

- **Cache locking / mutex**: Only one request rebuilds the cache; others wait.
- **Request coalescing**: Merge identical concurrent requests.
- **Early expiration (soft TTL)**: Serve stale data while one request refreshes.
- **TTL randomization (jitter)**: Avoid many keys expiring at the same time.
- **Write-through / refresh-ahead**: Update cache proactively.
- **Negative caching**: Cache "not found" results.
- **Rate limiting / backpressure**: Protect the backend during spikes.



what is the difference between cache locking and request coalescing.
They seemed alike to me

They're related but **not the same**. The difference is **where the coordination happens** and **who waits**.

Cache Locking (Mutex)

- **Where:** At the cache layer (e.g., Redis lock).
- **How:**
 - First request acquires a lock and rebuilds the cache.
 - Other requests **wait or retry** until the cache is filled.
- **Effect:** Prevents multiple backend recomputations.
- **Trade-off:** Waiting clients add latency.

Example:

Only one worker queries the DB; others block until cache is updated.

Request Coalescing

- **Where:** At the application / request handling layer.
- **How:**
 - Identical concurrent requests are **merged**.
 - One request hits the backend; others **attach to its result**.
- **Effect:** No blocking or retry loops—responses are shared.
- **Trade-off:** More complex app logic.

Example:

100 requests → 1 DB query → 100 responses from the same result.

Key Difference (One line)

- **Cache locking:** "Everyone wait while one rebuilds."
- **Request coalescing:** "Everyone share the same in-flight request."

They solve the same problem but at **different layers** and with **different latency behavior**.



Give me code example for both of them using java 25 spring boot

Below are **simple, practical examples** showing the **difference in behavior**, not production-ready code.

1 Cache Locking (Mutex using Redis)

Idea:

Only one request rebuilds the cache. Others wait or retry.

Service using Redis lock

java Copy code

```
@Service
public class ProductService {

    private final StringRedisTemplate redisTemplate;
    private final ProductRepository repository;

    public ProductService(StringRedisTemplate redisTemplate,
                          ProductRepository repository) {
        this.redisTemplate = redisTemplate;
        this.repository = repository;
    }

    public Product getProduct(String id) throws InterruptedException {
        String cacheKey = "product:" + id;
        String lockKey = cacheKey + ":lock";

        // 1. Try cache
        String cached = redisTemplate.opsForValue().get(cacheKey);
        if (cached != null) {
            return deserialize(cached);
        }
    }
}
```

```

// 2. Try acquire lock
Boolean lockAcquired = redisTemplate.opsForValue()
    .setIfAbsent(lockKey, "1", Duration.ofSeconds(5));

if (Boolean.TRUE.equals(lockAcquired)) {
    try {
        // 3. Rebuild cache
        Product product = repository.findById(id);
        redisTemplate.opsForValue()
            .set(cacheKey, serialize(product), Duration.ofMinutes(10));
        return product;
    } finally {
        redisTemplate.delete(lockKey);
    }
}

// 4. Lock not acquired → wait & retry
Thread.sleep(50);
return deserialize(redisTemplate.opsForValue().get(cacheKey));
}
}

```

Behavior:

- One thread hits DB
- Others wait and retry

2 Request Coalescing (In-Flight Request Sharing)

Idea:

All concurrent requests share the same future.

Service using `CompletableFuture`

```

java                                                                    Copy code

@Service
public class ProductService {

    private final ProductRepository repository;

```

```

private final ConcurrentHashMap<String, CompletableFuture<Product>> inFlight =
    new ConcurrentHashMap<>();

public ProductService(ProductRepository repository) {
    this.repository = repository;
}

public Product getProduct(String id) {
    CompletableFuture<Product> future = inFlight.computeIfAbsent(id, key ->
        CompletableFuture.supplyAsync(() -> repository.findById(key))
            .whenComplete((r, e) -> inFlight.remove(key))
    );

    return future.join();
}
}

```

Behavior:

- First request triggers DB call
- Others **attach to same future**
- **No waiting loops, no retries**

Key Difference in Code

Aspect	Cache Locking	Request Coalescing
Coordination	External (Redis)	In-memory
Waiting	Sleep / retry	Shared future
Latency	Higher	Lower
Complexity	Lower	Higher

One-line summary

- **Cache locking:** "Wait until cache is rebuilt"
- **Request coalescing:** "Reuse the same in-flight request"

If you want, I can also show **distributed request coalescing** or **hybrid (soft TTL + coalescing)** patterns.



