Lecture 01

Motivation, Evolution, and Big Data

NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Data as Infrastructure of Civilization

Human civilization always produced data. But never at this scale. Never this fast. Never this diverse.



4 000 – 3 000 BC
clay tables



accounts of the bailiffs
16th century

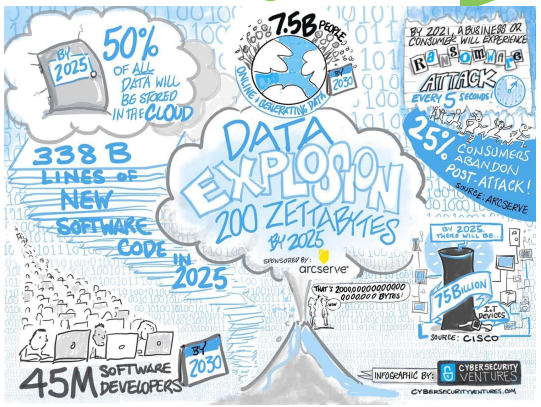


tax reform
King Gustav Vasa



20th century
paper filing cabinets

21st century

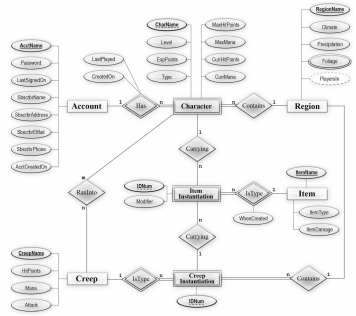


200 000 000 000 TB

Three Views of Data

Conceptual

- What exists in reality
  - Objects and relationships
- ER, UML, …



Logical

- How data is represented
- **Relational**, JSON, graph, …

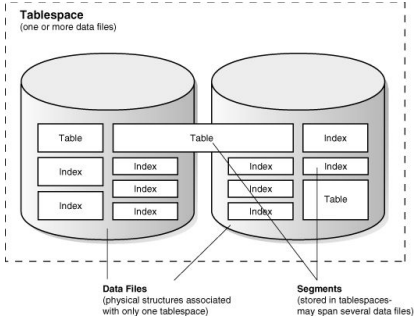
student_id	name	age
1	Akon	17
2	Bkon	18
3	Ckon	17
4	Dkon	18

subject_id	name	teacher
1	Java	Mr. J
2	C++	Miss C
3	C#	Mr. C Hash
4	Php	Mr. P H P

student_id	subject_id	marks
1	1	98
1	2	78
2	1	76
3	2	88

Physical

- How data is stored
- Files, indexes, …



Does Database Mean Relational?

- A common assumption for many years
- Why have relational databases dominated:
  - Simplicity
    - of the model
    - of the query language
  - Mature systems
  - Strong theory
  - …



VRTY

KOD	MISTO	X	Y	Z	PRUMER	HLOUBKA	TYP_CERP	PRIKON	VYKON
A12	Janov	485	1202	322	20	860	C-7a	2500	26
A18	Popice	477	1203	306	20	920	C-7a	3000	32
C22	Cerkev	481	1195	242	15	750	Brum	2000	11
S11	Malkov	480	1199	298	22	800	AVG	2500	29
A45	Borek	479	1207	331	15	840	SLIT	1800	17
B02	Mostek	475	1196	240	25	980	Brum	2000	11
B17	Svatava	483	1205	362	20	870	SLIT	1800	17
B18	Budov	486	1208	377	15	940	SLIT	1800	17

SOUVRSTVI

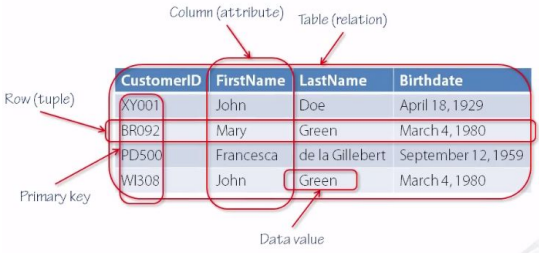
KOD	NAZEV
O	Ostravské souvrství
K	Karvinské souvrství

VRSTVY

SOUVRSTVI	KOD	STRED_MOC	PLOCHA
O	R02	42	6500
O	R05	17	7100
O	R07	23	6100
O	R10	11	5900

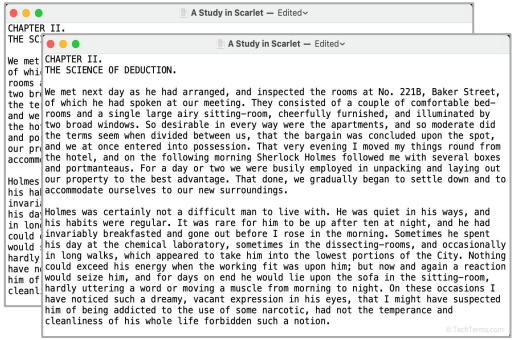
Relational databases became the default — not the universal solution.

