What happens when cache evict occurs in the middle of a query taking long time. Transaction isolation level is repeatable read. Does such inconsistency occur: Some of the entities has stale cache data and some entities has fresh

Short answer: **No, a cache eviction in the middle of a long-running query does *not* cause an inconsistency *within that query* under `REPEATABLE READ`.** You won't end up with "half stale, half fresh" *for the same query result set*. But there *are* nuances depending on **what cache** you're talking about and **where it sits**.

Let's break it down cleanly.

---

1. What `REPEATABLE READ` actually guarantees (database side)

Under **REPEATABLE READ**:

- The transaction sees a **consistent snapshot** of the database as of the time the transaction (or first statement, depending on DB) started.
- All reads within that transaction see the *same committed data*, even if other transactions commit changes meanwhile.
- This guarantee is enforced **entirely by the database**, independent of any application-level cache.

So from the DB's perspective:

> A long-running query will *never* see mixed old/new rows caused by concurrent commits.

---

2. What happens if cache eviction occurs mid-query?

Case A: Database internal cache (buffer pool / page cache)

Examples:

- InnoDB buffer pool
- PostgreSQL shared buffers

Eviction here is purely a performance concern, not a correctness one.

- Evicted pages are reloaded from disk if needed
- MVCC ensures the correct snapshot version is read
- You still get a consistent view

✅ **No inconsistency possible**

Case B: Second-level / ORM cache (e.g., Hibernate L2 cache)

This is where confusion often comes from.

Key points:

- ORM caches are **not transactional snapshots**
- They usually store **entity state at a point in time**
- Eviction may happen:
  - manually
  - due to TTL
  - due to memory pressure

Now, during a **single transaction**:

- If an entity is already loaded → it comes from the **first-level (session) cache**
- If not loaded yet:
  - ORM may check L2 cache
  - Or hit the database

If L2 cache entry is evicted mid-transaction:

- The ORM simply falls back to the DB
- The DB returns data consistent with the transaction's snapshot

So within one transaction:

> **You still won't get mixed snapshots**
> You may get *mixed cache sources*, but **not mixed data versions**

✅ Still consistent

Case C: Application-level / distributed cache (Redis, Memcached)

This is the risky one.

Important distinction:

- These caches are **outside** DB transactional guarantees
- They do **not** respect isolation levels

If your application:

- Reads *some* entities from cache
- Reads *others* from DB
- And cache eviction happens mid-query

Then yes:

⚠️ **You can observe logical inconsistency**, e.g.:

- Some entities reflect newer committed data
- Others reflect older cached values

But this inconsistency is **not caused by** `REPEATABLE READ` **failing** — it's caused by:

> Mixing transactional DB reads with non-transactional cache reads

This happens when:

- Cache is not transaction-aware
- Cache invalidation is async or eventual
- Reads don't enforce "DB-only within transaction"

---

3. So does "some stale, some fresh" ever happen?

Scenario	Can it happen?
DB buffer cache eviction	❌ Never
ORM L2 cache eviction	❌ Not within a transaction
App-level / distributed cache	✅ Yes

---

4. Best practices to avoid inconsistency

If consistency matters:

- **Within a transaction**, always:
  - Read from DB (or ORM session cache)
  - Avoid Redis/Memcached reads
- Or:
  - Make cache **read-through + write-through**
  - Or use cache only *outside* transactional logic
- If using Hibernate:
  - Rely on first-level cache for transactional consistency
  - Be cautious with L2 cache for mutable entities

TL;DR

- `REPEATABLE READ` guarantees a consistent snapshot **regardless of cache eviction**
- Cache eviction does **not** cause mixed snapshots *by itself*
- Mixed stale/fresh data only happens if you:
  - Mix DB reads with **non-transactional external cache reads**

If you want, tell me:

- which DB (MySQL/Postgres/Oracle),
- which ORM (Hibernate/JPA),
- and what cache (Redis, Ehcache, etc.)

I can get very concrete about your exact setup 👀

I did not get this: You may get mixed cache sources, but not mixed data versions. Consider this. I have a list of Country entity which is cached in L2. In the middle of query 2 countries (Country1 and Country 2 named)in this list are updated and cache ivalidated. My concern is this: After query finished. The result set contains C1 from old

cache and C2 from new cache because this is kind of inconsistent

Great question — this is *exactly* the right edge case to poke at. Let's walk it **step by step** and be very precise, because the intuition here is sneaky.