Negative caching: Cache “not found” results

Negative caching means **caching the “not found” result** (e.g., `null`, 404) so repeated requests don’t keep hitting the backend.

Why it’s needed

Without it:

- Missing keys cause **repeated DB calls**
- Leads to **hot misses** and cache stampede

Example (Java 25, Spring Boot)

Cache “not found” with a short TTL

```
java                                                                    Copy code

@Service
public class UserService {

    private final StringRedisTemplate redisTemplate;
    private final UserRepository repository;

    private static final String NOT_FOUND = "NF";

    public UserService(StringRedisTemplate redisTemplate,
                       UserRepository repository) {
        this.redisTemplate = redisTemplate;
        this.repository = repository;
    }

    public Optional<User> getUser(String id) {
        String key = "user:" + id;
```

```

String cached = redisTemplate.opsForValue().get(key);
if (cached != null) {
    if (NOT_FOUND.equals(cached)) {
        return Optional.empty(); // cached miss
    }
    return Optional.of(deserialize(cached));
}

// DB Lookup
Optional<User> user = repository.findById(id);

if (user.isEmpty()) {
    // Cache negative result with SHORT TTL
    redisTemplate.opsForValue()
        .set(key, NOT_FOUND, Duration.ofSeconds(30));
    return Optional.empty();
}

redisTemplate.opsForValue()
    .set(key, serialize(user.get()), Duration.ofMinutes(10));

return user;
}
}

```

Key Points

- Use short TTL for negative entries (seconds, not minutes)
- Prevents repeated DB hits for missing data
- Common for:
 - User IDs
 - Product IDs
 - API lookups

Risks & Mitigation

Risk	Mitigation
Data appears later	Short TTL
Cache pollution	Separate marker value
Legitimate null vs missing	Explicit sentinel (<code>NF</code>)

One-liner

Negative caching trades short staleness for massive backend protection.



In distributed systems can message queues (rabbitmq or kafka) be used instead of distributed lock for competing processes

Short answer:

Sometimes yes, often no — they solve *different coordination problems*.

Can MQ replace a distributed lock?

✅ Yes, if the problem is *work ownership / serialization*

Message queues naturally ensure **only one consumer processes a message**.

Examples where MQ works instead of a lock:

- Order processing
- Email sending
- Payment events
- Background jobs
- Task scheduling

Why it works:

- Each message is delivered to **one consumer**
- Ordering (Kafka partition / RabbitMQ queue) gives **implicit mutual exclusion**

👉 You don't need a lock if the queue already guarantees exclusivity.

❌ No, if the problem is *shared resource protection***

Distributed locks protect **critical sections over shared state**.

Examples where MQ cannot replace a lock:

- Leader election
- Preventing concurrent DB writes to same row
- Coordinating access to external systems
- Exactly-once state transitions across services

Why it fails:

- MQ does **not stop two services** from acting on the same resource outside the queue
 - Consumers can crash after side effects
 - Re-delivery ≠ mutual exclusion
-

Kafka vs RabbitMQ nuance

System	What it gives	Limitation
Kafka	Partition-level ordering	No exclusive consumer across groups
RabbitMQ	Exclusive queue consumers	Still not a lock

Rule of Thumb

- Use MQ → *"Who should do this work?"*
 - Use distributed lock → *"Who may touch this shared resource?"*
-

Practical takeaway

If you can **model the problem as messages**, prefer MQ.
If you must **guard shared mutable state**, you still need a lock.

One-liner

Queues serialize *work*; locks serialize *state*.

If you want, I can give **real-world redesign examples** where locks were removed using Kafka/RabbitMQ.



What can be used instead of distributed lock then?

There is **no single replacement** for a distributed lock.

Instead, you **change the coordination model** so locks aren't needed.

Below are the **main alternatives**, when to use them, and **why they work**.

1 Database constraints & transactions (Most common)

Use when: Shared data lives in a DB.