Relational Model (Codd, 1970)



- Core idea:
  - Represent data as relations (tables)
  - Manipulate them using declarative logic
- Concept:
  - Data = sets of tuples
  - Schema = relation definition
  - Integrity constraints
  - Declarative queries (SQL)
- Why it was revolutionary?
  - Separation of logical & physical layer
  - Data independence
  - Set-based operations
  - Strong mathematical basis

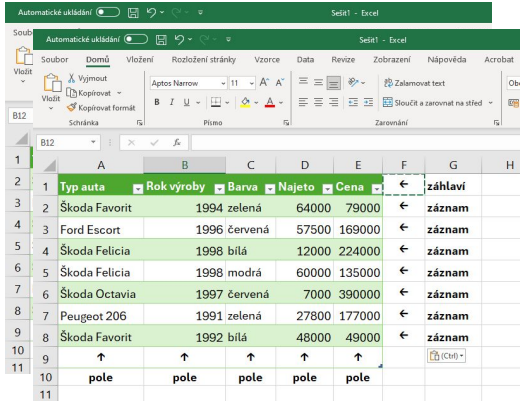
Not All Data Is Tabular



Documents, logs, e-mails, …

Connected

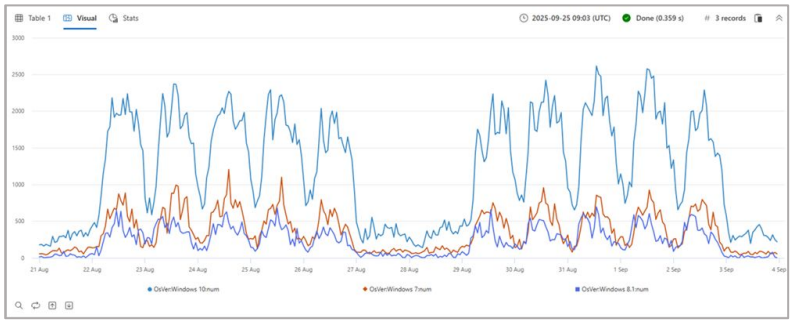




Tabular



Multimedia



Time series

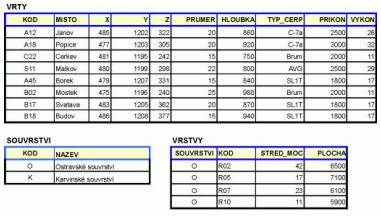


Spatial

One Person, Multiple Data Representations

Loan Application





VRTY

KOD	MISTO	X	Y	Z	PRUMER	HLOUBKA	TYP_CERP	PRIKON	VYKON
A12	Janov	485	1202	322	20	860	C-7a	2500	26
A18	Popice	477	1203	305	20	920	C-7a	3000	32
C22	Cerkev	481	1195	242	15	750	Brum	2000	11
S11	Malkov	480	1199	298	22	800	AVG	2500	29
A45	Borek	479	1207	331	15	840	SL1T	1800	17
B02	Mostek	475	1196	240	25	980	Brum	2000	11
B17	Svatava	483	1205	362	20	870	SL1T	1800	17
B18	Budov	486	1208	377	15	940	SL1T	1800	17

SOUVRSTVI

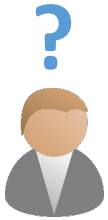
KOD	NAZEV
O	Ostravské souvrství
K	Karvinské souvrství

VRSTVY

SOUVRSTVI	KOD	STRED_MOC	PLOCHA
O	R02	42	6500
O	R05	17	7100
O	R07	23	6100
O	R10	11	5900

PostgreSQL

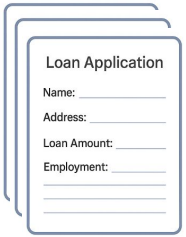
```
SELECT SUM(amount) AS monthly_income
FROM Transactions
WHERE customerId = 42
  AND type = 'INCOME'
  AND date >= CURRENT_DATE - INTERVAL '30' DAY;
```



Task:

Find everything about customer X that might affect credit risk:

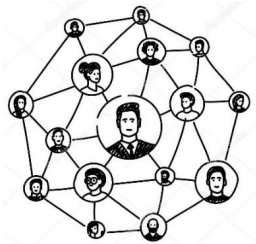
- financial history
- declared situation
- risky connections



mongoDB

```
db.loanApplications.find(
  { customerId: 42 },
  { declaredIncome: 1, employer: 1, loanPurpose: 1 }
)
```





