https://cw.fel.cvut.cz/b251/courses/b4m36ds2/start

Lecture 12

Transactions
Query Optimization

Mgr. **Yuliia Prokop**, Ph.D.

prokoyul@fel.cvut.cz

Telegram **@Yulia_Prokop**

Czech Technical University in Prague, Faculty of Electrical Engineering

Lecture outline

1. Transactions

- How each system handles transactions.
- What guarantees each system gives (and does NOT give).
- See where transactions really matter in analytical pipelines.
- Learn to combine transactions with good data modeling instead of relying only on ACID.

2. Optimization

- How to evaluate query performance and how to optimise queries.
- How to inspect the query plan or performance metrics.
- Practical Optimization techniques.

Transactions in Redis, MongoDB, Cassandra, and Neo4j

*A **transaction** is a **group** of operations that the database treats as a **single unit** of work: either all changes become visible together, or none of them do.*

*In some NoSQL systems, **transaction** may mean atomic execution without interleaving (but without rollback), so guarantees differ.*

ACID as reference (relational DBS)

- **Atomicity:**
  - all operations in a transaction succeed or none do.
- **Consistency:**
  - constraints hold before and after the transaction.
- **Isolation:**
  - concurrent transactions do not see each other's partial work.
- **Durability:**
  - committed changes survive crashes.

Classic RDBMS: multi-statement transactions, rollback on error, configurable isolation levels.
NoSQL systems partially implement these ideas with different scope and costs.

Transactions in Redis

- Command execution is single-threaded per Redis instance (event loop); commands are executed sequentially, so each command is **atomic**.
- When a client executes one INCR, HSET or ZINCRBY command, no other command can interleave in the middle of it.
- This is enough for many movie-analytics operations.
  - *For example: increasing the tag counter for one movie or updating the score of one actor.*
- However, a sequence of several commands is not automatically atomic as a group, so we need extra tools when multiple keys must change together.
- Redis provides **MULTI** and **EXEC** to group several related commands into one transaction block.
- Since Redis 6.0, I/O can be multi-threaded, but command execution remains single-threaded.

Transactions in Redis

- **MULTI** starts a transaction block.
- Commands are **queued** after MULTI, not executed.
- **EXEC** executes all queued commands **sequentially** and **without interleaving** from other clients.
- **DISCARD** – cancel the transaction.
  - All commands either run (if no prior error) or block is aborted before any run (e.g. after DISCARD)
- **WATCH** – implement **optimistic locking** for the read-modify-write cycle (protects reads that happen before MULTI)
  - If any watched key changes before EXEC, the transaction aborts.
- MULTI/EXEC guarantees that **no other client's command interleaves** between the commands in the transaction.
- **No rollback**
  - Errors can occur at queue time or at execution time; Redis does not roll back successful commands in EXEC.
- WATCH + MULTI/EXEC gives a **check-and-set** pattern
  - If a watched key changed, EXEC returns nil and the client can retry.

Transactions in Redis: example

```
MULTI

RPUSH search:log "user:42 | query: The Matrix | tmdb_id:603"
HINCRBY search:counts tmdb:603 1

EXEC
```

- We log that user 42 searched for the movie “The Matrix” (tmdb_id 603) and at the same time increase its search counter.
- Other clients cannot observe intermediate states, because no commands interleave during EXEC.

Transactions in Redis: example

User **42** rates movie **603** with **4.5**. We must update:

- raw “already rated” marker (so user can’t rate twice),
- per-movie rating summary,
- per-actor rating summary (lead actor **tmdb:6384**)
- **all together, only if it’s the first rating from this user**.

```
WATCH rated:movie:603

SISMEMBER rated:movie:603 42
#  if = 1 -> UNWATCH and return "already rated"

MULTI
SADD rated:movie:603 42
HINCRBY movie:603:rating count 1
HINCRBYFLOAT movie:603:rating sum 4.5
HINCRBY actor:6384:rating count 1
HINCRBYFLOAT actor:6384:rating sum 4.5
EXEC
#  if EXEC returns nil -> someone changed rated:movie:603 -> retry
```

Actor 6384 (Keanu Reeves) is the lead actor in movie 603 (The Matrix)

rated:movie:603 – SET of users who already rated this movie
movie:603:rating – HASH {count, sum}
actor:6384:rating – HASH {count, sum}

Redis transactions: two types of errors and no rollback

1. Queue-time errors (before EXEC)

- If a command has a syntax error or uses an unknown command while the transaction is being queued after MULTI, Redis marks the transaction as **invalid**.
- When the client then calls EXEC, Redis does **not** execute any of the queued commands and returns an error for the whole transaction.
- In this case nothing is written to the database.

