Data, Data Storage, Data Collection

Lecture 7: Data Storage

Romain Pascual

MICS, CentraleSupélec, Université Paris-Saclay

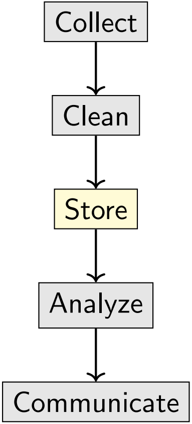
Recap

Recap: Data Storage

1. Understand the difference between storage **models** (flat, tabular, hierarchical, columnar) and **formats** (CSV, JSON, Parquet)
2. Choose the right format based on your use case: Excel for business, CSV/JSON for sharing, Parquet for analytics
3. Consider the trade-offs between readability, size, speed, and interoperability when selecting a format
4. For long-term storage, prioritize open standards and self-describing formats with proper metadata
5. Document your storage decisions and maintain data provenance for reproducibility
6. Consider ethical implications including privacy, accessibility, and sustainability

Introduction

Within the lifecycle



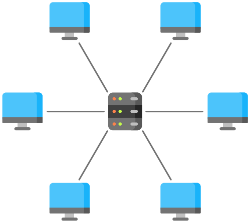
Session Objectives

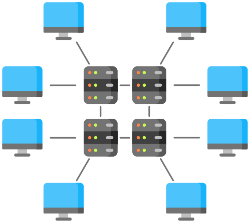
At the end of this session, you should be able to:

- Explain the differences between SQL and NoSQL databases.
- Describe the four families of NoSQL databases and their use cases.
- Perform basic CRUD operations in MongoDB.

Why NoSQL databases?

NoSQL for *Not only* SQL





Goals

- Provides better scalability for distributed databases
- Provides better support for semi-structured databases
- (Addresses the object-relational impedance mismatch: store data as objects, like in code, instead of tables)

Aggregate

Aggregate: The Basic Unit of NoSQL

Unlike relational databases, we can control how data is distributed. The atomic blocks for distribution are called aggregates.

Definition (Aggregate)

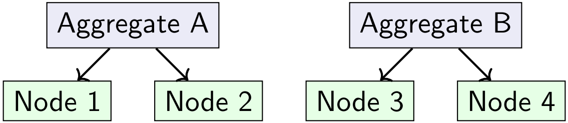
A **collection of data** that interacts as a single unit.

```json
{
  "user_id": 123,
  "name": "Alice",
  "balance": 1000,
  "orders": [
    {"order_id": 1, "amount": 99.99},
    {"order_id": 2, "amount": 49.99}
  ]
}
```

Aggregates for Distribution and Operations

Aggregates are the **atomic blocks** for:

- **Replication**: Copying data across nodes for redundancy.
- **Sharding**: Splitting data across nodes for scalability.
- **Operations**: CRUD operations access the complete unit.



Comparison with Relational Databases

- $\sim$ a row in relational databases: store a specific entity
- $\neq$ a row in relational databases: can nest other aggregates

```json
{
  "user_id": 123,
  "name": "Alice",
  "balance": 1000,
  "orders": [
    {"order_id": 1, "amount": 99.99},
    {"order_id": 2, "amount": 49.99}
  ]
}
```

Distributing a Relational Database

users

user_id	name
123	Alice
456	Bob

orders

order_id	user_id	amount
1	123	99.99
2	123	49.99
3	456	29.99

Distributing a Relational Database

users

user_id	name
123	Alice
456	Bob

Node 1

orders

order_id	user_id	amount
1	123	99.99
2	123	49.99
3	456	29.99

Node 2

Issue

- Data for an entity is split across nodes, e.g., Alice + her orders
- Queries require `JOIN`s, e.g., "Get all orders for Alice"

Denormalize the Table

users

user_id	name
123	Alice

orders

order_id	user_id	amount
1	123	99.99
2	123	49.99

user_id	name	amount
123	Alice	[99.99, 49.99]

Idea

- Simulate the JOINs such that all data for an entity is stored together (data co-location)

Example: Get All Orders for Alice

users

user_id	name
123	Alice
456	Bob

orders

order_id	user_id	amount
1	123	99.99
2	123	49.99
3	456	29.99

```json
{
  "user_id": 123,
  "name": "Alice",
  "balance": 1000,
  "orders": [
    {"order_id": 1, "amount": 99.99},
    {"order_id": 2, "amount": 49.99}
  ]
}
```

- SQL: Requires a `JOIN` between tables.
- NoSQL: Simply access the aggregate.