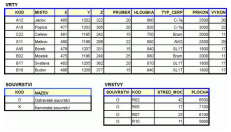
neo4j

```
MATCH (c:Person {id: 42})
      -[:FRIEND | CO_OWNER | LIVES_WITH*1..2]-
      (related)
RETURN related;
```

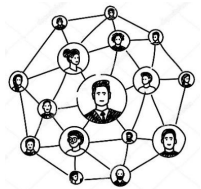
One Person, Multiple Data Representations

- Conceptual entity: one
- Logical representations: many
- Physical systems: even more

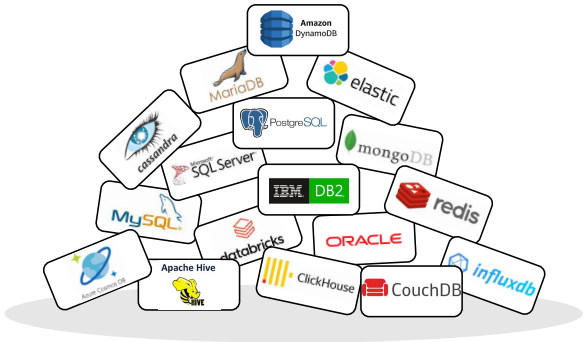




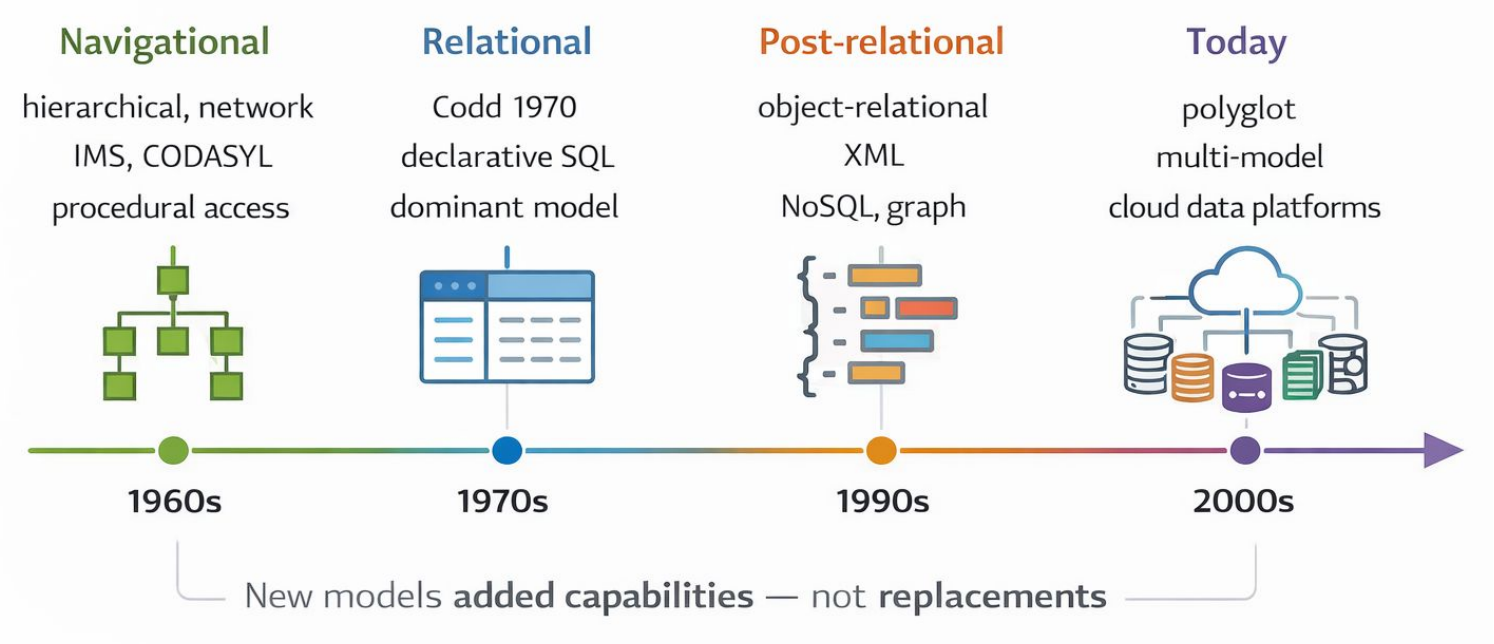




Message: Real-world entities are fragmented across data models and systems



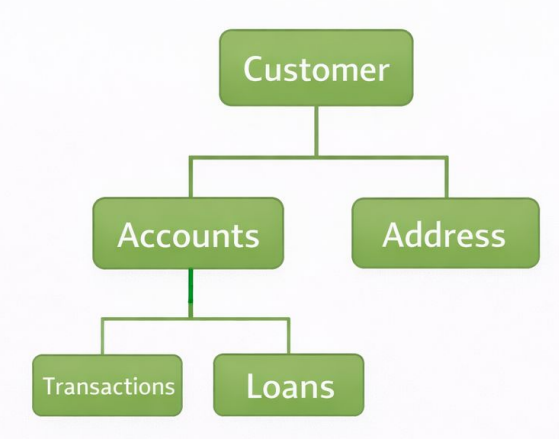
Evolution of Data Models



Hierarchical Model (1960s)

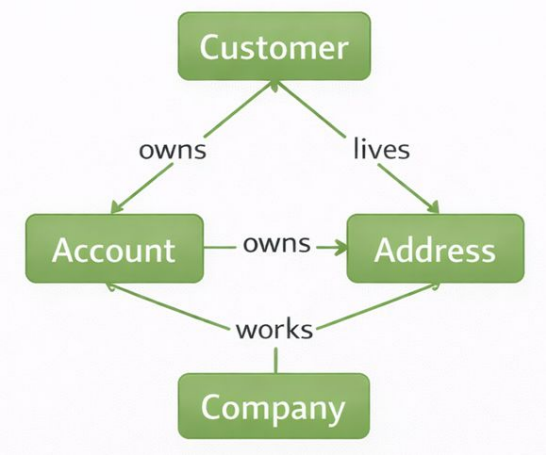
IBM IMS (1968)
is still used

- Data organized as trees
  - One-to-many relationships
- Procedural navigation (root → child)
  - Depth-first, left-to-right traversal order
- No logical data independence → application must know how data are stored



Network Model (1960s)

- Records connected as a graph
  - Supports many-to-many relationships
- Multiple access paths possible
  - Still procedural navigation
- Flexibility improved, independence not



CODASYL model

- Conference on Data Systems Languages
  - Defined the first database standards
