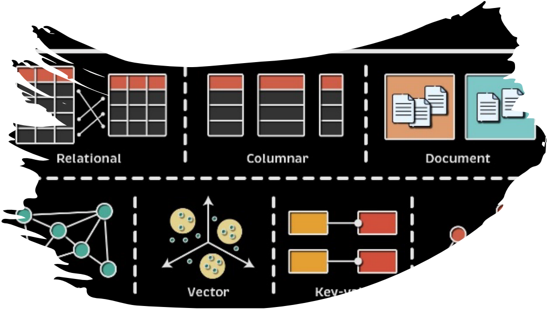


DATA MODELS

CMPU4003 Advanced Databases

Why are there different types?

Nature of applications and user demands has changed over time

- Relational databases dominated for decades
  - Data integrity and consistency valued over speed of retrieval
- Evolution of web applications, big data and real-time analytics led to alternatives being developed
  - Speed of retrieval became more highly valued
    - e.g. document, key-value stores, and graph database

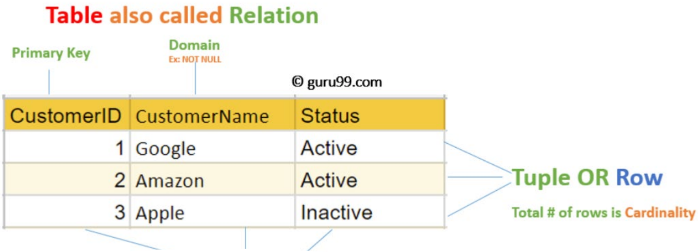
Different models

- Suit different application types
- Offer different
  - Optimization strategies
  - Scalability options
  - Flexibility options
  - Consistency and Availability options

How To Choose a Data Model

- **Nature of the data**
  - Structured, semi-structured, unstructured
- **Access patterns**
  - queries, transactions, analytics
- **Scalability and distribution needs**
- **Consistency vs availability trade-offs**
  - CAP theorem and PACELC extension
- **Integration and interoperability** with existing systems

Relational Model



Relational Data Model

- Database is a collection of relations
- Relations (Tables) are two dimensional
  - Each row represents an **entity**
  - Each column an **attribute** of that entity

Natural Relational Data

Transactional Data

Statistical Data

Basic Social Media Data

Relational Database – Rankings (https://db-engines.com/en/ranking/relational+dbms)

Rank			DBMS	Database Model	Score		
Sep 2025	Aug 2025	Sep 2024			Sep 2025	Aug 2025	Sep 2024
1.	1.	1.	Oracle	Relational, Multi-model	1170.62	-50.08	-115.97
2.	2.	2.	MySQL	Relational, Multi-model	891.77	-23.69	-137.72
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	717.32	-36.84	-90.45
4.	4.	4.	PostgreSQL	Relational, Multi-model	657.17	-14.08	+12.81
5.	5.	5.	Snowflake	Relational	190.19	+11.29	+56.47
6.	6.	6.	IBM Db2	Relational, Multi-model	124.19	-3.12	+1.14
7.	7.	↑ 9.	Databricks	Multi-model	124.06	+8.25	+39.82
8.	8.	↓ 7.	SQLite	Relational	107.88	-4.72	+4.53
9.	9.	↑ 10.	MariaDB	Relational, Multi-model	91.46	-2.13	+8.02
10.	10.	↓ 8.	Microsoft Access	Relational	83.61	-4.15	-10.15

Relational Data Model

Structure

- Schemata
  - Named, non-empty, typed, and unordered sets of attributes
  - Example: Person(ID, Surname, Name, Address)
- Instances
  - Sets of records, i.e., functions that assign values to attributes
  - Example: 12345,'Lawless',`Deirdre´,'TU Dublin, Central Quad')

Constraints

- Integrity constraints: data types, keys, foreign-keys, …

Operations

- Relational algebra (and relational calculus)
- Usually implemented as Structured Query Language (SQL)

Join at:
vevox.app

ID:
125-074-954

- Go to **vevox.app**
  - Enter the session ID: **125-074-954**
- Or
  - Scan the QR code

What do you know about the relational model?

1. Identify one strength of the relational data model?

What do you know about the relational model?

2. IDENTIFY ONE WEAKNESS OF THE RELATIONAL DATA MODEL?

Strengths of Relational Model

Mature and Standardized
- SQL is a well-established, standardized query language (ANSI SQL).
- Decades of research, vendor competition, and industry adoption
- Large ecosystem of tools, tutorials, and community support.

Proven Fit for Enterprise Systems
- Banking, finance, HR, supply chain, ERP — domains requiring accuracy and consistency rely on RDBMS.
- Good track record for mission-critical systems.

Optimized for Structured Data
- Well-suited for highly structured, tabular data with clear schema.
- Efficient storage and indexing mechanisms.
- Relational model is mathematically grounded (set theory, relational algebra).