Techniques

- Unique constraints
- Conditional updates (`WHERE version = ?`)
- Serializable / repeatable-read transactions

Example

sql

 Copy code

```
UPDATE account
SET balance = balance - 100
WHERE id = 1 AND balance >= 100;
```

- ✓ Atomic
- ✓ Crash-safe
- ✗ DB contention at scale

Replaces: Locks for data consistency

2 Optimistic Concurrency Control (OCC)

Use when: Conflicts are rare.

How

- Version numbers / timestamps
- Retry on conflict

Example

sql

 Copy code

```
UPDATE document
SET content = ?, version = version + 1
WHERE id = ? AND version = ?;
```

- ✓ No blocking
- ✗ Retries under contention

Replaces: Fine-grained locks

3 Idempotency + Deduplication


Use when: Exactly-once behavior is needed.

How

- Idempotency keys
- Dedup tables
- Natural keys

Example

text

 Copy code

```
order_id UNIQUE
```

✓ Works with retries & MQs

✗ Needs careful design

Replaces: Locks preventing double execution

4 Message Queues (Work serialization)

Use when: Problem can be modeled as tasks/events.

How

- One consumer per key/partition
- Sequential processing

Example

- Kafka key = `userId`
- RabbitMQ single consumer

✓ High scalability

✗ Not for shared state protection

Replaces: Locks for job ownership

5 Leader Election (Scoped locking)

Use when: Only one instance should act.

Tools

- ZooKeeper
- etcd
- Kubernetes leader election

✓ Clear ownership

✗ Coarse-grained

Replaces: Global locks

6 Fencing Tokens (Lockless safety)

Use when: You still need coordination but want safety.

How

- Monotonic token
- Reject stale writers

Example

```
text Copy code
```

```
token > last_seen_token
```

- ✓ Prevents split-brain
- ✗ Needs enforcement everywhere

Complements: Locks, or replaces unsafe ones

7 CRDTs / Conflict-free data structures

Use when: Eventual consistency is acceptable.

Examples

- Counters
 - Sets
 - Maps
- ✓ No locks
 - ✗ Complex mental model

Replaces: Locks in collaborative systems

Decision Cheat Sheet

Problem	Use
Protect DB row	DB transaction / OCC
Prevent duplicates	Idempotency
Serialize work	MQ
Single active instance	Leader election
Avoid stale writers	Fencing tokens
High scale, eventual consistency	CRDTs

Key Insight

Distributed locks are a last resort.

Most systems eliminate them by **changing ownership, data modeling, or consistency guarantees.**

One-liner

Instead of asking “How do I lock?”, ask “Who owns this state, and where can **atomicity** be enforced?”

If you want, tell me **your concrete use case** and I’ll map it to the best pattern.



Can Optimistic Concurrency Control (OCC) prevents db problems like read/write skew or any other under heavy thread?

Short answer:

OCC helps, but it does *not* prevent all anomalies.

What it prevents depends on **how it’s implemented** and the **DB isolation level**.

What OCC *does* prevent

OCC detects **lost updates**.

Example (lost update)

text Copy code

```
T1 reads version = 5
T2 reads version = 5
T1 updates WHERE version = 5 → success (version = 6)
T2 updates WHERE version = 5 → 0 rows → conflict detected
```

- ✓ Lost updates prevented
- ✓ Safe under heavy concurrency
- ✓ No blocking

What OCC does *NOT* prevent

✗ Read Skew

Problem: Transaction reads multiple rows that change independently.

Example

text Copy code

```
T1 reads A = 10
T2 updates A = 0, B = 20
T1 reads B = 0
```

OCC only checks **writes**, not **read consistency**.


✗ Read skew still possible

✗ Write Skew (classic anomaly)

Problem: Two transactions make valid independent updates that violate an invariant.

Example

text

 Copy code

Invariant: At least one doctor must be on call

T1 sees Dr A on call → removes self

T2 sees Dr B on call → removes self

Both commit successfully → invariant broken.

✗ OCC does NOT prevent write skew

✗ Phantom Reads

Problem: New rows appear between reads.

OCC doesn't detect inserts by others.

✗ Still possible

What *does* prevent these anomalies

Anomaly	Solution
Lost update	OCC
Read skew	Repeatable Read / Serializable
Write skew	Serializable isolation
Phantoms	Serializable or predicate locking

Heavy Thread / High Concurrency Reality

OCC under heavy load:

- High conflict → many retries
- Throughput may **collapse**
- Still correct for *single-row invariants only*

Correct mental model

OCC = "Detect conflicting writes"

Serializable isolation = "Prevent conflicting reads & writes"

OCC is **not** a replacement for serializable transactions.