  - Influenced early SQL standardization

Relational Model (1970s)

- Core idea
  - Represent data as relations
  - Query them declaratively
- The breakthrough
  - Data independence
    - Applications do not depend on storage structure
  - Set-based operations
  - Declarative queries (SQL)
    - Before: navigate how
    - After: declare what
  - Logical ≠ physical storage



Customer

id	name	risk
1	Alice	0.8
2	Bob	0.2
3	Charlie	0.9

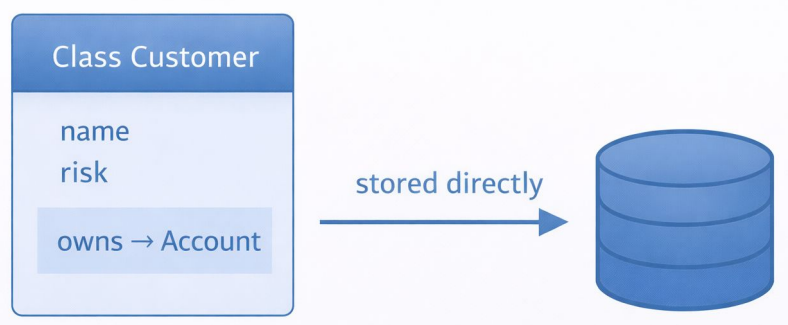
Order

id	customer_id	amount
101	1	250
102	2	100
103	1	300

Object Model & Object Databases (1990s)

- Relational model ≠ natural for objects
  - Mismatch with OO programming
- Store objects directly:
  - classes
  - inheritance
  - object identity
  - references
- No mapping to tables
- Never replaced relational databases

Objects ≠ tables
Identity ≠ primary key
References ≠ foreign keys



Object-Relational Databases (late 1990s–2000s)

- Idea: Extend relational model with richer types
  - Need to store:
    - multimedia
    - spatial data
    - complex objects
- Added features
  - User-defined types
  - Nested structures
  - Inheritance
  - References
- Relational model survived by evolving