Data Integrity and Consistency
- ACID (Atomicity, Consistency, Isolation, and Durability) Transactions
- Strong constraints (primary keys, foreign keys, unique, check)
- Referential integrity ensures data is not orphaned or inconsistent.

Expressive Querying

SQL supports complex joins, filtering, aggregation, grouping.
Optimizers rewrite queries into efficient execution plans.
Declarative style: "what you want" rather than "how to get it."

Security and Access Control
- Fine-grained access control (roles, grants, views).
- Mature audit, backup, and recovery systems.
- Built-in mechanisms for concurrency control.

Weaknesses of Relational Model

Schema Rigidity

- Schema changes are global → adding one new attribute means altering the entire table (and often app code).
- Poor fit for semi-structured or fast-evolving data (e.g., user preferences, IoT data).

Object–Relational Impedance Mismatch

- Objects in code (nested structures, pointers, lists) don't map neatly onto flat relational tables.
- Requires ORM frameworks (Hibernate, ActiveRecord), which add complexity and overhead.
- Leads to performance issues and "leaky abstractions."

Scalability and Distribution

- Designed for vertical scaling (bigger server), not horizontal (more servers).
- Harder to partition/shard relational data across clusters.
- Replication exists, but eventual consistency models (BASE) are better supported in NoSQL.

Performance Trade-offs

- Great at joins, but joins across huge datasets (billions of rows) can be slow.
- Not ideal for analytical workloads → need separate designs
- OLTP vs OLAP tension: same schema isn't good for both.

Handling Complex/Unstructured Data

- Text, JSON, XML, video, sensor logs
- Relational databases can store them (as BLOBs), but:
  - Querying/searching is inefficient.
  - Often pushed to external systems (Elasticsearch, S3, NoSQL).

Flexibility & Developer Experience

- In fast-moving projects, relational schema design can slow prototyping.
  - NoSQL/document models give developers more freedom to evolve schema alongside the app.

```
SELECT      <attribute list>
FROM        <relation list>
WHERE       <conditions>
GROUP BY    <grouping attributes>
HAVING      <grouping conditions>
ORDER BY    <attribute list>;
```

SQL

An example of a declarative query language

You specify the result of a query and not how it should be obtained:

- Easier to understand
- Transparently optimizable
- Implementation independent

Additional Keywords

DISTINCT, AS, JOIN

AND, OR

MIN, MAX, AVG, SUM, COUNT

NOT, IN, LIKE, ANY, ALL, EXISTS

UNION, EXCEPT, INTERSECT

Example

- Schema:
- Product(maker, model, type)
- PC(model, speed, ram, hd, rd)
- Laptop(model, speed, ram, hd, screen)

```
SELECT COUNT(hd)
FROM PC
GROUP BY hd
HAVING COUNT(model) > 2;
```

"How many hard disk sizes are built into more than two PCs?"

```
SELECT *
FROM PC PC1, PC PC2
WHERE PC1.speed = PC2.speed
AND PC1.ram = PC2.ram
AND PC1.model < PC2.model;
```

"Find all pairs of PCs with same speed and ram sizes."

```
 (SELECT DISTINCT maker
  FROM Product, Laptop
  WHERE Product.model = Laptop.model)
EXCEPT
 (SELECT DISTINCT maker
  FROM Product, PC
  WHERE Product.model = PC.model);
```

"Find all makers that produce Laptops but no PCs."

PostgreSQL

- Open-source, object-relational database management system (ORDBMS). Key features include:
- **ACID compliance** for reliable transactions.
- **Advanced SQL support**, including joins, subqueries, window functions, and triggers.
- **Extensibility**, allowing users to define custom data types, functions, and operators.
- **Support for JSON and XML**, enabling hybrid relational and document-based data handling.
- **Scalability and concurrency**, with strong performance for large datasets and many users.
- **Cross-platform compatibility** (Linux, macOS, Windows).

Guidelines for Choosing the Relational Model

Data Looks Structured

- The data has clear entities e.g. customers, orders, products.
- The relationships are predictable and repeat across records.
  - A relational model makes sense because tables and foreign keys map naturally to this structure.

Schema Isn't Changing

- Looking at the data, the columns and attributes are unlikely to change — e.g. always need name, price, quantity, etc.
  - A fixed relational schema makes sense when the data structure is stable.

Need Integrity and Validation

- I need to ensure data is valid.
  - e.g. product IDs always need to match existing products, and orders must link to customers.
  - Relational databases enforce these constraints automatically, so I don't have to handle them all in application code.

Consistency Is Critical

- If two customers check out at the same time, I can't risk selling the same item twice.
  - ACID transactions guarantee that either the full order is stored or none of it is, ensuring correctness.