Denormalization in Aggregates

Why It Matters

- **Performance**: Avoids complex `JOIN` operations, which can be slow in distributed systems.
- **Simplicity**: Simplifies queries by keeping related data together.
- **Flexibility**: Allows for flexible schema design, accommodating changes more easily.

Trade-offs

- **Redundancy**: Denormalization can lead to data redundancy, which may increase storage requirements.
- **Consistency**: Ensuring data consistency can be more challenging with denormalized data.

Aggregates are Schemaless

Aggregates might not have the same attributes.

```json
{
  "user_id": 123,
  "name": "Alice",
  "balance": 1000,
  "orders": [
    {"order_id": 1, "amount": 99.99},
    {"order_id": 2, "amount": 49.99}
  ]
}
```

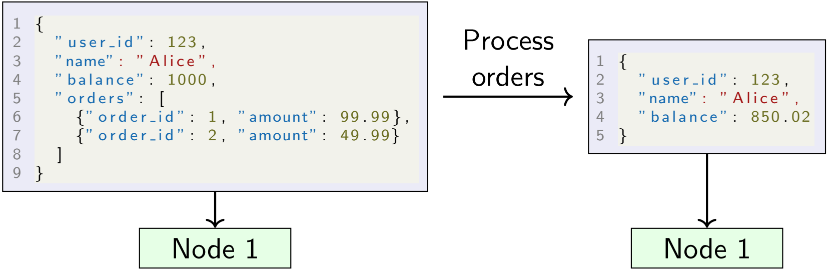
```json
{
  "user_id": 456,
  "name": "Bob",
  "balance": 0
}
```

- No need to fix a rigid schema.
- NULL values are avoided.

Atomic Operations on Aggregates

Operations on a single aggregate are **atomic**:

- The entire operation is completed as a single unit.
- No partial updates



- Ensures data consistency within an aggregate.
- Simplifies error handling and recovery.

Cross-Aggregate Operations

```
{
  "user_id": 123,
  "name": "Alice",
  "balance": 1000,
  "orders": [
    {"order_id": 1, "amount": 99.99},
    {"order_id": 2, "amount": 49.99}
  ]
}
```

Node 1

```
{
    "order_id": 1,
    "user_id": 123,
    "amount": 99.99,
    "status": "pending",
    "items": [
        {"item_id": 101,
         "name": "Laptop",
         "price": 999.99,
         "quantity": 1}
    ]
}
```

Node 1

Cross-Aggregate Operations

```json
{
  "user_id": 123,
  "name": "Alice",
  "balance": 1000,
  "orders": [
    {"order_id": 1, "amount": 99.99},
    {"order_id": 2, "amount": 49.99}
  ]
}
```

Node 1

```json
{
    "order_id": 1,
    "user_id": 123,
    "amount": 99.99,
    "status": "pending",
    "items": [
        {"item_id": 101,
        "name": "Laptop",
        "price": 999.99,
        "quantity": 1}
    ]
}
```

Node 2

Cross-aggregate operations can involve several nodes:

- No guarantee that related aggregates are stored on the same node.
- Network latency and partition might cause inconsistencies.