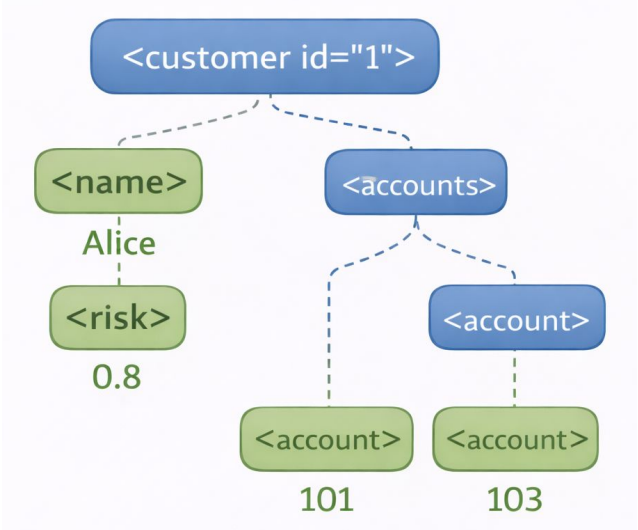


XML Model & XML Databases

- Context
  - Web exploded
  - Data became document-oriented and semi-structured
- Core idea
  - Data represented as nested elements:
    - Hierarchical
    - Self-describing
    - Flexible schema
- What changed
  - No fixed table structure
  - Irregular nesting allowed
  - Schema optional

Word Wide Web Consortium

W3C XML Recommendation (1998)
XML became standard for data exchange



But the Relational Model Still Dominates…

Core reasons

- Strong mathematical foundation
- Declarative query language (SQL)
- Optimizer & cost-based execution
- Standardization (SQL ISO)
- Huge ecosystem & tooling



Relational systems adapted instead of being replaced.

Then Big Data Arrived…



What is Big Data?

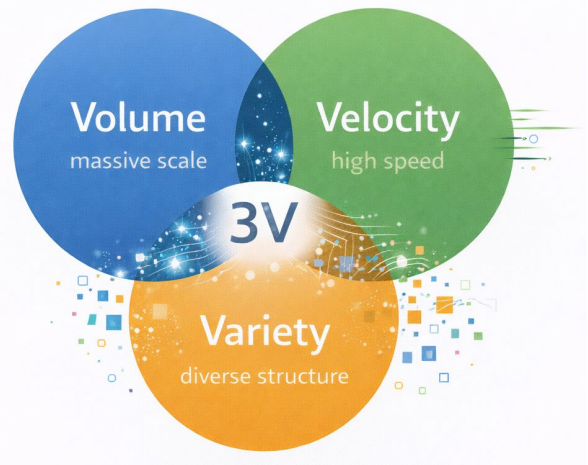


"Big data is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it."

Dan Ariely

What is Big Data?

Gartner (2001): *"**Big Data**" is high **v**olume, high **v**elocity, and/or high **v**ariety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.*



http://www.gartner.com/

Who is Gartner ?

- Information technology research and advisory company
  - Founded in 1979 by Gideon Gartner
- HQ in Stanford, Connecticut, USA
  - > 5,300 employees
  - > 12,400 client organizations
- Provides: competitive analysis reports, industry overviews, market trend data, product evaluation reports, …

Origins of Big Data



Social media and networks
(all of us are generating data)



Mobile devices
(tracking all objects)



Sensor networks
(measuring all kinds of data)



Scientific instruments
(collecting all sorts of data)

Data started to grow faster than traditional systems could handle.

Example: FaceBook

Statistics from 2010



- 500 million users
- 570 billion page views per month
- 3 billion photos uploaded per month
- 1.2 million photos served per second
- 25 billion pieces of content (updates, comments) shared every month
- 50 million server-side operations per second

2008: 10,000 servers
2009: 30,000 servers
…

→ One RDBMS may not be enough to keep this going on!