I'll assume:

- Hibernate / JPA
- L2 cache is **non-transactional** (the common case)
- DB isolation = **REPEATABLE READ**
- One long-running query inside **one transaction**

Key rule first (this is the crux)

> **Hibernate will never assemble a single result set from multiple versions of the same DB snapshot within one transaction.**

Even if L2 cache entries are invalidated *while the query is running*.

Now let's prove it with your exact scenario.

Your scenario, step by step

Initial state

- Transaction **T1** starts
- Query: `SELECT * FROM Country`
- L2 cache contains Country1, Country2, Country3…

What Hibernate actually does

Hibernate does **not**:

- Read Country1
- Pause
- Read Country2 later
- And interleave cache hits dynamically

Instead, for an entity query:

1. Hibernate sends **one SQL query** to the database
2. The database returns **rows from a single MVCC snapshot**
3. Hibernate materializes entities **from that result set**
4. L2 cache is only used to:
    - avoid hitting DB *if the entity is already cached*
    - or store the entity *after* materialization

Important:

> Hibernate **does not fetch entities one-by-one from L2 cache during a SQL result set iteration**

Now the “mid-query update” happens

While the DB is executing the query:

- Another transaction **T2** updates Country1 and Country2
- T2 commits
- L2 cache entries for Country1 and Country2 are invalidated

What effect does this have?

On the database query

None.

- DB snapshot was fixed when T1 started (or at first statement)
- The SQL query still returns **old versions** of Country1 and Country2

On Hibernate materialization

Hibernate already has the row data:

- It uses **row values**, not L2 cache, to build entities
- L2 cache eviction does *not* affect rows already returned

So the entities created in T1 are:

- Country1 → old version
- Country2 → old version

✅ **Consistent snapshot**

But what about L2 cache lookup during query execution?

This is the subtle misunderstanding.

Hibernate **does not do this**:

```
Row for Country1 → check L2 cache → use cache
Row for Country2 → check L2 cache → use cache
```

Instead:

- Rows are authoritative
- Cache is secondary

L2 cache **never overrides row data from the DB**.

What if the entity was already in the persistence context?

Even then:

- First-level cache (Session) wins
- It already reflects the transaction's snapshot
- L2 cache is ignored

When *would* your feared inconsistency happen?

Only if **you explicitly do something unsafe**, like:

- Manually loading entities from L2 cache
- Mixing:
  - `session.find(Country.class, id)` (DB-backed)
  - with direct cache access (Redis / Ehcache API)
- Or running **multiple separate queries** *outside* one transaction

Example that *can* break consistency:

```
Country c1 = countryCache.get(1L); // external cache Country c2 =
entityManager.find(Country.class, 2L); // DB snapshot
```

Now you can absolutely get:

- c1 = new
- c2 = old

But that's not Hibernate's L2 cache behavior.

Final answer to your concern (very directly)

> **No, you cannot end up with Country1 from old cache and Country2 from new cache in the same query result.**

Because:

- SQL result set comes from one DB snapshot
- Hibernate materializes entities from that result
- L2 cache eviction does not inject newer data mid-query

Your result will be either:

- All old (most common)
- Or all from the persistence context

But **never a mix of versions**.

One-liner intuition

> L2 cache helps Hibernate avoid DB hits — it does *not* define what version of data a transaction sees.

If you want, I can also:

- diagram the lifecycle visually
- show the exact Hibernate code path
- or explain how this differs for query cache vs entity cache (that's another subtle trap 😄)

Just say the word.

I am still confused. As you said L2 cache helps Hibernate avoid DB hits but you also said
DB snapshot was fixed when T1 started (or at first statement)
The SQL query still returns old versions of Country1 and Country2
If Country is a cached entity (it is) hibernate should not include country related sql in select statement and after result set is return hibernate should materialize country ids from getting them from cache (by iterating) and during this should not the scenario i described occur?

Ah — **this is the exact mental model that's tripping you up**, and you're not alone. Let's fix it cleanly by correcting **one wrong assumption**:

> ❌ *"If an entity is cached in L2, Hibernate does not include it in the SQL and later hydrates it from cache."*

That assumption is **false**.

Once that clicks, everything else snaps into place.

The single most important correction

Hibernate NEVER executes an entity query that returns only IDs and then hydrates entities from L2 cache

For this query:

```
SELECT c FROM Country c
```

Hibernate **always** generates SQL like:

```
SELECT c.id, c.name, c.code, c.population, ... FROM country c
```

Even if:

- Country is fully cached in L2

- Query cache is enabled
- Cache hit ratio is 100%

➡️ **Entity state always comes from the SQL result set**, not from L2 cache.

So what is L2 cache actually used for?