Aggregate-Based Databases

Schemaless

- No rigid schema
- Flexible evolution

Denormalization

- Fast queries
- Redundancy (cheap storage)

Distribution

- Aggregate in a single node
- Control

Aggregate Update

- Atomicity (all or nothing)

Cross-Aggregate

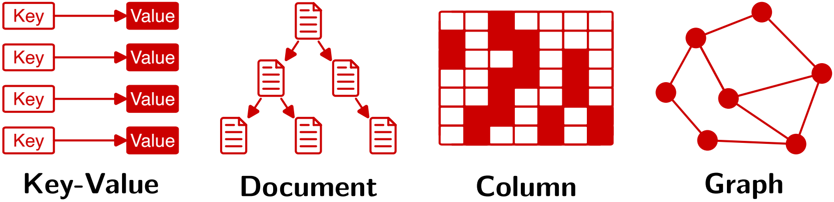
- No atomicity
- Availability vs. consistency

Design

- Optimize for query patterns
- Eventual consistency

NoSQL Databases

Four families of NoSQL

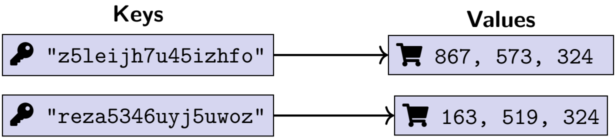


- NoSQL is not a single database or query language but a diverse family of technologies.
- The first three NoSQL families use aggregates, while graph databases were designed ACID-compliant.

Key-Value Databases

Description

- **Key**: Usually (auto-generated) an alphanumeric string.
- **Value**: An aggregate.
- Distribution based on the key (little to no integrity constraint)
- Optimized for speed (read/write queries) and scalability.



Examples

Amazon DynamoDB, Redis.

Document-Oriented Databases

Description

- Data is modeled as nested key-value pairs.
- Supports searching aggregates based on their attribute values.
- Values are typically JSON, or XML documents.

```json
{
  "user_id": 123,
  "name": "Alice",
  "balance": 1000,
  "orders": [
    {"order_id": 1, "amount": 99.99},
    {"order_id": 2, "amount": 49.99}
  ]
}
```

Examples

MongoDB, CouchDB.

Column-Oriented Databases

Description

- Data is stored in columns rather than rows.
- Columns (or column families) as aggregates.
- Optimized for analytical queries on large datasets.

Name	Age	City
Alice	28	Paris
Bob	32	London
Charlie	25	Berlin

Row Storage	
Row 1	Alice
	28
	Paris
Row 2	Bob
	32
	London
Row 3	Charlie
	25
	Berlin

Row-based access
reads entire row

Column Storage	
Name	Alice
	Bob
	Charlie
Age	28
	32
	25
City	Paris
	London
	Berlin

Column-based access
reads only needed columns

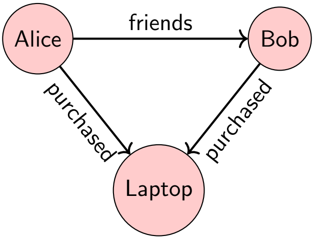
Examples

Apache Cassandra, HBase.

Graph Databases

Description

- Relationships are first-class citizens.
- Nodes, edges, and properties are the core components.
- Optimized for traversing relationships.

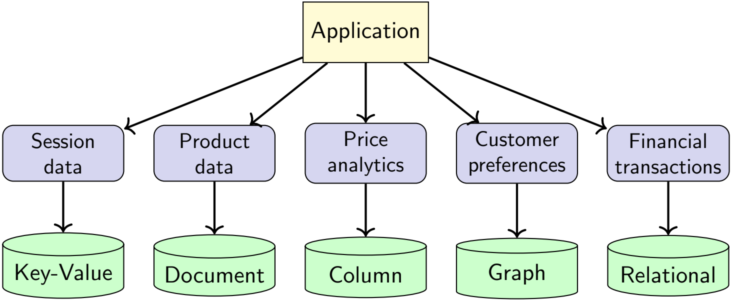


Examples

Neo4j

Polyglot Persistence

- NoSQL databases complement rather than replace relational databases.
- Use different data storage technologies based on data type and usage.