2. Execution-time errors (during EXEC)

- After EXEC, Redis runs the queued commands one by one; some of them may succeed and some may fail (for example, a type error on one key).
- Commands that succeed **stay applied**; Redis does **not** roll back the successful updates if another command in the same transaction fails.

Redis transactions on a single instance versus clusters

- Redis transactions operate on a single Redis instance and do not coordinate across multiple nodes in a cluster.
  - In Redis Cluster, MULTI/EXEC works only when all keys are in the same hash slot (e.g., {movie:603}:counter, {movie:603}:log).
  - Curly braces **{}** define a **hash tag** – Redis Cluster uses only the content inside braces to compute the hash slot.
- In practice, we design our key space so that strongly related keys that must be updated together share the same instance.
- For cross-cluster consistency, we usually rely on idempotent updates and higher-level application logic rather than on Redis transactions.

When Redis transactions are useful

- Redis transactions are most useful when a single logical user action needs to update several small, related pieces of data that must stay consistent.
- Typical examples are writing a log entry and updating one or more counters at the same time, or updating several keys that represent different views of the same entity.
- Another example is adjusting several ranking sorted sets together so that all rankings reflect the same logical change.
- For large analytical aggregations we usually do not rely on Redis transactions; instead we recompute or stream aggregates elsewhere and write only the final results into Redis.

Transactions in MongoDB

- **Single-document** operations (insert/update/replace) are **atomic** by design.
- **Multi-document ACID transactions:**
  - Snapshot isolation within the transaction.
  - All writes in the transaction **commit together** or are rolled back.
- Transactions run in the context of a **session.**
- You get **ACID guarantees** across multiple documents:
  - either all writes inside the transaction are committed,
  - or all of them are rolled back.
- More coordination and overhead, so you don’t want to wrap every single write in a transaction for no reason.
- Good design tries to:
  - Use single-document atomicity where possible.
  - Use multi-document transactions **only** where cross-document consistency is really required (e.g., financial transfers).

Transactions in MongoDB: example

Embedding per-movie rating summary

- Besides raw rating events, we store a small summary directly inside the movie document: **rating_count** and **rating_sum**.
- Each time a user rates a movie, we atomically increment these fields in the same document.
- Because all updates occur within a single document, MongoDB guarantees that rating_count and rating_sum remain internally consistent without a multi-document transaction.

```
db.movies.updateOne(
  { tmdb_id: 603 },
  { $inc: { rating_count: 1, rating_sum: 4.5 } }
)
```

Basic structure of a multi-document transaction

- Some invariants involve several collections, e.g., raw rating events in ratings and per-movie aggregates in movie_stats.
- MongoDB supports ACID transactions across multiple documents and collections (replica sets and sharded clusters).
- The application creates a **session**, starts a transaction, runs all operations **with this session**, and then either commits or aborts.

```
const s = client.startSession();
try {
  s.startTransaction();
  await op1({ session: s });
  await op2({ session: s });
  // read and write operations using { session: s }
  await s.commitTransaction();
} catch (e) {
  await s.abortTransaction();
} finally {
  await s.endSession();
}
```

In practice, drivers often provide helper APIs (e.g., withTransaction) to handle retries.

Basic structure of a multi-document transaction

- `startSession()` – opens a session that will own the transaction context.
- `startTransaction()` – begins the transaction; reads use a consistent view and writes are staged until commit.
- `op1/op2(..., { session: s })` – every read/write that must be part of the transaction **must include the same session**.
- `commitTransaction()` – makes all transaction writes visible together (all-or-nothing).
- `abortTransaction()` – discards all writes from this transaction if an error occurs.
- `endSession()` – always close the session (cleanup), even on errors.
- `await` ensures each DB operation finishes before moving to the next step (especially before commit/abort).

