NoSQL Databases

Jana Kostičová

Vybrané technológie analýzy dát, FMFI UK

12.12.2024

A bit of history

- Non-relational databases existed since 1960s
  - Hierarchical DBs, Network DBs, Object-oriented DBs, XML DBs, …
- **1998:** Carlo Strozzi used the term "NoSQL" for his relational database that does not use SQL:
  - Strozzi NoSQL
- **2009:** Johan Oskarson organized a meetup called "NoSQL 2009" on newly appeared distributed non-relational databases

→ **"NoSQL movement"**

NoSQL was first interpreted as "no SQL", later "not only SQL"

*STROZZI: "NoSQL is **a relational database** to all effects and just it **intentionally does not use SQL** as a query language"*

*"The newborn NoSQL movement **departs from the relational model** altogether and it should therefore have been called more appropriately **"NoREL"**, or something to that effect, since its not being SQL-based is just an obvious consequence of not being relational, and not the other way around."*

Today's reality:

- NoSQL databases = non-relational databases that has been part of "NoSQL movement"
- Many of these databases support SQL-like query languages

“NoSQL movement”

- Key-value stores
  - 2003 Memcached
  - 2009 Redis
- Document stores
  - 2005 CouchDB
  - 2009 MongoDB
- Column-wide stores
  - 2007 Apache HBase
  - 2008 Apache Cassandra
- Graph databases
  - 2005 AllegroGraph
  - 2007 Neo4j

+ many more appeared later

NoSQL databases - common properties

- Non-relational logical structure
- SQL-like or custom query language
- *Often* schema-less or with flexible schema → suitable for storing <u>semi-structured data</u>
- *Mostly* designed for distributed environment and high scalability
- *Mostly* relaxing ACID
- *Mostly* limited:
  - Referential integrity and constraints
  - The complexity of queries
- *Many* are primarily designed for cloud environment
  - Some of them designed <u>exclusively</u> for specific cloud: Amazon DynamoDB, Google BigTable, ..

NoSQL databases - common properties

- Non-relational logical structure
- SQL-like or custom query language
- *Often* schema-less or with flexible schema → suitable for storing <u>semi-structured data</u>
- *Mostly* designed for distributed environment and high scalabilitzý
- *Mostly* relaxing ACID
- *Mostly* limited:
  - Referential integrity and constraints
  - The complexity of queries
- *Many* are primarily designed for cloud environment
  - Some of them designed <u>exclusively</u> for specific cloud: Amazon DynamoDB, Google BigTable, ..

Logical data structure

1. Key-value pairs (key value stores)
2. Documents accessed by keys (document stores)
3. Wide-column format (wide-column stores)
4. Graph (graph databases)

Example

Product catalog in e-commerce store

Common product attributes: code, name, description, price, …

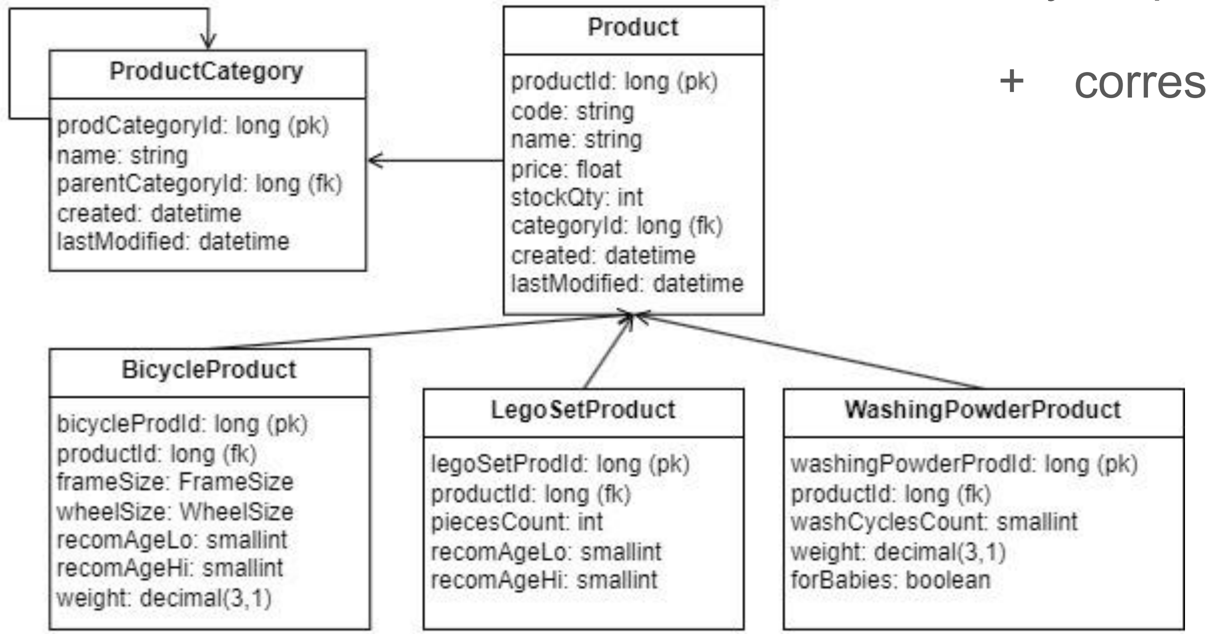
Category-specific attributes:

Category / attribute	Frame size (enum)	Wheel size (enum)	Recommended age (range of positive numbers)	Weight (decimal number)	Number of pieces (positive number)	Number of washing cycles (positive number)	For babies (yes/no)
Bikes	X	X	X	X			
Lego sets			X		X		
Washing powders				X		X	X

Relational model;

- Naive approach - wide, sparse table → inefficient

Relational logical data model 1



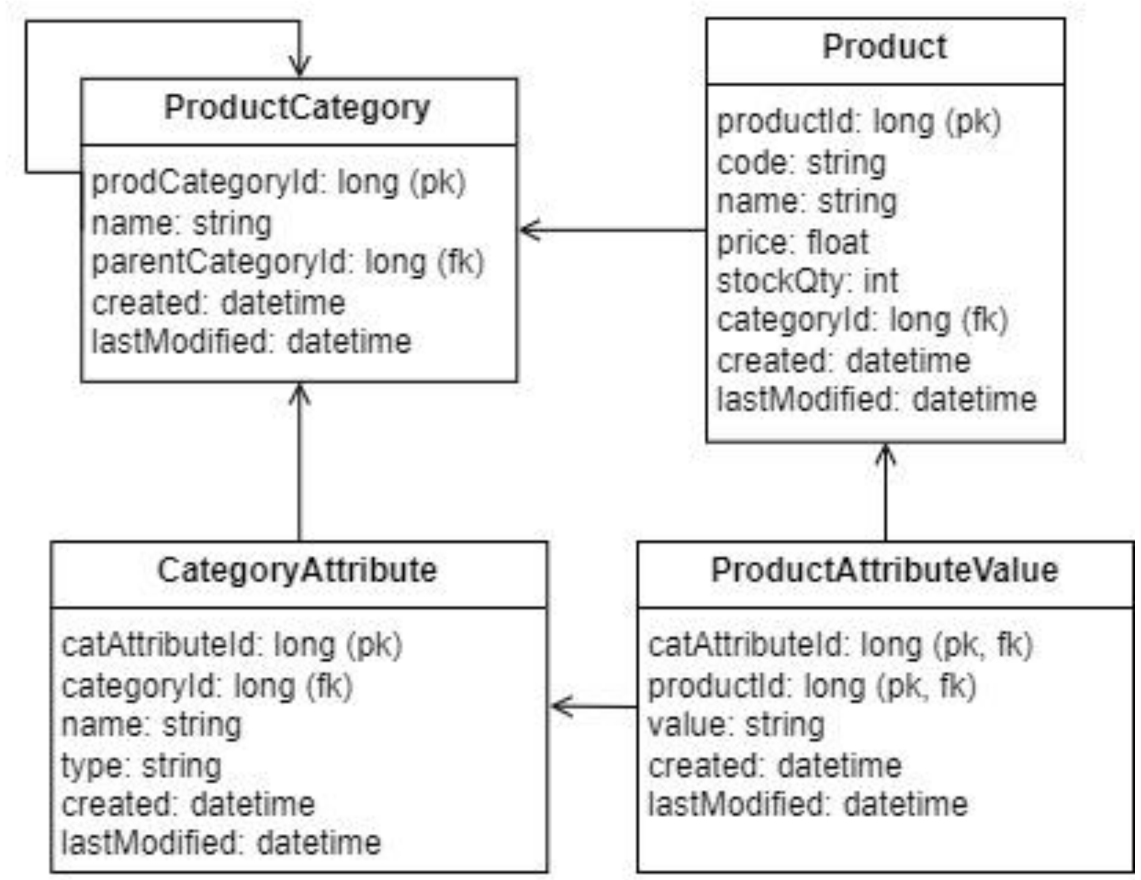
Polymorphic schema

+ corresponding constraints

Relational logical data model 2

EAV = entity-attribute-value model

- corresponding constraints

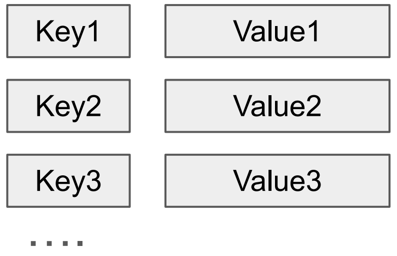


Key-value stores (1)