Guidelines for Choosing the Relational Model

Need to Run Complex Queries

- The business wants large scale reports like "total sales per customer by region, broken down by product category." involving large numbers of tables with analytics.
  - SQL in a relational database can handle multi-table joins, grouping, and aggregation efficiently.

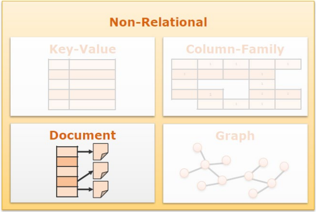
Need Mature Tooling

- Need tools like dashboards, ORMs, and BI tools to work out of the box.
  - Relational databases integrate smoothly with existing developer tools, making them less effort to maintain.

Compliance and Auditing Are Required

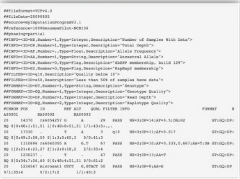
- The data looks sensitive —e.g. financial or healthcare-related.
  - Relational databases offer strong support for auditing, logging, and traceability.

Non-Relational Model



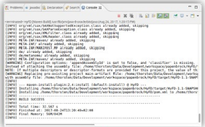
Document Data model
Natural Document Data


Digital Documents


Scientific Data Formats


Web Pages


Structured Data


Log Data

Document Oriented Stores (https://db-engines.com/en/ranking/document+store)

Rank			DBMS	Database Model	Score		
Sep 2025	Aug 2025	Sep 2024			Sep 2025	Aug 2025	Sep 2024
1.	1.	1.	MongoDB	Document, Multi-model	380.50	-15.08	-29.74
2.	2.	2.	Databricks	Multi-model	124.06	+8.25	+39.82
3.	3.	3.	Amazon DynamoDB	Multi-model	80.28	-3.20	+10.22
4.	4.	4.	Microsoft Azure Cosmos DB	Multi-model	23.94	+1.10	-1.03
5.	5.	↑6.	Firebase Realtime Database	Document	15.40	+1.01	+1.80
6.	6.	↓5.	Couchbase	Multi-model	12.58	-0.09	-4.16
7.	7.	↑9.	Google Cloud Firestore	Document	9.18	+0.72	+2.55
8.	↑9.	8.	Realm	Document	6.82	+0.39	-0.36
9.	↓8.	↓7.	CouchDB	Document, Multi-model	6.55	-0.26	-0.91
10.	10.	10.	Aerospike	Multi-model	5.10	+0.27	-0.06

Document Data Model

Structure

- Hash map: (large, distributed) key-value data structure
  - Values are documents or collections of documents that (usually) contain hierarchical data
- XML, JSON, RDF, HTML, …

Constraints

- Each value/document is associated with a unique key

Operations

- Store key-value pair
- Retrieve value by key
- Remove key-value mapping

Note:

- Document stores are often considered to be schemaless, but since the applications usually assume some kind of structure they are rather **schema-on-read** in contrast to **schema-on-write**.

Document Data Model

Relational data model

Highly-structured table organization with rigidly-defined data formats and record structure.

Document data model

Collection of complex documents with arbitrary, nested data formats and varying "record" format.

JSON

- **J**ava**S**cript **O**bject **N**otation
- Lightweight data interchange format
- Text format
- Semi-structured data

```
{ "studentDetails": {
      "name" : "Joe",
      "age" : 16,
      "dept" : "computers",
      "hobbies" : ["dance", "books", "public speaking",
"golf"],
      "isClassLeader" : false
      }
}
```

JSON

- JSON objects are written within {curly} braces.
- Each item is a key-value pair.
- The keys and string type values are written within double quotes.
  - Other data types—like Integer and Boolean—don't need to be written in quotes.
- Each item is separated from the next one using a comma (,). There is no comma after the last item.
- Arrays inside JSON strings are written within [square] brackets.
- Objects and arrays can be embedded within an object

```
{ "studentDetails": {
       "name" : "Joe",
       "age" : 16,
       "dept" : "computers",
       "hobbies" : ["dance", "books", "public speaking", "golf"],
       "isClassLeader" : false
       }
}
```

BSON

- BSON stands for Binary JSON.
  - It’s a binary-encoded serialization format that extends JSON with additional data types and faster encoding/decoding.
- Binary format
  - More compact and faster to parse than plain text JSON.
  - Rich data types
    - Supports everything JSON does (strings, numbers, arrays, objects) plus extra types like:int32 and int64 (different integer sizes), double (floating point), Boolean, date and timestamp, binary data (raw bytes, good for images/files)
    - ObjectId (unique document identifiers in MongoDB)null and regex
  - Traversable → Designed for fast in-memory traversal, which helps databases like MongoDB efficiently query nested fields.
  - JSON

```
{
  "name": "Joe",
  "age": 16,
  "hobbies": ["dance", "books"]
}
```