Best Practice Pattern

- Use **OCC** for single-entity updates
 - Use **DB constraints** for invariants
 - Use **Serializable isolation** for cross-row invariants
 - Avoid locks unless unavoidable
-

One-liner

OCC prevents lost updates, not logical anomalies like read/write skew — isolation levels do.

If you want, I can show **code examples** comparing OCC vs SERIALIZABLE in PostgreSQL or MySQL.



What can be used then in order to prevent all "anomalies" known?

Short, precise answer:

👉 Only ***Serializable isolation*** (true serializability) prevents ***all*** known transactional anomalies.

There is **no lighter-weight mechanism** that gives the same guarantee.

What "all anomalies" means

Serializable isolation prevents:

- Lost updates
- Dirty reads
- Non-repeatable reads
- Phantom reads
- Read skew
- Write skew
- Predicate anomalies
- Invariant violations under concurrency

The execution is equivalent to *some serial order* of transactions.

What actually provides this in practice

1 Serializable isolation in the database (Gold standard)

Examples

- PostgreSQL: `SERIALIZABLE` (SSI)
- SQL Server: `SERIALIZABLE`
- Oracle: `SERIALIZABLE`
- CockroachDB / YugabyteDB: Serializable by default

How

- Predicate locking or
- Serializable Snapshot Isolation (SSI)

Cost

- Lower throughput
- Possible aborts / retries

✓ Correctness guaranteed

✗ Performance tradeoff

2 Single-threaded ownership (outside DB)

How

- One writer per shard / key

- Actor model
- Kafka partition ownership

Guarantee

- No concurrency \Rightarrow no anomalies

Tradeoff

- Reduced parallelism
- Complex routing

3 Strong DB constraints (partial substitute)

Examples

- UNIQUE
- CHECK
- FOREIGN KEY
- EXCLUDE (Postgres)

Important

- Prevents *specific* anomalies
- Not general serializability

What does *not* fully work

Technique	Why insufficient
OCC	Only detects write conflicts
Distributed locks	Fail under partitions / crashes
MQ serialization	Doesn't protect shared state
Fencing tokens	Safety, not isolation
Retry logic	Doesn't fix skew

Industry reality

Most systems **do not use full serializable isolation everywhere** because:

- It's expensive
- High abort rates at scale

Instead they:

1. Use **Serializable** only for critical invariants
2. Use **OCC + constraints** elsewhere
3. Accept weaker consistency where safe

Decision Rule (Very important)

If you **cannot prove** your invariant holds under weaker isolation,
you must use **Serializable**.

One-liner takeaway

Serializable isolation is the only general solution to all concurrency anomalies; everything else is a performance optimization with caveats.

If you want, I can:

- Show **real failure examples** under READ COMMITTED / OCC
- Help **decide where Serializable is truly needed**
- Compare **Postgres SERIALIZABLE vs MySQL behavior**



does Making an update atomic prevents "anomalies"?
UPDATE account
SET balance = balance - 100
WHERE id = 1 AND balance >= 100;


Short answer:

✗ No. Making a single update atomic does *not* prevent all anomalies.

It prevents **one specific class** of problems only.

What this statement *does* prevent

sql

 Copy code

```
UPDATE account
SET balance = balance - 100
WHERE id = 1 AND balance >= 100;
```

Prevents:

- Lost updates
- Over-withdrawal on a single row

Why:

- The update is **atomic**
- Condition is checked **at write time**
- DB guarantees row-level atomicity

Even under heavy concurrency, balance never goes below 0.

What it does *NOT* prevent

1 Read skew

text

 Copy code

```
T1 reads account A
T2 updates account A and B
T1 reads account B → inconsistent view
```

Atomic update doesn't help—this is a **read anomaly**.

2 Write skew (cross-row invariant)

Invariant: total balance across two accounts ≥ 100

text

 Copy code

```
T1 withdraws from A
T2 withdraws from B
Both succeed → invariant broken
```

Each update is atomic, but **together they violate the rule**.

3 Multi-step business logic

If logic is:

text

 Copy code

```
SELECT balance
IF ok → UPDATE
```

Atomic UPDATE helps *only if all logic is inside the statement*.