- Unique keys - mostly strings
  - "KA23E", "product.KA23E", …
- Values:
  - Various types - string, set, list, JSON object, …
  - "Opaque" to the key-value store
- Only simple queries based on the key

- Most common use cases
  - Fast key-based lookups in distributed environment
  - In-memory key-value stores (Redis, Memcached) are used as distributed caches to store precomputed or intermediate results
    - Data caching for faster analytics
    - Data buffering in real-time stream processing pipelines
    - Temporary data storage for data transformation

- *Redis, Memcached, Amazon DynamoDB, CouchBase*



Key value stores (2)

Key	Value
product:KA23E	{"name": "BestBike Junior", "description": "This is the best bicycle for 4-6 years old kids"}
product:XT78S	{"name": "Lego SuperMario", "description": "Luigi's mansion", ...}
product:BG234W	{"name": "SuperColor Washing Powder", "description": "Bright colors!", ...}

```
SET product:KA23E '{"name": "BestBike Junior",...}'
```

- SETNX key value (set if not exists)
- MSET key1 value1 key2 value2 … (set multiple key-value pairs)

```
GET product:KA23E
```

- MSET key1 key2 … (get values for multiple keys)

```
EXISTS product:KA23E
DEL product:KA23E
```

Example key-based operations in Redis

Document stores

- A document is identified by a unique key
- Document format
  - JSON (BSON), XML, YAML, ..
- Less joins are needed

- Most common use cases:
  - Processing hierarchical data and/or semi-structured data in distributed environment
  - (outside data analytics) Fast and flexible development of web application
    - MERN stack: MongoDB, Express.js, React, Node.js

- *MongoDB, CouchDB, Couchbase, …*

```
[
    {
        "code": "KA23E",
        "name": "BestBike Junior",
        "description": "This is the best bicycle
           for 4-6 years old kids",
        "category": "Bicycles",
        "frameSize": "15",
        "wheelSize": "16",
        "recAgeLo": "4",
        "recAgeHi": "7",
        "weight": "4.5"
    },
    {
        "code": "XT78S",
        "name": "Lego SuperMario",
        "description": "Luigi's mansion",
        "category": "Lego sets",
        "piecesCount": "450",
        "recAgeLo": "7"
    },
    {
        "code": "BG234W",
        "name": "SuperColor Washing Powder",
        "description": "Bright colors!",
        "category": "Washing powders",
        "washCyclescount": "30",
        "weight": "3"
    }
]
```

JSON shredding

```
[
   {
    "firstName": "Wood",
     "lastName": "Lyons",
     "personalId": 87695,
     "addresses": [
       {
         "street": "Ralph Avenue",
         "number": 895,
         "city": "Bynum",
         "postalCode": 9981,
       },
       {
         "street": "Surf Avenue",
         "number": 386,
         "city": "Fairlee",
         "postalCode": 4470,
       },
       {
         "street": "Hull Street",
         "number": 210,
         "city": "Rockhill",
         "postalCode": 5301
       }
     ]
   }
 ]
```