  - BSON

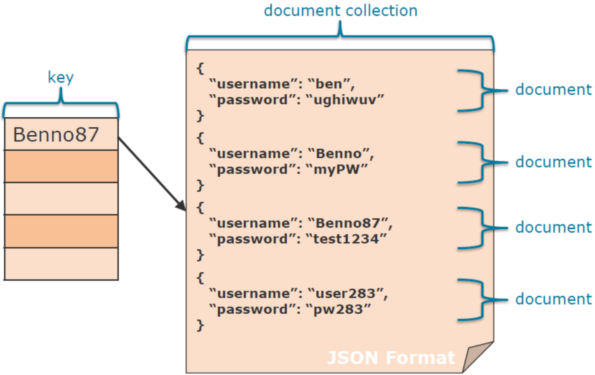
```
\x16\x00\x00\x00              // total document size
\x02 name \x00 Joe\x00        // string
\x10 age \x00 16              // 32-bit integer
\x04 hobbies \x00 ...         // array
\x00                          // end of document
```

JSON Schema (can but don’t have to use)

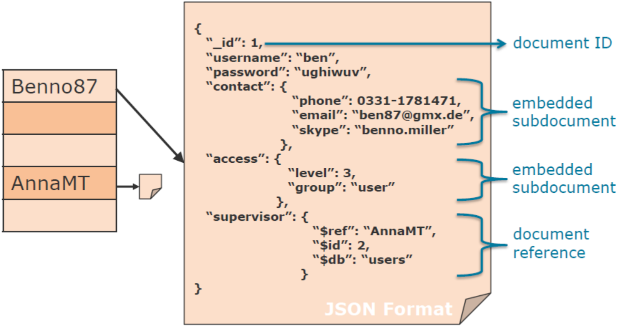
schema

```json
{
  "$id": "https://example.com/person.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Person",
  "type": "object",
  "properties": {
    "firstName": {
      "type": "string",
      "description": "The person's first name."
    },
    "lastName": {
      "type": "string",
      "description": "The person's last name."
    },
    "age": {
      "description": "Age in years which must be equal to or greater than zero.",
      "type": "integer",
      "minimum": 0
    }
  }
}
```

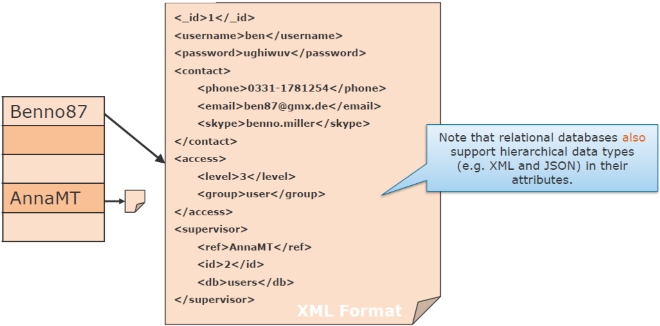
Document Data Model Example



Document Data Model Example



Document Data Model Example



CouchDB

```json
{
  "_id": "student_joe_001",
  "studentDetails": {
    "name": "Joe",
    "age": 16,
    "dept": "computers",
    "hobbies": [
      "dance",
      "books",
      "public speaking",
      "golf"
    ],
    "isClassLeader": false
  }
}
```

- _id: Required field in CouchDB.
  - You can provide your own meaningful string (e.g., "student_joe_001")
  - Or let CouchDB auto-generate one (if you omit it).
- _rev: Will be added automatically by CouchDB after the first save and updated on every modification.

Join at:
vevox.app

ID:
125-074-954

- Go to **vevox.app**
  - Enter the session ID: **125-074-954**
- Or
  - Scan the QR code

What do you know about the document model?

3. Identify one strength of the document data model?

What do you know about the document model?

4. IDENTIFY ONE WEAKNESS OF THE DOCUMENT DATA MODEL?

Strengths of Document model

Flexible Schema

- Can handle records with different fields without needing a fixed schema.
- New attributes can be added without database migrations.

Natural Representation

- JSON/BSON documents map neatly to objects in code (less ORM overhead).
- Hierarchical/nested data fits well (e.g., blog post with comments, product with variations).

Efficient for Whole-Object Access

- Fetching a full record is fast — no joins needed for nested data.
- Ideal when the application usually needs the whole document.

High Scalability and Distribution

- Designed to scale horizontally via partitioning (sharding).
- Can be easily replicated for fault tolerance and load balancing.

Good Fit for Web and APIs

- Works natively with JSON, which is common in REST and GraphQL APIs.

High Insert and Read Performance

- Writes are efficient since documents are stored as blobs.
- Reads are fast when fetching by ID or simple query.

Weaknesses of Document model

Poor at Complex Relationships

- No (or very limited) joins across collections.
- If relationships exist (e.g., users → orders → products), you may duplicate data or handle joins in the

Aggregation Limitations

- Some aggregations and analytics are harder or less efficient than in relational DBs.
- Developers often need to use pipelines or external processing.