Only in these cases:

1️⃣ `find()` / `get()` by primary key

```
Country c = entityManager.find(Country.class, 1L);
```

Here Hibernate:

- Checks first-level cache
- Checks L2 cache
- Hits DB *only if cache miss*

⚠️ But even here:

- The L2 cache entry was created **from a DB row**
- It represents *some committed version*
- Hibernate never merges multiple versions in one transaction

---

2️⃣ Storing entities *after* materialization

During your query:

1. SQL returns rows (old snapshot)
2. Hibernate materializes Country objects
3. Hibernate may:
    - put them into L1 cache
    - put them into L2 cache **if allowed**

But L2 cache **never feeds data into the query execution path**.

3️⃣ Query cache (different thing!)

Even with **query cache** enabled:

- The cache stores:
  - **IDs**
  -
    - a timestamp / invalidation key
- Hibernate still:
  - checks DB-level invalidation
  - fetches entities by ID
  - those fetches are still subject to L1 cache rules

Even here:

> DB snapshot still governs correctness.

Why Hibernate *cannot* do what you're imagining

If Hibernate tried to:

- run `SELECT id FROM country`
- then hydrate entities from L2 cache

It would:

- violate transaction isolation
- bypass locking / MVCC
- break repeatable read
- break dirty checking

So it **never does this**.

Now replay your scenario with the correct model

Timeline

1. T1 starts (`REPEATABLE READ`)
2. Hibernate executes:

```
SELECT id, name, ... FROM country
```

3. DB returns **old snapshot rows**
4. T2 updates Country1 and Country2, commits
5. L2 cache entries are invalidated
6. Hibernate materializes entities **from row data already returned**

Result:

- Country1 → old
- Country2 → old

No cache lookup happens during step 6.

When CAN "mixed versions" actually happen?

Only if **you explicitly mix access paths**, for example:

- Direct Redis/Ehcache access
- Native SQL + cache reads
- Multiple transactions
- StatelessSession
- Custom loaders
- Read-only caches used outside Hibernate

But **not** from a normal Hibernate entity query.

Mental model to keep forever 🧠

> **SQL defines the truth. Cache only avoids future SQL.**

Or even shorter:

> **L2 cache is write-behind memory, not a read-ahead data source for queries.**

If you want, next we can:

- contrast entity cache vs query cache with diagrams
- show Hibernate's actual `Loader` flow
- or walk through a real log (`hibernate.show_sql + cache logs`)

You're asking the *right* questions — this is deep Hibernate internals territory.

But what is the point of marking a @OneToOne or @OneToMany relation as @Cachable then?

YES — this is *the* right follow-up. And the answer is much simpler (and less magical) than it sounds.

Short version first, then the concrete mechanics.

Short answer

> Marking a relationship `@Cacheable` **does not mean Hibernate skips SQL for that relationship in a query**.

It means:

> **Once the association has been loaded at least once, Hibernate can reuse it later without re-querying the DB.**

Caching is about **subsequent access**, not about how the *original query* is executed.

What `@Cacheable` on an association actually does

Let's say:

```
@Entity @Cacheable public class Country { @OneToMany(mappedBy = "country")
@Cacheable private Set<City> cities; }
```

There are **two different caches involved**:

1 Entity cache (`Country` , `City`)

Stores:

- Entity state (columns)
- Indexed by entity ID

2️⃣ Collection cache (`Country.cities`)

Stores:

- **Only the IDs of associated entities**
- Example:

```
Country(1).cities → [10, 11, 12]
```

⚠️ Important:

- **No SQL rows are cached**
- **No joins are cached**
- **No snapshots are shared across transactions**

What happens on first access

```
Country c = em.find(Country.class, 1L); c.getCities().size();
```

First time:

1. Hibernate loads `Country` (DB or L2 cache)
2. Sees `cities` is lazy
3. Executes:

```
SELECT * FROM city WHERE country_id = 1;
```

4. Materializes City entities
5. Stores:
   - City entities in entity cache
   - City IDs in collection cache

What happens on second access (cache hit)

Later (same or different transaction):

```
Country c = em.find(Country.class, 1L); c.getCities().size();
```

If caches are valid:

1. Country loaded from entity cache
2. Hibernate checks **collection cache**
3. Finds `[10,11,12]`
4. Loads City entities:
    - from L1 cache if present
    - otherwise from L2 entity cache
    - DB hit only if cache miss

🚀 **No SQL for the collection**

What `@Cacheable` does NOT do

It does **not**:

❌ Eliminate SQL joins in JPQL queries
❌ Allow partial hydration from cache
❌ Override transaction isolation
❌ Provide snapshot consistency
❌ Prevent SQL execution on first load

Why this design exists (and why it must)

Hibernate has to:

- Honor DB isolation levels
- Respect MVCC
- Support locking & versioning
- Guarantee dirty checking correctness

If Hibernate tried to “assemble object graphs from cache” during queries:

- Repeatable Read would be broken

- Phantom reads would appear
- Locking semantics would be impossible

So cache is **post-query optimization**, not **query replacement**.

