FAKULTA ELEKTROTECHNICKÁ

České vysoké učení technické v Praze

B4M36DS2 – Database Systems 2

Practical Class 5

Types of NoSQL data stores

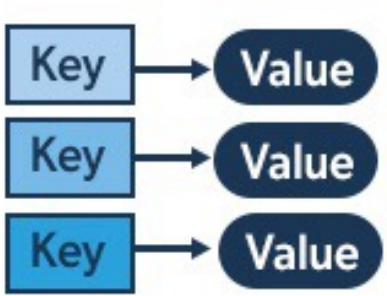
Yuliia Prokop

prokoyul@fel.cvut.cz, Telegram **@Yulia_Prokop**

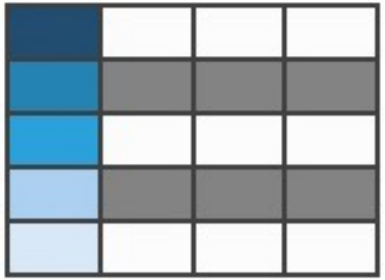
ČVUT FEL CourseWare Wiki https://cw.fel.cvut.cz/b251/courses/b4m36ds2/start

Types of NoSQL Databases

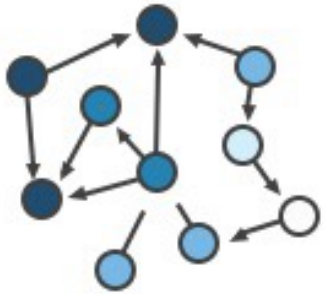
Key-Value



Column-Family



Graph



Document

Source: https://www.geeksforgeeks.org/types-of-nosql-databases/

Types of NoSQL data stores: key–value, in-memory

Key–value, in-memory (Redis)

Strengths (+)

- Extremely fast
- Perfect for counters and leaderboards
- Excellent caching layer
- Good for temporary or session data
- Very simple data model

Weaknesses (–)

- ✗ Filtering and search are limited
- ✗ No general-purpose query language in core
- ✗ Not designed for long-term or large-scale storage
- ✗ Durability is limited

When to use: caching, counters, rate limiting, real-time leaderboards, short-term statistics, and ephemeral queues.

Types of NoSQL data stores: wide-column

Wide-column, distributed storage (Cassandra)

Strengths (+)

- Very high write throughput
- Scales horizontally without downtime
- Excellent for time-series and activity logs
- High availability
- Predictable performance for predefined queries

Weaknesses (–)

- ✗ No joins at all
- ✗ Filtering is limited
- ✗ Limited query or analytics language
- ✗ Data model must be planned in advance (query-driven)

When to use: large event streams, logs, time-series data, fixed-pattern queries where keys are known in advance.

Types of NoSQL data stores: document-oriented

Document-oriented (MongoDB)

Strengths (+)

- Flexible schema
- Powerful filtering and aggregation
- Secondary indexes
- Suitable for moderate-scale analytics
- Stores complex objects naturally

Weaknesses (–)

- ✗ Not ideal for extremely high write rates
- ✗ Slow for cross-collection operations
- ✗ Less efficient for very long time series
- ✗ Performance depends heavily on indexes

When to use: flexible analytical reports by attributes or time, content metadata, recommendations, and moderate aggregations.

Types of NoSQL data stores: graph database

Graph database (Neo4j)

Strengths (+)

- Optimized for relationships
- Excellent for recommendations and community analysis
- Flexible node and edge types
- Efficient for multi-step traversals
- Intuitive model

Weaknesses (–)

- ✗ Not suited for large-scale tabular reports
- ✗ Horizontal scaling limited
- ✗ Not designed for time-series or streaming data
- ✗ Updating large subgraphs is expensive
- ✗ High memory use

When to use: relationship-driven recommendations, social and content networks, pathfinding, and community detection.

Types of NoSQL data stores

System	Core model	Best for	Avoid for	Key strengths	Main limitations
Redis	Key–value in memory	Caching, counters, leaderboards, rate limits	Analytics, filters, history	Ultra-fast, simple	No filtering, limited durability
Cassandra	Wide-column (distributed)	High-volume writes, time-series	Ad-hoc analytics, joins	Scalable, predictable, always on	No joins, limited filtering
MongoDB	Document	Flexible queries, metadata analytics	Extreme write loads, deep joins	Powerful filtering, flexible schema	Slower for cross-collection queries
Neo4j	Graph	Recommendations, relationships, path queries	Bulk aggregation, time series	Relationship traversal, intuitive model	Hard to scale horizontally

Example 1

1. Count ratings by users from a given country during the previous calendar month

Primary DBMS: MongoDB (document)

- **Attribute + time filtering in one place**

  The task **selects** events by a user attribute (country) and by a precise calendar window (previous month). A document store keeps flexible fields together, so you can filter directly on attributes and timestamps without redesigning storage.