Update Costs

- Updating a large document may require rewriting the whole object.
- If document size changes a lot, storage fragmentation can occur.

Inconsistent Schema Enforcement

- Flexibility can become a problem if documents drift apart in structure.
- Application code must enforce consistency.

Distribution Requires Planning

- Developers must carefully design partition keys/shards.
- Poor choices can cause hotspots and uneven load.

Indexing Trade-offs

- Indexes improve performance but come at high storage and update costs.
- Multi-field indexes are less flexible than SQL query optimisers.

Guidelines for Choosing the Document Model

Data Looks Semi-Structured

- A rigid schema doesn't fit — a flexible document model (JSON) is more natural.
- E.g. looking at product catalogs where each item has different attributes- books have authors, electronics have warranty info, clothing has sizes.

Schema Changes Frequently

- e.g. today, need to store deliveryInstructions; tomorrow might need giftWrapOption.
- With documents, can add new fields without changing the whole database schema.

Records Are Hierarchical or Nested

e.g. a blog post has comments, tags, likes, and embedded user details.

- All this can live inside a single JSON document instead of spreading across multiple tables.

Data Will Be Accessed as Whole Objects

e.g. the app often fetches an entire user profile or order with all details at once.

- A single document read is faster than joining multiple relational tables.

Guidelines for Choosing the Document Model

Easy Distribution Required

Expect high read/write throughput globally.

- Document databases naturally partition (shard) across servers using document IDs.

Queries Are Mostly Lookup and Aggregation

e.g. “Show me all orders from this user in 2023,” or “calculate total sales by region.”

- Many document DBs (like MongoDB) now have built-in aggregation frameworks.

JSON Is the Native Format

The app is already using JSON in APIs.

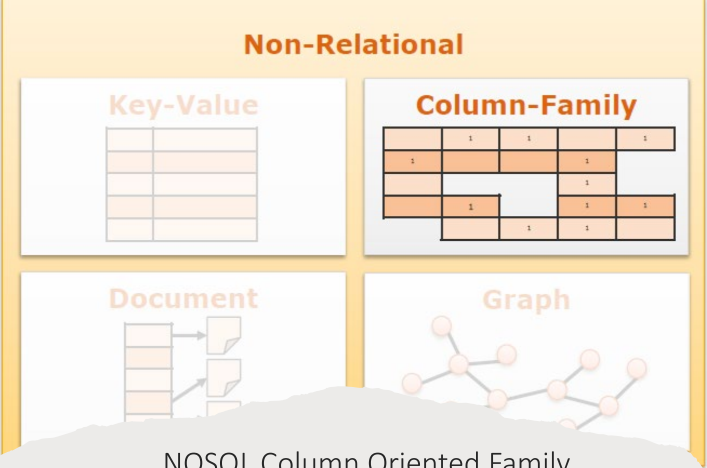
- No need to map objects into relational tables — just store the JSON directly.

JSON and JSONB in PostgreSQL

- PostgreSQL introduced the JSON data type with Postgres 9.2.
  - Allowed Postgres to start becoming a direct competitor to NoSQL technologies.
  - JSON data is not much more than a simple text field.
  - Can perform some basic JSON operations, such as extracting the value associated with an object key.
    - These operations are rather slow and are not optimized for large JSON data.

JSON and JSONB in PostgreSQL

- PostgreSQL added the JSONB data type in Postgres 9.4
  - The 'b' at the end of the data type name stands for 'better'.
  - Jsonb stores JSON data in a special binary representation, whose format is compressed and more efficient than text
- JSONB is based on an optimized format that supports many new operations.
- Extracting the value associated with an object key is very fast.
- Jsonb also allows you to:
  - Set a new key
  - Update the value of an existing key
  - Set a value in a nested object
  - Update the value of a nested key
  - Delete a key
  - Delete a nested key
  - Concatenate JSON objects
  - Deal with JSON array



NOSQL Column Oriented Family

Column-Oriented Family (https://db-engines.com/en/ranking/wide+column+store)

Rank			DBMS	Database Model	Score		
Sep 2025	Aug 2025	Sep 2024			Sep 2025	Aug 2025	Sep 2024
1.	1.	1.	Apache Cassandra	Wide column, Multi-model	106.98	-1.53	+8.04
2.	2.	↑ 3.	Microsoft Azure Cosmos DB	Multi-model	23.94	+1.10	-1.03
3.	3.	↓ 2.	Apache HBase	Wide column	21.39	-0.69	-6.02
4.	4.	↑ 5.	ScyllaDB	Wide column, Multi-model	3.99	+0.29	-0.17
5.	5.	↓ 4.	Datastax Enterprise	Wide column, Multi-model	3.68	+0.36	-1.15
6.	6.	6.	Microsoft Azure Table Storage	Wide column	2.84	+0.14	-0.71
7.	7.	↑ 8.	Google Cloud Bigtable	Multi-model	2.75	+0.15	-0.22
8.	8.	↓ 7.	Apache Accumulo	Wide column	2.55	+0.27	-0.68
9.	9.	9.	Amazon Keyspaces	Wide column	1.23	+0.09	+0.18
10.	10.	10.	HPE Ezmeral Data Fabric	Multi-model	0.82	-0.03	-0.06