MongoDB

MongoDB: Generalities

General Characteristics

- Document-oriented NoSQL database (JSON-like documents)
- Client-server architecture
- Open-source (Community Edition) and commercial (Enterprise Edition)
- Cloud-based solution: MongoDB Atlas

Key Advantages

- Flexible schema for evolving data requirements
- Horizontal scalability with sharding
- Rich query language and aggregation framework

Data Organization and Storage Format

Data Structure

- **Document**: nested dictionaries
- **Collection**: set of documents
- **Database**: set of collections

BSON (Binary JSON)

- Faster parsing and processing
- More space-efficient
- Supports Data types

Every document must have an `_id` field:

- Auto-generated as `ObjectId` if not specified

```
{
  "_id": ObjectId("507f1f77bcf86cd799439011"),
  "title": "The Shawshank Redemption",
  "year": 1994,
  "directors": ["Frank Darabont"],
  "actors": ["Tim Robbins", "Morgan Freeman"],
  "rating": 9.3,
  "awards": {"nominations": 7, "wins": 0}
}
```

Seting Up MongoDB

Local Installation: MongoDB Community Server

Server :

- Start server with: `mongod`
- Default port: 27017
- Can manage multiple databases

Client Options

- CLI: `mongosh`
- GUIs: Compass, Atlas UI
- Drivers for all major languages

Cloud Solution: MongoDB Atlas

- Fully-managed cloud database
- Connection through an API
- Web-based GUI for database management

Connecting to a MongoDB Database

Use the MongoDB shell to connect:

- mongosh: connect to the server (localhost:27017 by default)
- use <database>: switch to a specific database
- db.createCollection(<collection>): create a collection in the database

```
$ mongosh
```

Connecting to a MongoDB Database

Use the MongoDB shell to connect:

- `mongosh`: connect to the server (localhost:27017 by default)
- `use <database>`: switch to a specific database
- `db.createCollection(<collection>)`: create a collection in the database

```
$ mongosh
> use moviedb
switched to db moviedb
```

Connecting to a MongoDB Database

Use the MongoDB shell to connect:

- mongosh: connect to the server (localhost:27017 by default)
- use <database>: switch to a specific database
- db.createCollection(<collection>): create a collection in the database

```
$ mongosh
> use moviedb
switched to db moviedb
> db.createCollection("movies")
{ ok: 1 }
```

Connecting to a MongoDB Database

Use the MongoDB shell to connect:

- `mongosh`: connect to the server (localhost:27017 by default)
- `use <database>`: switch to a specific database
- `db.createCollection(<collection>)`: create a collection in the database

```
$ mongosh
> use moviedb
switched to db moviedb
> db.createCollection("movies")
{ ok: 1 }
> show collections
movies
```

Connecting to a MongoDB Database

Use the MongoDB shell to connect:

- `mongosh`: connect to the server (localhost:27017 by default)
- `use <database>`: switch to a specific database
- `db.createCollection(<collection>)`: create a collection in the database

```
$ mongosh
> use moviedb
switched to db moviedb
> db.createCollection("movies")
{ ok: 1 }
> show collections
movies
```

⚠ In the lab, we will manage these steps by accessing the API via Python

Manipulating Data with MongoDB

MongoDB provides a set of functions to apply CRUD operations on the data.

CRUD: **C**reate, **U**pdate, **R**ead, **D**elete

Format of a CRUD function in MongoDB

```
db.collection.function()
```

- `db`: current database
- `collection`: collection where the CRUD operation is applied.
- `function`: invoked function.

Create

Insert Operations in MongoDB

- `insertOne`: adds a new document to a collection
- `insertMany`: adds multiple documents to a collection

```
db.movies.insertOne({
  "title": "Inception",
  "year": 2010,
  "genres": ["Action", "Sci-Fi", "Thriller"],
  "director": "Christopher Nolan",
  "actors": ["Leonardo DiCaprio", "Ken Watanabe"],
  "rating": 8.8
})
```

⚠ If the collection does not exist, any insert operation will create it.

Read

Query Operations in MongoDB

- `find`: retrieve documents matching a query
- `findOne`: retrieve a single document
- Query operators: `$gt`, `$lt`, `$in`, `$regex`, etc.

```
db.movies.find(      // Find the title of all Sci-Fi movies
    {"genres": "Sci-Fi"},
    {"title": 1, "_id": 0}
)
```

```
SELECT title FROM movies WHERE genres = 'Sci-Fi'
```

Update

Update Operations in MongoDB

- `updateOne`: update the first document matching the filter
- `updateMany`: update all documents matching the filter
- Update operators:
  - `$set`: Set field value
  - `$unset`: Remove field
  - `$push`: Add to array
  - `$inc`: Increment numeric value

```
db.movies.updateOne(              // Update movie rating
  {"title": "Inception"},
  {"$set": {"rating": 8.9}}
)
db.movies.updateOne(              // Add award to movie
  {"title": "Inception"},
  {"$push": {"awards.wins": ["Best Visual Effects"]}}
)
```

There is also `replaceOne` to replace the entire document.