Transactions in MongoDB: example

- When a user rates a movie, we may want to store both the raw rating and an updated aggregate in a separate collection.
- Without a transaction, a crash could leave a raw rating event without updated aggregates (or vice versa).

```
const s = client.startSession();
try {
  s.startTransaction();
  await db.ratings.insertOne({ user_id, tmdb_id: 603,
    rating: 4.5, created_at: now }, { session: s });
  await db.movie_stats.updateOne( { tmdb_id: 603 },
    { $inc: { rating_count: 1, rating_sum: 4.5 } },
    { upsert: true, session: s }
  );
  await s.commitTransaction();
} catch (e) {
  await s.abortTransaction();
} finally {
  await s.endSession();
}
```

When MongoDB transactions are worth using

- Multi-document transactions are valuable when a business rule requires several collections or documents to change together and remain consistent.
  - For example, user creation plus a profile document plus initial movie preferences.
- They are also useful during schema migrations, when old and new structures must be updated in sync for a limited period of time.
- For high-volume workloads with many small events, we usually prefer idempotent writes and background aggregation instead of wrapping every single event in a transaction.
- In practice we use transactions only for low-frequency, high-value operations that truly require ACID guarantees and are not easily solved by better data modelling.
- Isolation controls are limited compared to many RDBMS; transaction semantics focus on snapshot isolation.

Limitations and differences from relational ACID

- MongoDB transactions are usually limited in size and duration.
  - Large or long-running transactions consume memory for snapshots and may be aborted.
  - Default transaction timeout: 60 seconds.
  - WiredTiger cache pressure increases with long transactions.
- Transactions do not magically fix a poor data model
  - If we constantly need multi-document transactions for common operations, our document design is probably wrong.
- Compared to relational databases, MongoDB does not offer a rich set of isolation levels
  - It focuses on snapshot isolation within a transaction and simpler concurrency semantics.
- In a distributed cluster, cross-shard transactions are more expensive than single-document operations.

Transactions in Cassandra

- Cassandra **does not** support general ACID transactions across arbitrary rows like an RDBMS. Instead, it offers:

- **Lightweight Transactions (LWT)** via IF conditions on INSERT/UPDATE/DELETE.
  - Implemented using Paxos consensus.
  - Guarantee a linearizable compare-and-set behavior.
  - Much slower than normal writes, so used only for rare operations that require strong consistency (e.g., unique username).

- **BATCH:**
  - Group multiple writes.
  - Single-partition batches are UNLOGGED (default). Multi-partition batches are LOGGED (use batchlog, more overhead).
  - BATCH is not a general transaction mechanism. It provides atomicity for mutations within a partition; multi-partition batches add overhead and do not give RDBMS-like isolation.
  - Large multi-partition batches are an anti-pattern.

Example: Lightweight Transactions (LWT) for unique actor slugs

- Some operations in the movie domain require uniqueness, such as a human-readable slug for each actor.
- Cassandra offers Lightweight Transactions using **IF** conditions to ensure that only one of several concurrent writes succeeds.
- Internally this uses a **consensus protocol** and is much slower than normal writes, so it should be used sparingly.

```
INSERT INTO actors_by_slug (slug, actor_id, name)
VALUES ('keanu-reeves', 6384, 'Keanu Reeves')
IF NOT EXISTS;
```

- This example creates a record for actor 6384 with slug keanu-reeves only if that slug is not already taken by another actor.
- If two clients try to insert the same slug at the same time, only one of them will succeed and the other will see [applied] = false.

Example: Atomic BATCH within one partition

- Use BATCH when you need to update **multiple rows** that share the **same partition key**, and you want those row-level mutations to be applied together **within that partition**.
- Example schema (two metrics stored as two rows):

```
CREATE TABLE movie_stats_by_day (
  tmdb_id int,
  day date,
  metric text,      -- 'count' | 'sum'
  value counter,
  PRIMARY KEY ((tmdb_id, day), metric)
);
```

Event: user rates movie 603 with 4 on 2025-12-05

Both updates target the same partition key:

```
(tmdb_id=603, day='2025-12-05')
```

Example: Atomic BATCH within one partition

Event: user rates movie 603 with 4 on 2025-12-05
Both updates target the same partition key: (tmdb_id=603, day='2025-12-05')

```
BEGIN COUNTER BATCH
  UPDATE movie_stats_by_day
  SET value = value + 1
  WHERE tmdb_id = 603 AND day = '2025-12-05' AND metric = 'count';

  UPDATE movie_stats_by_day
  SET value = value + 4
  WHERE tmdb_id = 603 AND day = '2025-12-05' AND metric = 'sum';
APPLY BATCH;
```

- BATCH updates **two different rows** (metric='count' and metric='sum').
- Both rows are in the **same partition** (tmdb_id, day).
- The batch provides **atomicity** (no isolation!) **within that partition**: the partition does not end up with only one of the two metric updates applied.
- If you are updating fields in the **same row**, prefer a **single UPDATE** instead of a BATCH.