Persons

_id	firstName	lastName	personalId
1	Wood	Lyons	87695
…	…	…	…

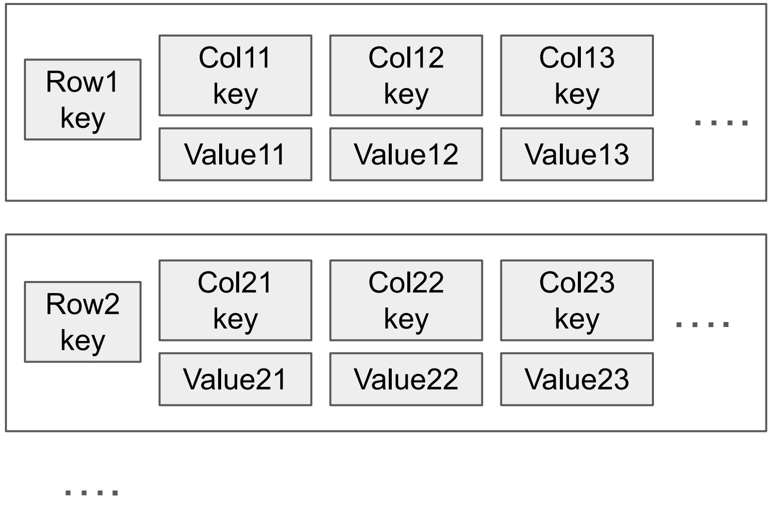
Addresses

id	street	number	city	postal Code	person Id
1	Ralph Avenue	895	Bynum	9981	1
2	Surf Avenue	386	Fairlee	4470	1
3	Hull Street	210	Rockhill	5301	1
…	…	…	…	…	…

Wide-column stores

- Unique row keys
- Column keys:
  - Not prescribed, each row can have different columns
  - Unique for a specific row key
- Row-based queries
  - Single key / range
- Column-based queries
  - Selective column retrieval
- Use cases:
  - Semi-structured / evolving data, sparse data, time series data (timestamps in row keys)
  - Applications that require scalability, high-throughput read and write operations

- *Apache Cassandra, Apache HBase, Google BigTable*



Another abstractions:
- 2-dimensional array of key-value pairs
- Sparse matrix

Graph databases

- Store efficiently (and natively) graph structures
  - Nodes, edges and their properties
- Highly scalable (Neo4j from 2020), ACID is mostly guaranteed
- Use cases;
  - Processing graph data, also in distributed environment

- *Neo4j, AllegroGraph, ..*

Example - Neo4j

- Nodes: id, properties (key-value pairs), labels and relations (=edges)
- Edges: type (FRIENDS, LIKES, BELONGS), direction and properties (key-value pairs)

Example

Cypher language - find all movies in which Tom Hanks acted:

```
MATCH (tomhanks:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movies)
RETURN movies;
```

NoSQL databases - common properties

- Non-relational logical structure
- *Often* schema-less or with flexible schema → suitable for storing semi-structured data
- SQL-like or custom query language
- *Mostly* designed for distributed environment and high scalability
- *Mostly* relaxing ACID
- *Mostly* limited:
  - Referential integrity and constraints
  - The complexity of queries
- *Many* are primarily designed for cloud environment
  - Some of them designed <u>exclusively</u> for specific cloud: Amazon DynamoDB, Google BigTable, ..

Centralized database vs distributed database

Centralized database

=> runs and stores data in a single machine

Distributed database

=> runs and stores data across multiple computers (possibly in multiple physical locations)

Why to distribute data? - avoid single point of failure, scalability, availability, reliability, response time, ..

Drawbacks: increased operational complexity (network communication), increased learning curve, …

Replication vs partitioning (1)

- The process of distributing a database across multiple nodes typically involves two key concepts:
  1. Replication
  2. Partitioning

The process is very similar to the one applied in distributed storage systems (Hadoop, cloud object storages)

- **Replication** - storing separate copies of data at two or more nodes
  - Leader(s) - server(s) that receives writes
  - Single leader, Multileader, Leaderless architectures (single leader is the most common)
- **Partitioning (sharding)** - data are divided into smaller parts and then store on separate nodes, i.e., particular partitions (shards) do not share data

Replication vs partitioning (2)

Replication

→ fault tolerance (avoiding single point of failure)
→ load balancing
→ response time (latency)
→ availability

Partitioning

→ (horizontal) scalability
→ efficient querying

- NoSQL DBs mostly support both concepts extensively
- Automatic process with little manual intervention
- Many NoSQL DBs can be used also in a single machine

ACID (RDBMSs)

Atomicity: Either all the changes within the transaction are committed to the database, or none of them are.

Consistency: A transaction brings the database from one consistent state to another, maintaining database invariants.

Isolation: Concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.

Durability: Once a transaction is committed, its changes are permanent and will survive system failures, such as power outages or crashes.

Examples: financial systems, healthcare databases

CAP theorem

We cannot achieve all consistency, availability, and partition tolerance in asynchronous network model.

- **Consistency:** Every read operation returns the most recent write result (= strong consistency)
- **Availability:** Every request receives a response (without guaranteeing it's the most recent data).
- **Partition tolerance:** The system can continue to operate even in the presence of network partitions or communication failures.