Delete

Delete Operations in MongoDB

- `deleteOne`: delete the first matching document
- `deleteMany`: delete all matching documents

```
// Delete a specific movie
db.movies.deleteOne({"title": "Inception"})

// Delete all movies from a year
db.movies.deleteMany({"year": 2010})
```

⚠ Delete operations cannot be undone: always verify filters

Aggregation

Aggregation operations process multiple documents and return computed results.

Simple Aggregation Methods

- `countDocuments`: count documents in a collection
- `distinct`: find distinct values for a field

```
// Count movies by genre
db.movies.countDocuments({"genres": "Sci-Fi"})

// Find all unique directors
db.movies.distinct("director")
```

⚠ For complex aggregations, use **aggregation pipelines**.

Aggregation Pipelines

Definition (Aggregation Pipelines)

A sequence of data processing stages where each stage transforms the documents as they flow through sequentially.

Common Pipeline Stages

- `$match`: Filter documents
- `$group`: Group documents
- `$sort`: Sort documents
- `$unwind`: Deconstruct array
- `$addFields`: Add fields

```
pipeline = [
    {"$stage1": {...}},  // First transformation
    {"$stage2": {...}},  // Second transformation
    ...
]
```

Aggregation Example: What Does it Compute?

```
db.movies.aggregate(
[    // Pipeline
     {"$match": {"rating": {"$gte": 8.0}}},   // Stage 1
     {"$group": {                              // Stage 2
         "_id": "$director",
         "count": {"$sum": 1},
         "avg_rating": {"$avg": "$rating"}
     }},
     {"$sort": {"avg_rating": -1}},            // Stage 3
     {"$limit": 3}                             // Stage 4
])
```

Aggregation Example: What Does it Compute?

```
db.movies.aggregate(
[    // Pipeline
     {"$match": {"rating": {"$gte": 8.0}}},  // Stage 1
     {"$group": {                             // Stage 2
         "_id": "$director",
         "count": {"$sum": 1},
         "avg_rating": {"$avg": "$rating"}
     }},
     {"$sort": {"avg_rating": -1}},           // Stage 3
     {"$limit": 3}                            // Stage 4
])
```

This pipeline finds the top 3 directors with the highest average ratings among movies rated 8.0 or higher, showing how many qualifying movies each director has and their average rating.

Data Modeling in MongoDB

Embedding

- Store related data in same document
- Better for read performance
- Use when data is frequently accessed together
- Use when data has one-to-few relationships

Referencing

- Store references between documents
- Better for write performance
- Use when data changes frequently
- Use for many-to-many relationships

```
{
  "title": "The Dark Knight",
  "cast": [
    {"name": "Christian Bale",
     "role": "Bruce Wayne"},
    {"name": "Heath Ledger",
     "role": "Joker"}
  ]
}
```

```
{
  "title": "The Dark Knight",
  "actor_ids": [
    ObjectId("..."),
    ObjectId("...")
  ]
}
```

Takeaways about MongoDB

- Document-oriented database (JSON-like documents stored in collections)
- Offers both local and cloud-based solutions

SQL	**MongoDB**
Tables	Collections
Rows	Documents
Columns	Fields
Joins	Embedded documents or references
GROUP BY	`$group` aggregation stage

Conclusion

Takeaways: NoSQL

1. NoSQL databases address challenges of distributed systems and semi-structured data
2. The **aggregate** is the fundamental unit of organization in NoSQL databases
3. Data modeling is crucial for performance in NoSQL databases
4. Four main families of NoSQL databases: Key-value, Document-oriented, Column-oriented, Graph databases
5. NoSQL databases complement rather than replace relational databases (polyglot persistence)

Forget SQL vs NOSQL!