Transactions in Cassandra: limitations

- Cassandra does not provide cross-row transactions with rollback and isolation levels like a relational database.
- LWT **is good** for:
  - Ensuring **uniqueness** (usernames, reservation IDs)**.**
  - Guarding updates with conditions (“update only if version = X”).
- LWT **should not** be used for every write:
  - Significantly higher **latency** because of extra coordination round-trips.
- BATCH is **not** a general transaction mechanism for unrelated partitions
  - It’s mostly about efficiency and grouping writes, plus per-partition atomicity.

Designing tables instead of relying on transactions

- We push consistency requirements into the table design.
  - For example, we might have a table movie_ratings_by_user and a separate table movie_ratings_by_movie, each optimized for its own query.
  - Both tables are updated directly when a new rating event arrives (in the application), without a multi-row transaction.

```
CREATE TABLE movie_ratings_by_user  (
user_id int,
tmdb_id int,
rated_at timestamp,
rating double,
PRIMARY KEY ((user_id), rated_at, tmdb_id)
);
```

```
CREATE TABLE movie_ratings_by_movie (
tmdb_id int,
user_id int,
rated_at timestamp,
rating double,
PRIMARY KEY ((tmdb_id), rated_at, user_id)
);
```

- If an occasional inconsistency occurs, we handle it with idempotent updates or background repair jobs rather than locking the entire system.

Transactions in Neo4j

- Full ACID transactions around Cypher.
- Every Cypher query that modifies the graph runs in a **transaction**.
  - If there's no explicit transaction, Neo4j will **start one, run the query, and commit it** automatically (auto-commit).
- Drivers and tools like cypher-shell allow **explicit** transaction control:
  - `:begin, :commit, :rollback` in cypher-shell.
- Neo4j gives **ACID guarantees** for each transaction: either the whole graph change is committed, or nothing is.
- It's natural to think in terms of "graph operations" per transaction:
  - Example: "create user + connect to groups + log event" is a single transaction.

Transactions in Neo4j : example

In cypher-shell:

```
:begin
MATCH (u:User {id: $userId})
CREATE (t:Ticket {id: $ticketId})
CREATE (u)-[:BOUGHT]->(t)
:commit
```

Using an **auto-commit** query:

```
MATCH (u:User {id: $userId})
CREATE (t:Ticket {id: $ticketId})
CREATE (u)-[:BOUGHT]->(t)
```

Both run in a transaction; the difference is:

- With explicit :begin / :commit, you can group **multiple Cypher statements** in one transaction.
- With auto-commit, each statement is its own transaction.

Transactions in Neo4j : example

- When we import new movies and actors, we often want to create nodes and relationships together.
- The following example guarantees that the movie “The Matrix”, the actor Keanu Reeves and the PLAY relationship between them appear together or not at all.

```
:begin
MERGE (m:MOVIE {tmdb_id:603, title:'The Matrix'})
MERGE (p:ACTOR {tmdb_id:6384, name:'Keanu Reeves'})
MERGE (m)-[:PLAY]->(p)
:commit
```

Transaction costs and best practices

- Very large transactions that touch many nodes can consume a lot of resources.
- We usually split heavy imports or graph updates into many small transactions instead of one huge one.
- Each transaction should perform a focused part of the work.
  - For example: importing one movie with its cast or computing similarity for one target user.
  - This keeps transaction logs small, reduces the risk of timeouts and fits better into the Neo4j execution model.

Comparing Neo4j and relational transactions

- Like relational databases, Neo4j offers atomicity, consistency, isolation and durability for graph operations.
- The units of work are usually graph patterns that combine several nodes and relationships rather than rows in tables.
- We still avoid long-running transactions and prefer short, well-defined updates.
- For analytical queries over the graph we rely more on read-only transactions and graph algorithms than on complex write transactions.

Consistency & isolation – what you can actually control

- In distributed deployments, explicitly define consistency per stage (write path, read API, analytics, cache).
- **MongoDB:** choose writeConcern, readConcern, and readPreference (e.g., majority for correctness vs. secondary reads for scale).
  - Available readConcern levels: local, majority, snapshot, linearizable.
- **Cassandra:** choose consistency level per operation (ONE, LOCAL_QUORUM, QUORUM, ALL); use **LWT** only for conditional uniqueness / “first write wins”.
- **Redis:** single commands are atomic; MULTI/EXEC provides atomicity on a single node (not a distributed transaction).
- **Neo4j:** ACID transactions; in clusters use driver routing/causal consistency (bookmarks) to avoid stale reads after writes.
- Rule of thumb: use the weakest consistency that still preserves the required business invariant.