- **Multi-level aggregation without precomputation**

  The result needs per-user totals and each user's percentage of the country's total. Document analytics can compute group totals, overall totals for the same filter, and derived percentages within a single server-side aggregation pass.

- **Monthly batch favors flexibility**

  This runs once per month, so a clear analytical pipeline and freedom to add new breakdowns later (e.g., language, registration cohort) are more valuable than extreme ingestion throughput.

Example 1 - 2

Alternative

Cassandra (Wide-column store)

- **How it would work:** Maintain **during ingestion** per-country, per-month, per-user counters (i.e., the exact totals your report needs). Reading the monthly report then becomes a direct lookup of those pre-rolled totals.
- **When to choose it:** When rating **volume is very high** and the report shape is **stable and repeated** (always "country × previous month × per-user"), so the cost and rigidity of pre-aggregation are justified by predictable, low-latency reads at massive scale.
- **Trade-off:** Cassandra does not support ad-hoc filtering across arbitrary attributes; if you later need new dimensions (e.g., add "city" or "platform"), you must **extend ingestion-time pre-aggregation** or redesign tables.

Why are others weaker?

Redis: No filtering capabilities. Can only retrieve values if the key is known. Not suitable for analytical queries or grouping by country.

Neo4j focuses on paths, which are not needed here.

Example 1 - 3

Expected Output Format

user_id | username | display_name | number_of_ratings | percentage_of_country_ratings

Sorted by ratings_count_in_month (top 50).

Freshness / Latency Goal

Runs at the beginning of each month. Response time of seconds to a minute is acceptable.

Access Pattern

- Filter by user.country = selected country
- Filter by timestamp ∈ previous month
- Group by user
- Sort by count (top 50)

Example 2

2. Timeline of all actions for a specific user in the last 7 days

Primary DBMS: Cassandra (wide-column)

- **Per-user, time-ordered storage**

The query always targets one user and a short, recent time window. A wide-column layout can place all events for that user together and order them by timestamp, so a 7-day slice is a single contiguous read rather than many scattered lookups.

- **Sustained high write throughput**

Action events arrive continuously. Cassandra is designed to absorb steady streams of small writes without locks, keeping recent data hot and readable.

- **Natural ordering for output**

Because events are clustered by time within the user's partition, returning them from newest to oldest requires no expensive resorting at query time.

Example 2 - 2

2. Timeline of actions for one user (last 7 days)

Alternatives:

MongoDB

- **How it would work:** Store events as documents with user ID and timestamp fields; filter by user and by the last 7 days, then sort.
- **When to choose it:** When volumes are moderate **and** you also need richer attribute-based filters within the same timeline (for example, “only ratings and tags,” or “only actions on documentaries”). MongoDB’s flexible filtering is valuable in that case.
- **Trade-off:** At very high write rates or with many concurrent timeline reads, Cassandra’s time-clustered layout typically remains more predictable.

Why are others weaker?

Neo4j is for paths; this is a linear time series.

Redis has no filtering and is not a reliable historical store for a period.

Example 2 - 3

Expected Output Format

timestamp | action_type | target_kind (movie/person/playlist/review) | target_id | optional details

Freshness / Latency Goal

On-demand; sub-second to a few hundred milliseconds for typical recent windows.

Access Pattern

- Filter by user_id = selected_user
- Restrict by timestamp ∈ [now−7d, now]
- Read a contiguous range within the user's time-ordered events; return newest first

4. Top movies by new tags (past 7 days)

Primary DBMS: Redis (in-memory key–value)

- **Continuously changing, small ordered list**
  The output is a short ranking (e.g., top 20) that must reflect each new tag event almost immediately. Keeping tiny per-movie counters in memory allows instant updates and fast retrieval from a maintained ZSET window.

- **Short, sliding time window**
  The seven-day window is relatively small and must be refreshed frequently. In-memory management of rolling windows (for example, by incrementing today's bucket and decrementing the bucket that just fell out) avoids heavy recomputation.

- **Interactive latency requirement**
  Product teams often surface this ranking in dashboards or modules that expect sub-second responses. An in-memory system is well-suited to this kind of usage.

Example 3 - 2

Alternative:

MongoDB

- **How it would work:** Store raw tag events with timestamps and compute the 7-day counts in a scheduled job (e.g., hourly or daily), persisting only the aggregated results for retrieval.
- **When to choose it:** When “live” freshness is not needed and a scheduled recomputation is acceptable; when you also need flexible retrospective analysis on the same tag events (by genre, by country, etc.).
- **Trade-off:** Scheduled recomputation introduces lag; you will not see each new tag immediately reflected in the top list.

Why are others weaker?