Column-Oriented Family Data Model

Structure

- Multi-dimensional hash map
  - (large, distributed) key-value data structure that uses a hierarchy of up to three keys for one typed value
- Conceptually equivalent to sparse relational tables, i.e., each row supports arbitrary subsets of attributes

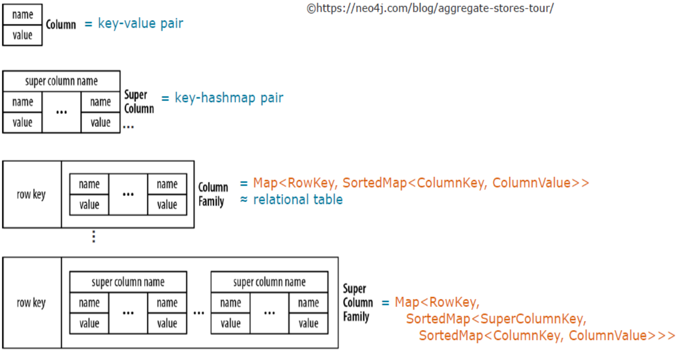
Constraints

- Each value is associated with a unique key
- Hierarchy of keys is a tree
- Integrity constraints: keys, foreign-keys, cluster-keys (for distribution), …

Operations

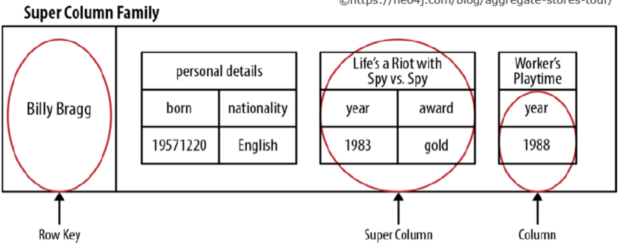
- At least: store key-value pair; retrieve value by key; remove key-value pair
- Usually: relational algebra support without joins (with own SQL dialect)

Column-Oriented Family Example



Column-Oriented Family Example

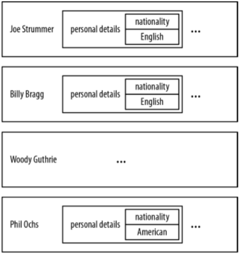
©https://neo4j.com/blog/aggregate-stores-tour/



Hierarchy of keys enables:

- Flexible schemata (column names model attributes and row keys records)
- Value groupings (by super column names and row keys)

Column-Oriented Family Example



Analogy:

Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

Hierarchy of keys enables:

- Flexible schemata (column names model attributes and row keys records)
- Value groupings (by super column names and row keys)

Column-Oriented Family Example

- Cassandra Query Language CQL …is an SQL dialect (same syntax)
  - https://cassandra.apache.org/_/index.html
  - Supports all DML and DDL functionalities
  - Does not support:
    - joins, group by, triggers, cursors, transactions, or (stored) procedures
    - OR and NOT logical operators (only AND)
  - Subqueries
- With the following key differences:
  - WHERE conditions should be applied only on columns with an index
  - Timestamps are comparable only with the equal operator (not <,>,<>)
  - UPDATE statements only work with a primary key (they do not work based on other columns or as mass update)
  - INSERT can override existing records, UPDATE can creates new records

CQL Example

Schema:

first key attribute(s) = **partition key** (determines which node stores the data)

- `Playlists(id, song_order, album, artist, song_id, title)`

further key attribute(s) = **cluster key** (keys within a partition/node)

Query:

```
SELECT *
FROM Playlists
WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
ORDER BY song_order DESC
LIMIT 4;
```

= key attribute

= key attribute

Result:

id	song_order	album	artist	song_id	title
62c36092...	4	No One Rides for Free	Fu Manchu	7db1a490...	Ojo Rojo
62c36092...	3	Roll Away	Back Door Slam	2b09185b...	Outside Woman Blues
62c36092...	2	We Must Obey	Fu Manchu	8a172618...	Moving in Stereo
62c36092...	1	Tres Hombres	ZZ Top	a3e63f8f...	La Grange

CQL Example

SQL:

```
CREATE DATABASE myDatabase;
```

```
SELECT *
FROM myTable
WHERE myField > 5000
AND myField < 100000;
```

CQL:

```
CREATE KEYSPACE myDatabase
WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': 1};
```

```
SELECT *
FROM myTable
WHERE myField > 5000
AND myField < 100000
ALLOW FILTERING;
```

Otherwise:

Bad Request: Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute it despite the performance unpredictability, use ALLOW FILTERING.