Why distributed transactions are expensive

- Multi-node transactions require coordination (consensus/2PC-like overhead) → higher latency and lower throughput.
- Conflicts cause abort/retry loops (optimistic concurrency), which amplifies load under spikes.
- Long transactions increase resource holding time (locks/state/memory) and reduce concurrency.
- Stronger durability/replication (e.g., majority acknowledgements) increases write latency.
- Prefer alternatives when possible: keep transactions local, use idempotency + de-dup markers, and use sagas/outbox for cross-service workflows.

Query Evaluation and Optimization

Measuring performance in Redis

- Redis is very fast, but poorly designed data structures can still create bottlenecks in analytical tasks.
- For performance diagnostics Redis provides tools such as the slowlog, latency graphs and simple timing from the client.
  - **Built-in latency diagnostics:** LATENCY DOCTOR, LATENCY LATEST, LATENCY HISTORY <event>
  - **Inspect slow commands:** SLOWLOG GET <N> (and SLOWLOG LEN, SLOWLOG RESET when needed)
  - **Measure end-to-end latency:** report p50/p95/p99 from the application side
  - **Track key memory hotspots:** MEMORY USAGE <key> and INFO memory (when investigating growth)
- The slowlog records commands that exceed a configurable execution time threshold.
- This example shows the last ten slow commands so we can see which operations on movie or actor keys are unexpectedly expensive.

```
SLOWLOG GET 10
```

Optimization technique 1: pre-aggregated sorted sets

- A common analytical question is “Which movies received the most tags during the last week?”.
- Scanning raw events in MongoDB or Cassandra each time is too expensive for a real-time dashboard.
- In Redis we can maintain one sorted set per day. The following pattern pre-aggregates daily tag counts per movie and then uses ZUNIONSTORE to quickly compute weekly top movies.

```
ZINCRBY movie_tags:2025-12-05 1 "tmdb:603"
ZINCRBY movie_tags:2025-12-05 1 "tmdb:862"
ZUNIONSTORE movie_tags:week 7 movie_tags:2025-11-29
    movie_tags:2025-11-30 movie_tags:2025-12-01
    movie_tags:2025-12-02 movie_tags:2025-12-03
    movie_tags:2025-12-04 movie_tags:2025-12-05
ZRANGE movie_tags:week 0 19 REV WITHSCORES
```

ZUNIONSTORE materializes a new zset and requires memory.

Optimization technique 1: pre-aggregated sorted sets (cont.)

Materialized weekly top lists

- **Set TTL on materialized keys** to avoid unbounded growth, e.g.

  `EXPIRE movie_tags:top:week:2025-50 1209600`

  (keep 14 days).

- **Clarify refresh strategy:** recompute periodically (e.g., every hour/day) and treat it as **cache** derived from the “source of truth”.

- **Use deterministic key naming** for time windows:

  `movie_tags:top:week:<ISO_YEAR>-<ISO_WEEK>`

  (easy to rotate and debug).

Optimization technique 2: using hashes instead of many small keys

- Another typical task is to count events per genre or per actor country for movies.
- Storing each counter as a separate key leads to many keys and higher memory overhead.
- Redis hashes allow us to group related counters under one logical key and update them efficiently.
- In the example one hash holds all genre counters, which is more compact and easier to manage than thousands of separate keys.

```
HINCRBY stats:genre "Action" 1
HINCRBY stats:genre "Drama" 1
HINCRBY stats:genre "Comedy" 1
HGETALL stats:genre
```

Evaluating MongoDB queries with explain

- MongoDB provides the explain() method to inspect how a query will be executed.
- The **executionStats** mode returns the chosen plan and important metrics, such as the number of examined documents and index keys.
- High values of **totalDocsExamined** or **totalKeysExamined** indicate that the query is scanning too much data.
- This example reveals whether the query “last 50 ratings for The Matrix” does a COLLSCAN, whether it uses an index for filtering, and whether it still needs an in-memory sort.

```
db.ratings.find({ tmdb_id: 603 }).sort({ created_at: -1
}).limit(50)
  .explain("executionStats")
```

Evaluating MongoDB queries with explain

```
{
  queryPlanner: {
    namespace: "db.ratings",
    parsedQuery: { tmdb_id: { $eq: 603 } },
    winningPlan: { ... },
    rejectedPlans: [ ... ]
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 50,
    executionTimeMillis: 3,
    totalKeysExamined: 50,
    totalDocsExamined: 50,
    executionStages: { ... }
  },
  serverInfo: { ... },
  ok: 1
}
```

For this result, a compound index `{tmdb_id: 1, created_at: -1}` is needed.