Cassandra

Can maintain per-movie, per-day counts and then assemble a 7-day sum, but achieving “always up-to-date” top-N requires additional sorting infrastructure or an external compute step. This is heavier operationally than an in-memory ranking for such a small result.

Neo4j not relevant—no traversal.

Example 3 - 3

Expected Output Format

A short ranking for the past 7 days:

rank | movie_id | title | new_tag_count_last_7_days

Freshness / Latency Goal

Near real-time; sub-second reads. The counts should reflect new tag events within moments of ingestion.

Access Pattern

- Global top over many movies within a fixed, short window
- Single read returns a tiny result set (e.g., 20 rows)

5. Users who applied the same tag to the same movie (last 30 days)

Primary DBMS: Neo4j (graph)

- **The problem is inherently relational**
  You are looking for groups of users who are connected to the same movie through the same tag. In a graph, users and movies are nodes, and each “tag added” action is a relationship labeled with the tag text. Finding “everyone connected to this movie via this tag” is a direct neighborhood query rather than a large table scan.
- **Group construction is local to the graph neighborhood**
  Once you anchor on a specific pair (tag text, movie), the group is simply the set of adjacent users connected by relationships carrying that tag label. Counting group size and collecting member lists is done without joining unrelated data.

Example 4 - 2

Alternative:

MongoDB

- **How it would work:** Treat each tag event as a document carrying user, movie, tag, and timestamp. For the last 30 days, group documents by (tag, movie) and collect the set of users per group. Filter groups by size (2–20), then format output.
- **When to choose it:** When you only need **periodic batch** results (monthly) and you do **not** plan to explore overlaps between groups, cross-group connections, or further community structure.
- **Trade-off:** As data grows, grouping by (tag, movie) across a month can be heavy. It remains feasible for batch, but less convenient if you later need graph-style explorations (e.g., users co-occurring across many tags and movies).

Why are others weaker?

Redis lacks filtering and set logic over large historical windows.

Cassandra – there are no joins and only limited filtering within a partition.

Example 4 - 3

Expected Output Format

For each (tag_text, movie_id) with 2–20 distinct users during the last 30 days:

tag_text | movie_id | movie_title | [list_of_user_ids] | [list_of_usernames] | group_size

Ordered primarily by tag_text (alphabetical), and within each tag by group_size descending.

Freshness / Latency Goal

Monthly (on the last day of the month). Seconds are acceptable since this is an analytical report, not a user-facing real-time feature.

Access Pattern

- Consider all tag events from the last 30 days
- Group by (tag_text, movie_id)
- For each group, gather users and apply size constraints (2–20)
- Sort as specified and emit the groups

Exercises

Task 4 – Top movies by new tags (top N)

- Identify which movies received the most new user tags during the past week to track emerging interests.

Task 6 – Most active reviewers (top N)

- List the users who wrote the highest number of reviews in the previous week to reward active contributors.

Task 7 – Average rating by genre (aggregate)

- Calculate the average user rating per genre during the last quarter to compare genre performance and identify trending or declining genres.

Task 8 – Growth of playlists (trend)

- Monitor how many new playlists users created each day in the last ninety days to see the growth of this feature.

Exercises

Task 9 – Subscriptions to persons (aggregate)

- Count how many users subscribed to a specific person (actor or director) during the last month to measure the person's popularity and inform marketing decisions.

Task 10 – Search term popularity (top N)

- Identify the most frequent search queries entered by users in the last seven days to improve search suggestions.

Task 11 – User similarity based on rating patterns (graph + aggregate)

- Find users with similar movie tastes to a given user based on their rating history to improve friend recommendations.

Task 12 – Movies by shortest path to subscribed persons (graph)

- Recommend movies to a user based on the actors/directors they subscribe to, prioritizing movies with multiple connections.

Exercises

Task 13 – Weekly retention cohort analysis (time series + aggregate)

- Track how many users from each registration week remain active in subsequent weeks to measure retention.

Task 14 – Genre affinity by user country (aggregate + group by)

- Identify which genres are most popular in each country to tailor regional content recommendations.

Task 15 – Playlist collaboration opportunities (graph)

- Identify pairs of users who have similar movie preferences in their playlists to suggest potential collaboration or sharing.

Task 16 – Trending tags in real-time (time window + aggregate)

- Identify tags that are surging in popularity in the last 24 hours compared to the previous week to highlight trending topics.

Exercises

Task 17 – User churn prediction features (aggregate)

- Calculate per-user activity metrics over rolling windows to feed into a churn prediction model.

Task 18 – Cross-genre recommendation paths (graph)

- Find movies that bridge different genres based on users who rated movies from both genres highly.

Task 19 – Review influence analysis (graph + aggregate)

- Measure how influential specific reviewers are by tracking subsequent user actions after review publication.