Concrete example where association caching shines

Classic N+1 problem:

```
List<Country> countries = em.createQuery("from Country",
Country.class).getResultList(); for (Country c : countries) {
c.getCities().size(); // N+1 }
```

With collection cache enabled:

- First execution → N+1 (fills cache)
- Second execution → **0 SQL for cities**

That's the win.

Mental model (this usually clicks)

> **Queries load data. Caches remember it for later.**

Or even simpler:

> **Hibernate cache is a memory of the past, not a shortcut for the present query.**

Final takeaway

You were imagining:

> "Hibernate uses cache *instead of* SQL"

Reality is:

> "Hibernate uses cache *after* SQL to avoid future SQL"

Once that distinction is clear, all the isolation-level confusion disappears.

If you want, next we can:

- contrast `JOIN FETCH` vs collection cache
- talk about why query cache is dangerous
- or map which cache types are actually worth using in 2026

Just tell me where you want to go next 👌

OK. But please summarize all up for me and please let me know if i must know anything else about hibernate L1 and L2 cache mechanism

You got it. Here's a **clean, end-to-end summary** that ties everything together and highlights the few *extra things worth knowing* so there are no lingering surprises.

Hibernate L1 & L2 Cache — Clear Mental Model

1️⃣ The single most important rule

> **SQL results define what a transaction sees.**
> **Caches only avoid future SQL — they never replace SQL for entity queries.**

Everything else flows from this.

2️⃣ Transaction isolation vs cache

- `REPEATABLE READ` is enforced **only by the database**
- Hibernate **never bypasses DB snapshots**
- Cache eviction during a query **cannot cause mixed versions** in one result set

Why?

- Hibernate executes **one SQL statement**
- DB returns rows from **one snapshot**
- Hibernate materializes entities **from those rows**

- L2 cache is *not consulted* while building query results

---

3 First-level cache (L1 / Persistence Context)

What it is

- One per `Session` / `EntityManager`
- Always enabled
- Transaction-scoped

What it does

- Guarantees **repeatable reads inside the transaction**
- Ensures **identity** (`same ID → same Java object`)
- Prevents duplicate SQL for the same entity

Key rule

> **L1 cache always wins.**
> If an entity is in L1, Hibernate never goes to L2 or DB.

---

4 Second-level cache (L2)

What it is

- Optional
- Shared across sessions
- Lives outside transactions

What it stores

- **Entity cache** → entity state by ID
- **Collection cache** → *IDs only* of related entities
- **Query cache** → IDs + invalidation timestamps (not rows)

What it is used for

- `find()` / `get()` by ID
- Lazy association initialization

- Avoiding repeated DB hits **across transactions**

What it is NOT used for

❌ Building JPQL/HQL query result sets
❌ Replacing SQL
❌ Providing snapshot isolation
❌ Preventing SQL on first load

5️⃣ Why your "mixed old/new entity" fear cannot happen

You worried about this scenario:

> "Country1 comes from old cache, Country2 from new cache in the same query"

That **cannot happen** because:

- Hibernate **never hydrates query results from L2 cache**
- Entities come from SQL rows, not cache
- Cache invalidation mid-query does not affect already returned rows

Result set = **all old or all new**, never mixed.

6️⃣ What `@Cacheable` on associations actually means

Entity cache

Stores entity state (columns).

Collection cache

Stores **only IDs**, e.g.:

```
Country(1).cities → [10, 11, 12]
```

Benefit

- Avoids **extra SQL later**
- Helps with **N+1 problems across transactions**

Non-benefit

- Does **not** eliminate joins
- Does **not** change query execution
- Does **not** affect isolation

7️⃣ Correct lifecycle of a typical query

```
Transaction starts ↓ Hibernate executes SQL ↓ DB returns snapshot-consistent rows ↓ Hibernate builds entities from rows ↓ Entities go into L1 cache ↓ Optionally stored in L2 cache ↓ Transaction ends
```

Cache is **downstream**, not upstream.

8️⃣ When inconsistency CAN happen (important!)