Optimization technique 1: single-field index for filters

- Many analytical queries filter by one field, for example all ratings for a given movie.
- Without an index on this field MongoDB will scan the entire collection, which is slow for large datasets.
- Creating an index on the filter field allows the query engine to jump directly to relevant documents.

```
db.ratings.createIndex({ tmdb_id: 1 });

db.ratings.find({ tmdb_id: 603 }).limit(100)
  .explain("executionStats")
```

If you also sort by created_at, consider a compound index to avoid an in-memory sort.

Indices in MongoDB: https://cw.fel.cvut.cz/wiki/_media/courses/b4m36ds2/b4m36ds2-lecture-8-mongodb2025.pdf

Optimization technique 2: compound index

- Some queries combine equality, sorting and a range filter, for example a timeline of user actions.
- Example: “find all actions of user X for the last seven days sorted by time”.
- A compound index that respects the order “equality → sort → range” can support such queries efficiently. When sort and range operate on the same field, the index supports both operations.
- This example shows a compound index optimised for a user’s recent movie actions: first equality on user_id, then sorting and range on created_at.

```
db.events.createIndex({ user_id: 1, created_at: -1 });

db.events.find({
  user_id: 42,
  created_at: { $gte: ISODate("2025-12-01") }
})
.sort({ created_at: -1 }).limit(200)
```

Pitfalls to avoid in MongoDB Optimization

- Over-indexing can hurt write performance because each insert or update must maintain multiple indexes.
- Indexes that do not match any realistic query pattern only consume memory without bringing benefits.
- When a query still performs badly after indexing, we reconsider the document structure or the aggregation pipeline instead of adding more random indexes.

Evaluating Cassandra queries with tracing

- Cassandra does not have a traditional query planner, but it provides a tracing facility for analysing how a query is executed.
- When tracing is enabled, each query writes detailed information about contacted nodes, consistency level, and the number of read rows.
- This helps to detect queries that accidentally scan many partitions or use ALLOW FILTERING.
- This example enables tracing for a query that reads events for one user on a given day, so we can check whether it hits a single partition or scans a larger part of the cluster

```
TRACING ON;
SELECT * FROM movie_events
WHERE user_id = 42
  AND day = '2025-12-05';
TRACING OFF;
```

As a result, you always get:

- the **normal result set**, plus
- a **trace log** with per-step activities and timings.

In this example, user_id is an INT for readability; in a real system it would typically be a UUID.

Evaluating Cassandra queries with tracing

```
 user_id | day        | ts                              | event_type | ...
---------+------------+---------------------------------+------------+-----
      42 | 2025-12-05 | 2025-12-05 10:15:01.123         | rating     | ...
      42 | 2025-12-05 | 2025-12-05 10:16:37.987         | tag_added  | ...
(2 rows)

Tracing session: 3e817cf0-...-...

 activity                                  | timestamp                  | source    | source_elapsed
-------------------------------------------+----------------------------+-----------+----------------
 Parsing SELECT statement                  | 2025-12-05 10:20:00.001    | 10.0.0.1  |  0
 Preparing statement                       | 2025-12-05 10:20:00.003    | 10.0.0.1  |  200
 Executing single-partition query on ...   | 2025-12-05 10:20:00.005    | 10.0.0.2  |  400
 Read 2 live rows and 0 tombstones         | 2025-12-05 10:20:00.006    | 10.0.0.2  |  700
```

Tracing adds overhead; use for diagnostics, not continuously in production.

How many partitions are being read? - **Executing single-partition query only – OK**
How many rows are read? - **Read 2 live rows and 0 tombstones – OK**
How long did it take? - **source_elapsed = 700 – OK**
Which nodes were involved? - **source**

Evaluating Cassandra queries with tracing: anti-example

```
CREATE TABLE movie_events (
  user_id    int,
  day        date,
  ts         timestamp,
  event_type text,
  tmdb_id    int,
  PRIMARY KEY ((user_id, day), ts)
);

TRACING ON;
SELECT * FROM movie_events
WHERE event_type = 'rating'
  AND day = '2025-12-05'
ALLOW FILTERING;
TRACING OFF;
```

```
CREATE TABLE events_by_type_day (
  event_type text,
  day      date,
  ts       timestamp,
  user_id  int,
  tmdb_id  int,
  PRIMARY KEY ((event_type, day), ts)
);
```

The query searches for “all rating events on 2025-12-05”, but the partition key is (user_id, day), not (event_type, day).