Distributed DB must always guarantee partition tolerance. Thus it has do decide between **consistency** and **availability.**

ACID vs CAP consistency

If a distributed database guarantees ACID consistency, it must also guarantee CAP consistency.

If a distributed database guarantees CAP consistency, it must compromise availability.

ACID Consistency: A transaction brings the database from one consistent state to another, maintaining database invariants.

CAP consistency: Every read operation returns the most recent write result

Compromised availability

Why NoSQL DBs are easier to distribute

- Relaxed ACID guarantees & no integrity constraints
  - Less communication between network nodes, less locks
  - **BASE approach** (eventual consistency)
- Flexible schema or schema-less approach
  - Easier partitioning
  - Faster writes (no need to validate data against schema)
- Related data are stored together (document stores)
  - Less joins between documents
  - Less communication between network nodes
- Support for distributed environment by design
  - Both for replication and partitioning

Not all NoSQL databases provide the same level of support for horizontal scaling!

BASE (NoSQL DBs)

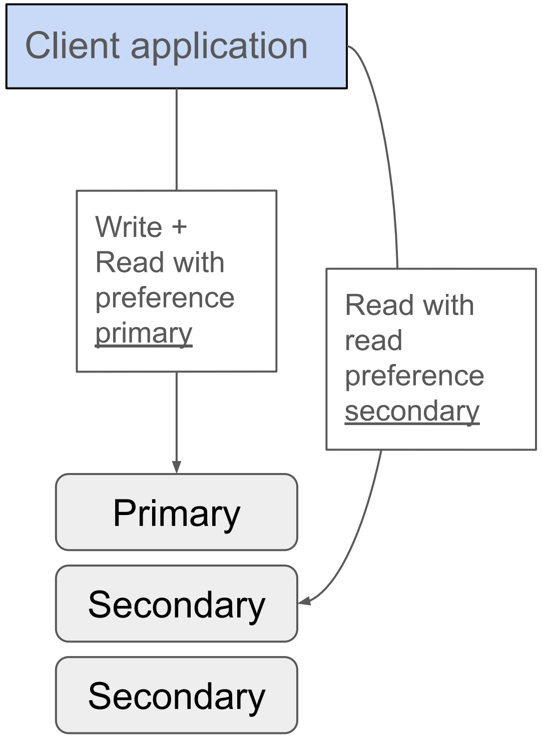
Basically Available: The system prioritize high availability, even in case of network partitions or failures.

Soft state: The system can be in a "soft" or intermediate state, which means that data consistency is not guaranteed at all times.

Eventually consistent: Data consistency is achieved over time. There is no requirement for immediate consistency, and different replicas of the data may be out of sync temporarily. However, over time, the data will become consistent through mechanisms like background reconciliation and conflict resolution.

Examples: social media platforms, distributed content delivery networks

Example - MongoDB replication


Client application
Write + Read with preference primary
Read with read preference secondary
Primary
Secondary
Secondary

- Single leader replication
- **Read preference**
  - Specifies where the data are read from
  - Primary, secondary, primaryPreferred, secondaryPreferred, nearest
- **Write concern**
  - Specifies how many replicas must acknowledge a write operation before it is considered successful
    - w = 1: primary node only - lowest latency, highest risk of data loss
    - w = majority: majority of replicas
    - w = all: all replicas - highest latency, highest data durability
- If primary node fails, a new one is elected
- Distributed transaction with read operations must use read preference primary

New approaches in “relational” world

- Goal: Relational tables + horizontal scalability

1. **Columnar databases**
   - Columnar physical storage
   - ACID not guaranteed
   - Primarily intended for analytical processing
   - *Google BigQuery, Amazon Redshift, Apache Druid, Apache Kudu*
2. **NewSQL databases**
   - Mostly row-based physical storage
   - ACID guaranteed
     - Availability is compromised in case of network partitions
   - Primarily intended for transactional processing (i.e., sources for data analytics)
   - *Google Spanner, Cockroach DB, …*
3. **Support for horizontal scaling in traditional RDBMS**
   - Automatic replication mostly supported
   - Partial support for partitioning (often manual approaches through 3rd party tools)

columnar DBs
≠
wide-column DBs

Resources

- Robert Lukotka: Persistence and Databases
- C. M. Ricardo, Susan D. Urban: Databases Illuminated, 3rd edition, 2015.
- Wikipedia: NewSQL
- S.Gilbert, N.Lynch: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant webservices
- Cassandra Documentation
- MongoDB Documentation
- Redis Documentation