http://royal.pingdom.com/2010/06/18/the-software-behind-facebook/

63 Facebook Statistics You Need to Know in 2022

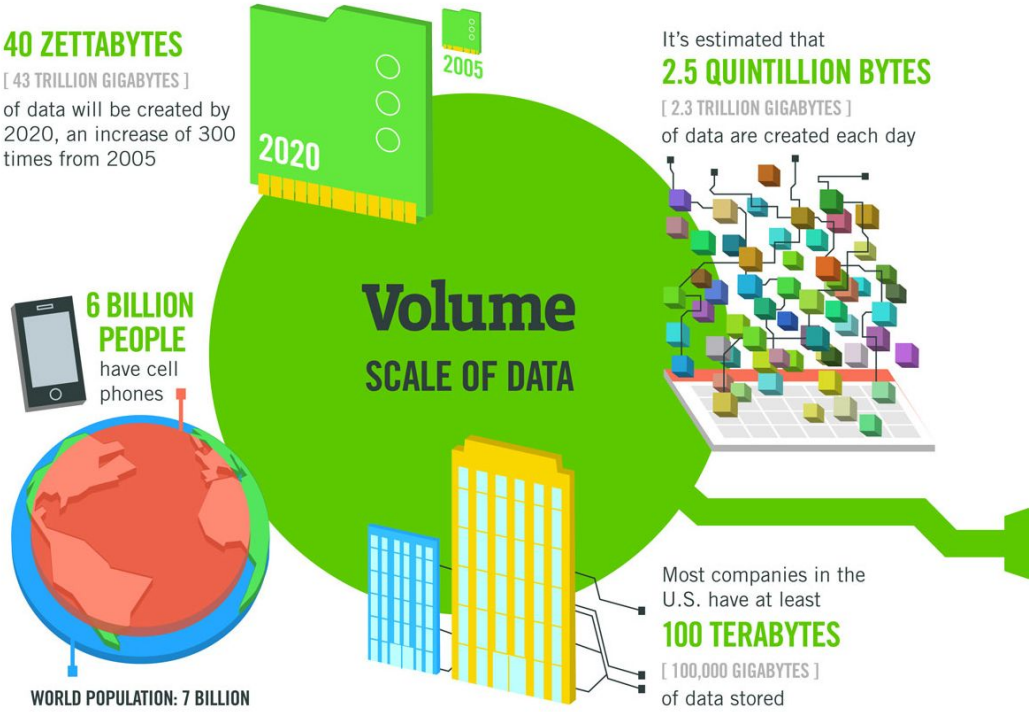


- 2.91 billion monthly active users
- Facebook has over 10 million advertisers
  - A Facebook user clicks on 12 ads on average every month
- Users spend 34 minutes on Facebook every day
- Over 3.5 billion live feeds on Facebook towards the end of 2018
- 500 million people use Facebook Stories daily

https://www.omnicoreagency.com/facebook-statistics/

http://www.ibmbigdatahub.com/

Big Data Characteristics: Volume (Scale)

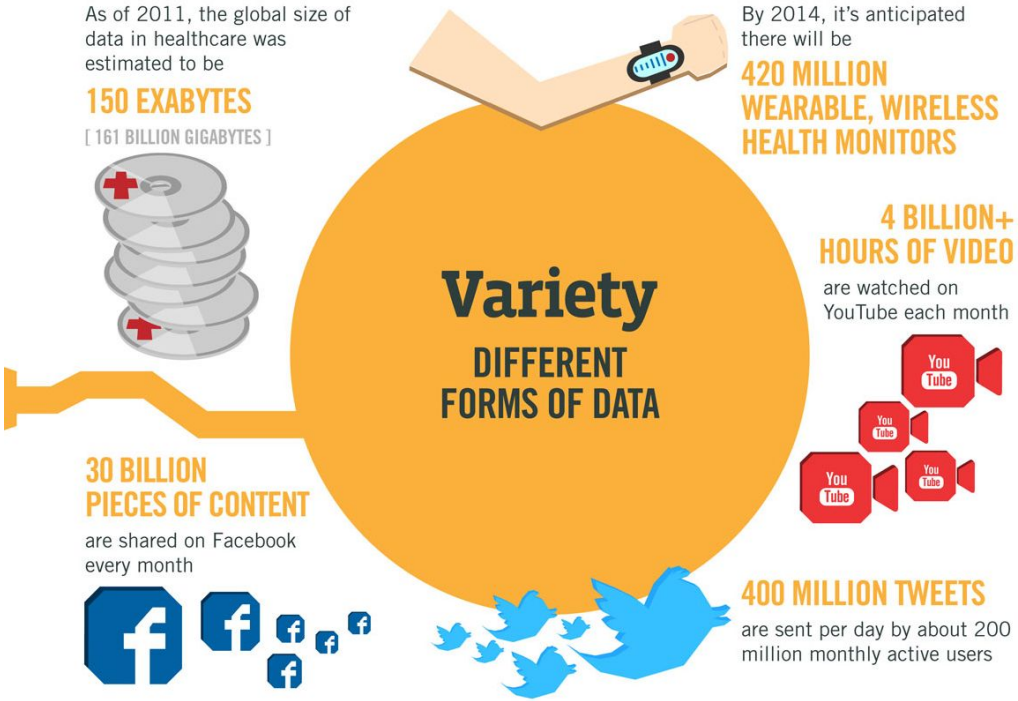


Massive data volume

- terabytes ⟶ petabytes ⟶ exabytes
- continuous growth
- data stored across many machines

Data no longer fit on a single server.