Strengths of Column-Oriented model

- Efficient storage: fast inserts of data items
- Efficient retrieval: fast point queries, i.e., value look-ups
- Data structure is easy to distribute across multiple machines
- Data structure can be replicated for fault-tolerance and load balancing
- Flexible schemas

Weaknesses of Column-Oriented model

No join and limited filtering support (filtering might also be super slow)

- Must be done by the application (or distributed computing framework)

Multi-key structure groups values to entities but general groupings and aggregations are not supported

Non-point queries, i.e., those that read more than one mapping, are costly

Join at:
vevox.app

ID:
125-074-954

- Go to **vevox.app**
    - Enter the session ID: **125-074-954**
- Or
    - Scan the QR code

5. *If you had to build a* ***Netflix-style recommendation system,*** *would you choose a relational model?*

Discuss in small groups. Then answer Yes/No and give a reason

*If you had to build a **Netflix-style recommendation system**, would you choose a relational model?*

- **Yes, Relational Could Work (but with trade-offs)**
  - Relational databases are mature and can model users, movies, and ratings in normalized tables.
  - SQL makes it easy to join and aggregate
    - e.g., “top movies by user’s demographic.”
  - Strong integrity ensures valid data.
  - **BUT**:
    - Scaling to millions of users and billions of interactions would be challenging.
    - Horizontal scaling is hard.
    - Queries could become slow.

If you had to build a ***Netflix-style recommendation system****, would you choose a relational model?*

User–Movie Ratings Matrix

User / Movie	Movie A	Movie B
User 1	5	0
User 2	0	0
User 3	0	4
User 4	2	0

- **Total entries** = 20
- **Non-zero ratings** = 5
- **Most entries are 0 (unrated movies)** → This is a spa

- **No, Relational Isn't the Best Fit**
  - Recommendation systems often deal with **huge, sparse matrices** of user-item interactions.
    - Rows = Users
    - Columns = Movies
    - Values = Rating (or interaction, e.g., 1 if watched, 0 if not)
    - Sparse: Users only rate or interact with a few items compared to the full catalog.
    - If Netflix has 10 million users and 50,000 movies, the full matrix would have 500 billion entries.
    - Each user only watches/rates a tiny fraction → more than 99.9% are zeros.

  - Relational systems struggle with high write throughput and distributed scalability.

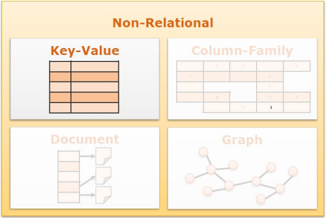
*If you had to build a **Netflix-style recommendation system**, would you choose a relational model?*

- **Maybe, As Part of a Hybrid Approach**
  - Relational DB could store **core reference data** (movies, users, metadata).
  - A NoSQL system (wide-column, key-value, or graph) could handle the **recommendation engine**.
  - Many real-world companies (Netflix, Spotify) use **polyglot persistence**.

Polyglot persistence

- **Data management approach** where an organization uses different types of databases, each chosen because it is best suited for a particular component of an application or a specific type of data.
- Instead of trying to fit all data into a single database system (like only using relational databases), polyglot persistence embraces diversity in database technologies.
- **Why it exists**
  - Different databases excel at different tasks:
  - **Relational databases (SQL)** → great for structured data, consistency, transactions.
  - **Document stores (e.g., MongoDB)** → flexible schemas, good for JSON-like data.
  - **Key-value stores (e.g., Redis)** → ultra-fast lookups and caching.
  - **Graph databases (e.g., Neo4j)** → efficient for handling relationships and network data.
- No single database technology is optimal for *every* workload.
- Polyglot persistence allows systems to leverage the strengths of multiple databases within one architecture.

Non-Relational Key-Value



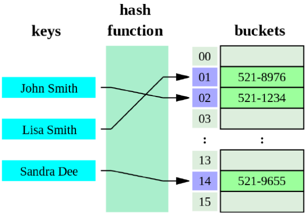
Key-Value Stores– Rankings (https://db-engines.com/en/ranking/key-value+store)

Rank			DBMS	Database Model	Score		
Sep 2025	Aug 2025	Sep 2024			Sep 2025	Aug 2025	Sep 2024
1.	1.	1.	Redis	Key-value, Multi-model	145.17	-2.02	-4.25
2.	2.	2.	Amazon DynamoDB	Multi-model	80.28	-3.20	+10.22
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	23.94	+1.10	-1.03
4.	4.	4.	Memcached	Key-value	16.16	+0.01	-0.68
5.	5.	5.	etcd	Key-value	7.44	+0.56	+0.40
6.	↑ 7.	6.	Hazelcast	Key-value, Multi-model	5.26	+0.56	-0.46
7.	↓ 6.	7.	Aerospike	Multi-model	5.10	+0.27	-0.06
8.	8.	↑ 11.	Oracle NoSQL	Multi-model	3.62	+0.16	+0.55
9.	↑ 10.	↓ 8.	Ehcache	Key-value	3.48	+0.24	-1.31