If not → anomalies return.

4 Phantom / predicate anomalies

Atomic updates don't protect:

- Range queries
 - Aggregates
 - `COUNT`, `SUM`, existence checks
-

When atomic updates are *enough*

Use them when:

- The invariant is **single-row**
- All logic fits in **one SQL statement**
- No cross-row or cross-table dependencies

Examples

- Inventory decrement
- Rate limiting counters

- Quota usage
 - Idempotent state transitions
-

When they are *not enough*

Use **SERIALIZABLE** isolation or constraints when:

- Invariants span **multiple rows**
 - You rely on **reads before writes**
 - Business rules involve **aggregates**
-

Key mental model

| Atomic statements protect data, not logic.

They guarantee correctness **only for what's inside the statement**.

One-liner

Atomic updates prevent lost updates, not higher-level anomalies like read/write skew.

If you want, I can show **real SQL examples** where atomic updates still break invariants.



what can be used for preventing all "anomalies" except for
SERIALIZABLE isolation level?

If you **cannot or do not want to use SERIALIZABLE isolation**, there is **no single catch-all mechanism** that prevents *all anomalies*. But you can **combine techniques** to achieve similar safety for critical invariants.

Here's a structured breakdown:

1 Atomic statements (single-statement invariants)

- Use **single SQL statements** with conditions.
- Guarantees correctness **for single-row or single-resource invariants**.

Example

sql

 Copy code

```
UPDATE account
SET balance = balance - 100
WHERE id = 1 AND balance >= 100;
```

✅ Prevents:

- Lost updates
- Over-withdrawal on a single row

❌ Does not prevent:

- Cross-row invariants
 - Read/write skew
 - Phantoms
-

2 Database constraints

- UNIQUE, CHECK, FOREIGN KEY, EXCLUDE (Postgres)
- Enforce **structural invariants at the DB layer**

Example

sql

 Copy code

```
ALTER TABLE accounts
ADD CONSTRAINT positive_balance CHECK (balance >= 0);
```

✅ Prevents:

- Certain cross-row anomalies when combined with careful transactions

❌ Cannot prevent:

- Write skew across multiple rows in a single transaction unless each row is constrained
-

3 Optimistic Concurrency Control (OCC)

- Use version numbers / timestamps for updates
- Detect conflicting writes

Example

sql

 Copy code

```
UPDATE document
SET content = ?, version = version + 1
WHERE id = ? AND version = ?;
```

✅ Prevents:

- Lost updates

❌ Does not prevent:

- Read skew
- Write skew
- Phantom reads

4 Application-level locking / coordination

- Distributed locks, leader election, or single-writer queues
- Ensure only **one actor updates a shared resource**

✅ Can prevent:

- Write skew
- Invariant violations if designed carefully

❌ Risks:

- Deadlocks, availability issues, complexity
- Not transparent like SERIALIZABLE isolation

5 Idempotency + deduplication

- Use **idempotency keys** for operations
- Ensures repeated or concurrent requests do not violate invariants

✅ Prevents:

- Duplicate execution
- Some lost updates

✗ Does not prevent:

- Cross-row anomalies unless combined with other techniques
-

6 CRDTs / Conflict-free replicated data structures

- For **distributed, eventually consistent systems**
- Merges concurrent updates in a safe way

✓ Prevents:

- Lost updates
- Some write conflicts

✗ Does not prevent:

- Arbitrary business invariants
 - Complex multi-row dependencies
-

💡 Practical takeaway

To mimic **SERIALIZABLE** without using it:

1. Use **atomic SQL statements** wherever possible.
2. Enforce **DB constraints** for simple invariants.
3. Apply **OCC or version checks** for concurrent writes.
4. Use **distributed locks / single-writer pattern** for multi-row invariants.
5. Make operations **idempotent** to prevent duplicate execution.

This is a **hybrid approach**: no single technique guarantees all anomalies, but together you can cover most practical cases safely.

One-liner

Without **SERIALIZABLE**, you must combine atomic updates, constraints, OCC, and careful coordination to prevent most—but not all—concurrency anomalies.

If you want, I can make a **table mapping every anomaly** → **which technique prevents it without SERIALIZABLE**, which is super practical for system design. Do you want me to do that?