Only if you do **one of these**:

- Mix Hibernate with direct Redis/Ehcache access
- Read cache outside a transaction
- Use `StatelessSession`
- Use native SQL + cache reads
- Use async / eventual cache invalidation
- Assume query cache = entity cache (it isn't)

Hibernate alone **will not** create inconsistency.

9️⃣ Practical best practices (2026-ready)

✅ Use L2 cache for

- Read-mostly reference data
- Lookup tables (Country, Currency, TimeZone)
- Expensive lazy associations
- Reducing DB load across requests

❌ Avoid L2 cache for

- Highly mutable entities

- Strong consistency requirements
- Large collections with frequent updates

🧠 Golden rules

- Trust the database for correctness
- Trust Hibernate L1 for transactional consistency
- Treat L2 cache as a **performance hint**, not a truth source

---

1️⃣0️⃣ One-screen mental cheat sheet

```
SQL decides correctness
L1 cache decides identity
L2 cache decides performance
```

Or even shorter:

> **Hibernate never guesses. It always asks SQL first.**

If you want, next we can:

- compare Hibernate L2 vs Redis
- explain why query cache is often a foot-gun
- walk through a real Hibernate log example
- or design a “safe caching strategy” for your use case

You’ve clearly been thinking about the *right* problems — this was a deep one 👍

You said Query cache → IDs + invalidation timestamps (not rows). I have query that has no parameters (repository.findAll()) and i need this lookup list to send client and it is important to me that this query should not

be run over and over again. Does not hibernate cache all List<Country> then?

Great question — this is where **Hibernate's query cache is often misunderstood**, so let's be very precise and practical.

Short answer (no suspense)

> **No, Hibernate does *not* cache `List<Country>` as objects.**
> It caches **the list of IDs**, plus an invalidation mechanism.

Even for `findAll()` with no parameters.

What Hibernate actually caches for `findAll()`

Assume:

```
@QueryHints(@QueryHint(name = "org.hibernate.cacheable", value = "true"))
List<Country> findAll();
```

When query cache is enabled:

What is stored in query cache

```
QueryKey:
  "from Country"
CachedValue:
  [1, 2, 3, 4, 5]
  + timestamp / region version
```

That's it.

❌ Not:

- `Country` objects
- column values
- entity snapshots

What happens on a query cache hit

Second call to `findAll()`:

1. Hibernate finds cached IDs `[1..5]`
2. For each ID:
   - Check L1 cache
   - Else check L2 entity cache
   - Else hit DB by PK

So:

Cache state	DB activity
L2 entity cache warm	**0 SQL**
L2 partially warm	few PK selects
L2 cold	N selects

Why Hibernate does it this way (must-know)

Hibernate **cannot** safely cache rows because:

- Query results depend on transaction snapshot
- DB isolation rules would be violated
- Locking / version checks would break
- Different DBs behave differently

So Hibernate keeps:

- **structure** (IDs)
- **invalidation logic**
- but not **row state**

Your requirement: "This query must not run over and over again"

Option 1️⃣ Hibernate L2 + query cache (OK, but limited)

Works well **only if**:

- `Country` is rarely updated
- L2 entity cache is enabled
- Cache invalidation cost is acceptable

Still:

- Hibernate must iterate IDs
- Might hit DB if L2 evicts

Option 2️⃣ Read-through external cache (recommended)

If:

- This is a pure lookup list
- Sent to clients often
- Must be extremely fast
- Can tolerate eventual consistency

Then **Hibernate query cache is the wrong tool**.

Instead:

```
@Cacheable("countries") public List<CountryDTO> getAllCountries() { return repository.findAll(); }
```

Using:

- Redis
- Caffeine
- Hazelcast

This:

- Caches the **actual list**
- Avoids Hibernate completely
- Is simpler and faster

Option 3️⃣ Database-level solution (often overlooked)

For truly static data:

- Materialized view
- Read replica
- DB cache does the heavy lifting

Hibernate cache is not meant for “API response caching”.

When Hibernate query cache *does* make sense

- Pagination queries
- Complex joins
- Search results
- Reporting queries
- Read-mostly datasets with frequent re-use

Even then:

> Query cache is a **multiplier** for entity cache, not a replacement.

Final practical guidance

If your goal is:

Goal	Use
Avoid repeated SQL	Query cache
Avoid repeated entity loads	L2 entity cache
Avoid Hibernate entirely	External cache
Serve API responses fast	External cache + DTO

One-liner to remember

> **Hibernate query cache remembers *what* to load, not *the data itself*.**

If you want, tell me:

- update frequency of `Country`
- max size
- SLA / latency needs

I can suggest the **exact caching stack** that fits your case best