Big Data Characteristics: Variety (Complexity)

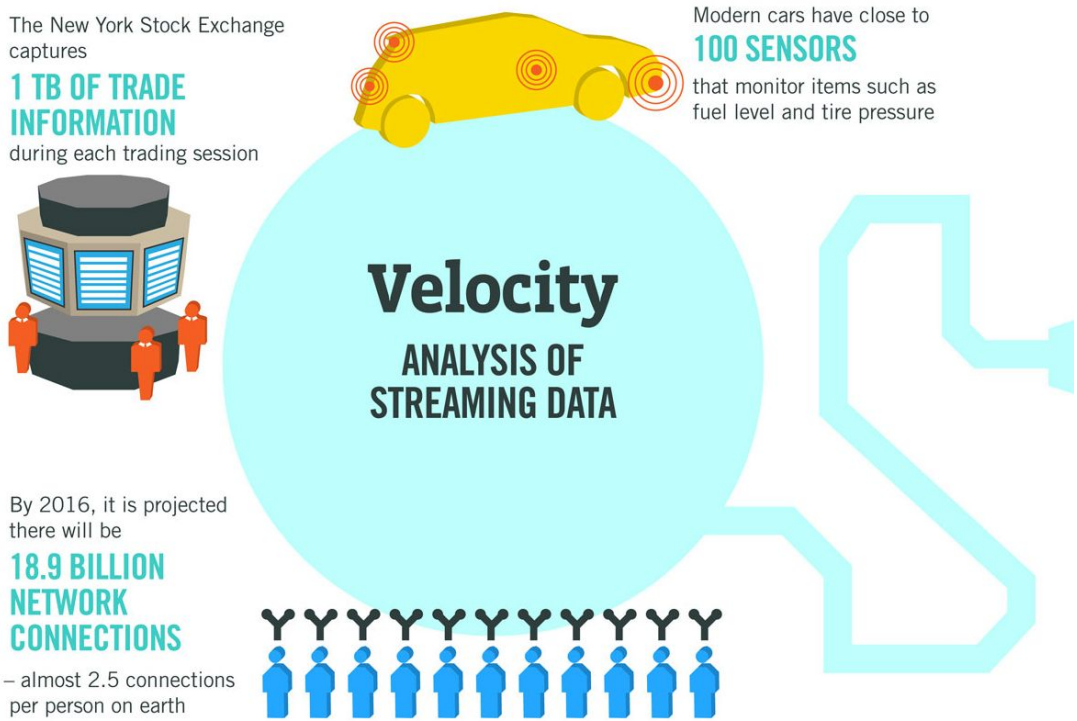


Data comes in many forms

- structured (tables)
- semi-structured (JSON, XML)
- unstructured (text, images, video)
- graph & network data

Not all data fits into relational tables.

Big Data Characteristics: Velocity (Speed)

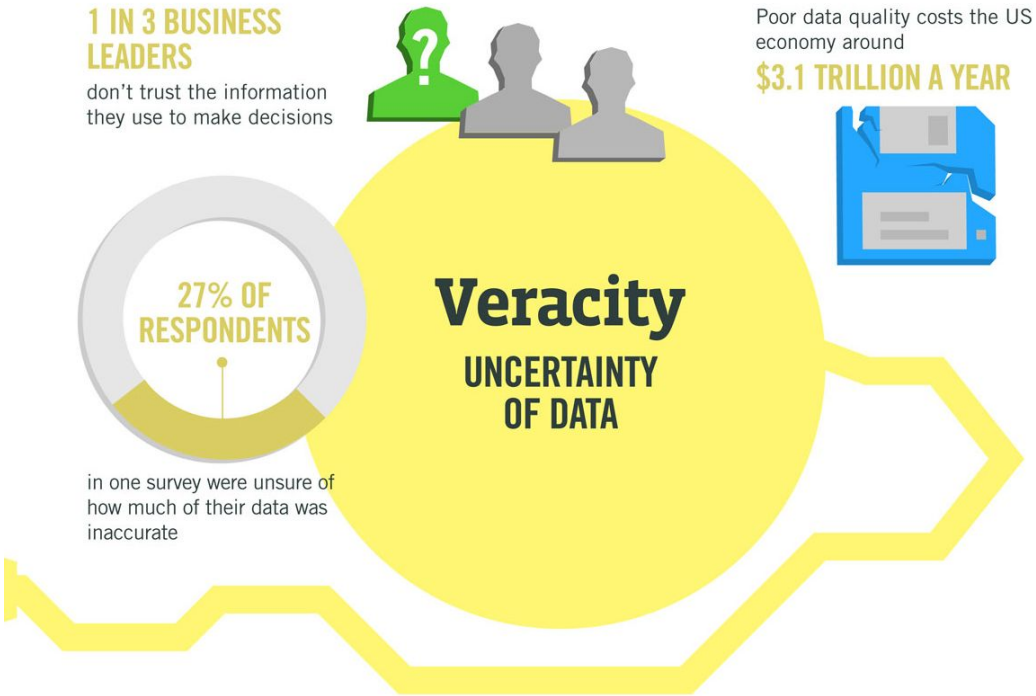


Continuous data streams

- real-time user interactions
- sensor streams
- financial transactions
- logs & monitoring data

Data must be processed in real time.