Key-Value Data Model

- Structure
  - Hash map: (mostly large, distributed) key-value data structure
- Constraints
  - Each value is associated with a unique key
- Operations
  - Store key-value pair
  - Retrieve value by key
  - Remove key-value mapping

Example



©Jorge Stolfi (https://commons.wikimedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg)

Querying – Redis

- In-memory key-value store with file persistence on disk
- Supports five data structures for values:
    - Strings: byte arrays that may represent actual strings or integers, binary serialized objects, …
    - Hashes: dictionaries that map secondary keys to strings
    - Lists: sequences of strings that support insert, append, pop, push, trim, and many further operations
    - Sets: duplicate free collections of strings that support set operations such as diff, union, intersect, …
    - Ordered sets: duplicate free, sorted collections of strings that use explicitly defined scores for sorting and support range operations

Querying – Redis API

- Strings:
  SET hello "hello world"
  GET hello
  → "hello world"
  SET users:goku {race: 'sayan', power: 9001}
  GET users:goku
  → {race: 'sayan', power: 9001}
- Hashes:
  HSET users:goku race 'sayan'
  HSET users:goku power 9001
  HGET users:goku power
  → 9001

"<group>:<entity>" is a naming convention.

- Lists:
  LPUSH mylist a // [a]
  LPUSH mylist b // [b,a]
  RPUSH mylist c // [b,a,c]
  LRANGE mylist 0 1
  → b, a
  RPOP mylist
  → c
- Sets:
  SADD friends:lisa paul
  SADD friends:lisa duncan
  SADD friends:paul duncan
  SADD friends:paul gurney
  SINTER friends:lisa friends:paul
  → duncan
- Ordered sets:
  ZADD lisa 8 paul
  ZADD lisa 7 duncan
  ZADD lisa 2 faradin
  ZRANGEBYSCORE lisa 5 8
  → duncan
  → paul

Strengths of Key-Value Model

- Efficient storage: fast inserts of key-value pairs
- Efficient retrieval: fast point queries, i.e., value look-ups
- Key-value pairs are easy to distribute across multiple machines
- Key-value pairs can be replicated for fault-tolerance and load balancing

Weaknesses of Key-Value Model

- No filtering, aggregation, or joining of values/entries
- Must be done by the application (or distributed computing framework)

Overall Rankings

Rank			DBMS	Database Model	Score		
Sep 2025	Aug 2025	Sep 2024			Sep 2025	Aug 2025	Sep 2024
1.	1.	1.	Oracle	Relational, Multi-model	1170.62	-50.08	-115.97
2.	2.	2.	MySQL	Relational, Multi-model	891.77	-23.69	-137.72
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	717.32	-36.84	-90.45
4.	4.	4.	PostgreSQL	Relational, Multi-model	657.17	-14.08	+12.81
5.	5.	5.	MongoDB	Document, Multi-model	380.50	-15.08	-29.74
6.	6.	↑7.	Snowflake	Relational	190.19	+11.29	+56.47
7.	7.	↓6.	Redis	Key-value, Multi-model	145.17	-2.02	-4.25
8.	8.	↑9.	IBM Db2	Relational, Multi-model	124.19	-3.12	+1.14
9.	9.	↑14.	Databricks	Multi-model	124.06	+8.25	+39.82
10.	10.	↓8.	Elasticsearch	Multi-model	118.26	+3.99	-10.53
11.	11.	↓10.	SQLite	Relational	107.88	-4.72	+4.53
12.	12.	↓11.	Apache Cassandra	Wide column, Multi-model	106.98	-1.53	+8.04
13.	13.	↑15.	MariaDB	Relational, Multi-model	91.46	-2.13	+8.02
14.	14.	↓12.	Microsoft Access	Relational	83.61	-4.15	-10.15
15.	15.	↑17.	Amazon DynamoDB	Multi-model	80.28	-3.20	+10.22
16.	16.	16.	Microsoft Azure SQL Database	Relational, Multi-model	79.18	+3.34	+6.23
17.	17.	↑18.	Apache Hive	Relational	76.10	+5.06	+23.02
18.	18.	↓13.	Splunk	Search engine	75.77	+6.00	-17.26
19.	19.	19.	Google BigQuery	Relational	66.00	+0.82	+13.33
20.	20.	↑21.	Neo4j	Graph	53.78	-0.69	+11.10

https://db-engines.com/en/ranking