Evaluating Cassandra queries with tracing: anti-example

```
user_id | day         | ts                                | event_type | tmdb_id
---------+-------------+-----------------------------------+-------------- +--------
   ...     | 2025-12-05 | 2025-12-05 09:12:01.111  | rating       |   603
   ...     | 2025-12-05 | 2025-12-05 09:13:22.543  | rating       |   862
(200 rows)

Tracing session: 91c5b6b0-...-...

 activity                                                  | source      | source_elapsed
-----------------------------------------------------------+-------------+----------------
 Parsing SELECT statement                                  | 10.0.0.1    |          0
 Preparing statement                                       | 10.0.0.1    |        300
 Executing read on ... for range of key values             | 10.0.0.2    |      50000
 Read 50000 live rows and 30000 tombstones                 | 10.0.0.2    |     120000
 ...
```

“Read 50000 live rows and 30000 tombstones” to return only **200 rows – BAD!**
source_elapsed is large **– BAD**

Optimization technique 1: query-specific tables instead of ALLOW FILTERING

- For analytical queries such as “all tag_added events on a given day,” we must create a dedicated table with a matching primary key.
- This table is populated at write time from the same event stream that feeds the generic log.
- In the following example, the partition **key (type, day)** ensures that all tag events for a given day and type are colocated and can be read efficiently without ALLOW FILTERING.

```
CREATE TABLE events_by_type_day (
  event_type text,
  day date,
  ts timestamp,
  event_id timeuuid,
  user_id int,
  tmdb_id int,
  tag_text text,
  PRIMARY KEY ((event_type, day), ts, event_id)
);
```

```
SELECT * FROM events_by_type_day
WHERE event_type = 'tag_added'
  AND day = '2025-12-05';
```

Optimization technique 2: clustering keys for sorted results

- Many analytical tasks require sorted results, for example, movie interactions ordered by time or ratings ordered by tmdb_id.
- In Cassandra, the order of clustering columns defines the natural sort order inside each partition.
- If we design the clustering keys to match the expected sort order, queries become efficient without extra sorting on the client.
- This example shows events for movie 603 on a given day, already sorted by timestamp, because ts is the clustering key.

```
CREATE TABLE movie_events_by_movie (
  tmdb_id int,
  day date,
  ts timestamp,
  event_id timeuuid,
  user_id int,
  event_type text,
  PRIMARY KEY ((tmdb_id, day), ts, event_id)
);
```

```
SELECT * FROM movie_events_by_movie
WHERE tmdb_id = 603
  AND day = '2025-12-05';
```

Cassandra: Typical Optimization guidelines

- Design a separate table for each important analytical query pattern instead of one generic “events” table.
- Avoid using ALLOW FILTERING in production; if a query needs it, treat that as a strong signal that the table design is wrong.
- Keep partitions at a reasonable size (Cassandra recommendation: up to 100MB, up to 100K rows per partition). Extremely large partitions for very “hot” keys can create performance hotspots.
- Use tracing and monitoring to verify that queries touch only a small number of partitions and read only the columns that are actually needed

Evaluating Cypher queries with PROFILE

- Neo4j provides the **PROFILE** keyword to show the execution plan and runtime statistics for a Cypher query.
- The plan reveals which operators are used, such as **NodeIndexSeek**, **NodeByLabelScan** or **Expand(All)**, and how many database hits they perform.
- Queries whose plan starts with an **unbounded NodeByLabelScan** are usually inefficient because they scan all such nodes in the dataset.
- This example shows how Neo4j executes a query that finds the top 20 movies for actor 6384 by popularity; we expect the plan to use an index on :ACTOR(tmdb_id):

```
PROFILE MATCH (a:ACTOR {tmdb_id: 6384})<-[:PLAY]-(m:MOVIE)
RETURN m
ORDER BY m.popularity DESC
LIMIT 20;
```

Evaluating Cypher queries with PROFILE

```
+--------------------+---------+---------+--------------------------------+
| Operator           | Rows    | DbHits | Details                         |
+--------------------+---------+---------+--------------------------------+
| Limit              | 20      | ...     | 20                             |
| Sort               | 100     | ...     | m.popularity DESC              |
| Expand(Into)       | 100     | ...     | (a)<-[:PLAY]-(m)               |
| NodeIndexSeek      | 1       | ...     | :ACTOR(tmdb_id) = 6384|
+--------------------+---------+---------+--------------------------------+
```

For a well-optimised query, we expect the plan to:

- start with a **NodeIndexSeek on :ACTOR(tmdb_id)** to find that single actor;
- then use an **expand** step from the ACTOR node to the related MOVIE nodes via [:PLAY];
- finally, apply **Sort** on m.popularity DESC and **Limit 20**.
- If the plan instead shows a **NodeByLabelScan on :ACTOR** (scanning all actors), the query is logically correct but **not well optimised**, and we should create an index on :ACTOR(tmdb_id) or adjust the model so the index is used.

Optimization technique 1: indexes on lookup properties

- Many graph queries start by looking up one movie, one actor, or one user by an ID from the dataset.
- Without an index Neo4j may scan all nodes with a given label and filter by property, which is slow for large graphs.
- In the following example, the indexes support fast lookup of movies and actors by their TMDB IDs, which is the common entry point for our queries.

```
CREATE INDEX movie_tmdb_index FOR (m:MOVIE) ON
(m.tmdb_id);
CREATE INDEX actor_tmdb_index FOR (p:ACTOR) ON
(p.tmdb_id);

MATCH (m:MOVIE {tmdb_id:603})
RETURN m;
```

For identifiers, prefer a **UNIQUENESS constraint** (which also provides an index).”

Optimization technique 2: start from the most selective node

- A good Cypher query starts from the most selective part of the pattern using an index, then expands via relationships.
  - For example, “movies co-starring with Keanu Reeves” should start from the actor node, not from all movies.
- This example begins with a selective lookup of actor 6384 from graph, then traverses only relevant PLAY relationships instead of scanning all movies.

```
MATCH (p:ACTOR {tmdb_id:6384})<-[:PLAY]-(m:MOVIE)-
[:PLAY]->(co:ACTOR)
RETURN co.name, count(m) AS shared_movies
ORDER BY shared_movies DESC
LIMIT 10;
```

Neo4j: Practical Optimization tips

- Create indexes on the small set of properties that are used as entry points into the graph, such as external IDs, usernames or other unique identifiers.
- Avoid query patterns that start with an unbounded label scan and then apply heavy filtering or sorting over all nodes of that label.
- Use PROFILE to verify that queries use NodeIndexSeek (or other index-based operators) where you expect them and to detect unnecessary expansions in the graph.
- For complex or frequently repeated analytics, consider precomputing and storing additional relationships or summary nodes instead of recomputing expensive traversal patterns on every request.

Micro-optimizations (small wins that add up)

- Project only needed fields (avoid fetching large documents/props you don’t use).
- Limit early: use tight predicates + LIMIT; avoid large sorts without supporting indexes.
- Prefer keyset pagination over OFFSET/SKIP for deep paging (stable + faster).
- Batch writes and use pipelining/bulk APIs where available.
- Use TTL for ephemeral data (Redis/Cassandra/MongoDB); keep hot keys/documents small.
- Avoid N+1 access patterns (prefetch/denormalize/cache selectively).

Optimization checklist (works for any NoSQL)

- Start from a concrete query pattern + expected result size; measure the baseline (SLOWLOG / explain() / TRACING / PROFILE).
- Reduce work first: filter early, project only needed fields, avoid scans (ALLOW FILTERING / NodeByLabelScan).
- Add the right access path: indexes, clustering keys, pre-aggregations (zsets), or query-specific tables/summary nodes.
- Validate with metrics (docs/keys examined, partitions touched, DbHits, p95 latency) and re-check after data grows.

General principles

Aspect	Redis	Cassandra	MongoDB	Neo4j
Main lever	Data structures	Data modeling	Indexes	Indexes + graph model
Bottleneck	Memory, network	Disk I/O, compaction	Working set, indexes	Traversal depth
Scaling	Cluster sharding	Linear horizontal	Sharding	Causal cluster
Profiling	SLOWLOG, INFO	nodetool, tracing	explain, profiler	PROFILE, EXPLAIN

Conclusions

- NoSQL systems provide different transaction scopes; don’t assume RDBMS-style ACID everywhere.
- Good data modeling + idempotency often beats “bigger transactions” for analytical pipelines.
- Optimization is measurement-driven: plans/traces are the source of truth.
- Avoid common anti-patterns (over-indexing, multi-partition batches, unbounded scans); precompute when it pays off.