Big Data Characteristics: Veracity (Uncertainty)



Data may be:

- incomplete
- noisy
- inconsistent
- uncertain

Sources differ in reliability.

More data does not always mean better data.

How Many V’s Do We Need?

Beyond Volume, Velocity, Variety, Veracity:

- **V**alue — extracting useful insights
- **V**alidity — correctness & trustworthiness
- **V**olatility — data lifetime & relevance
- **V**ariability — changing meaning & context
- …

Big Data is not defined by a fixed number of V’s.

Changing Data Assumptions

RDBMS	NoSQL
integrity is mission-critical	OK as long as most data is correct
data format consistent, well-defined	data format unknown or inconsistent
data is of long-term value	data are expected to be replaced
data updates are frequent	write-once, read multiple (no updates, or at least not often)
predictable, linear growth	unpredictable growth (exponential)
non-programmers writing queries	only programmers writing queries
regular backup	replication
access through master server	sharding across multiple nodes

Why Traditional Database Systems Struggled

Traditional DB assumptions

- data fits on one machine
- mostly structured
- moderate update rate
- centralized architecture

Big Data reality

- distributed storage
- heterogeneous data
- high-speed streams
- massive scale

Scaling vertically was no longer enough.

Vertical Scaling (scaling up)

Traditional approach

- strong consistency
- centralized architecture
- scaling by bigger machines

More powerful server:

- more CPU
- more RAM
- more storage

Works well for many workloads

…but has limits

Limitations

- expensive hardware
- vendor lock-in
- proprietary solutions
- physical limits of a single machine

Horizontal Scaling (scaling out)

Idea

- systems distributed across many nodes
- commodity hardware (cost-effective)
- virtually unlimited scalability
- often surpasses vertical scaling

But distributed systems are hard…

Fallacies of distributed computing

- the network is reliable
- latency is zero
- bandwidth is infinite
- the network is secure
- topology does not change
- there is one administrator
- transport cost is zero
- the network is homogeneous

Scalability solved one problem — created many others.

Modern Database Systems for Big Data

New requirements:

- distributed by design
- horizontally scalable
- fault tolerant
- flexible schemas
- high throughput

Result - new database families:

- NoSQL databases
  - Key/value, column, document
  - Graph
- NewSQL databases
- Multi-model databases
- Array databases
- …

„NoSQL“

NoSQL = Not Only SQL

- Not „no to SQL“
  - Another option, not the only one
- Not „not only SQL“
  - Oracle DB or PostgreSQL would fit the definition

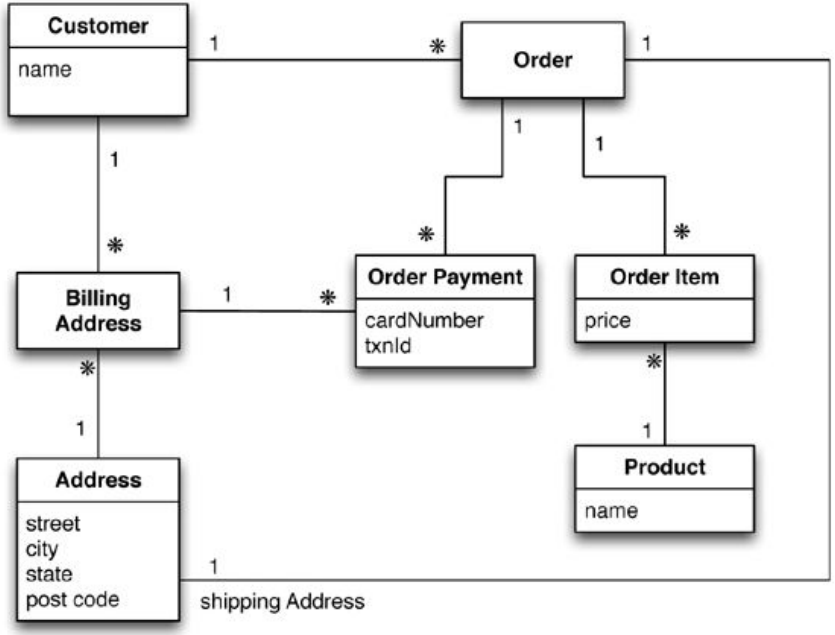
NoSQL movement = “the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for”

NoSQL Data Model - Aggregates

- Relational model
  - Data normalized
  - Many tables
  - JOIN to reconstruct objects
- NoSQL model
  - Data stored together
  - One aggregate = one unit
  - No joins required

Aggregate = data that is stored and accessed together

Relational Model: Normalized Data



Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

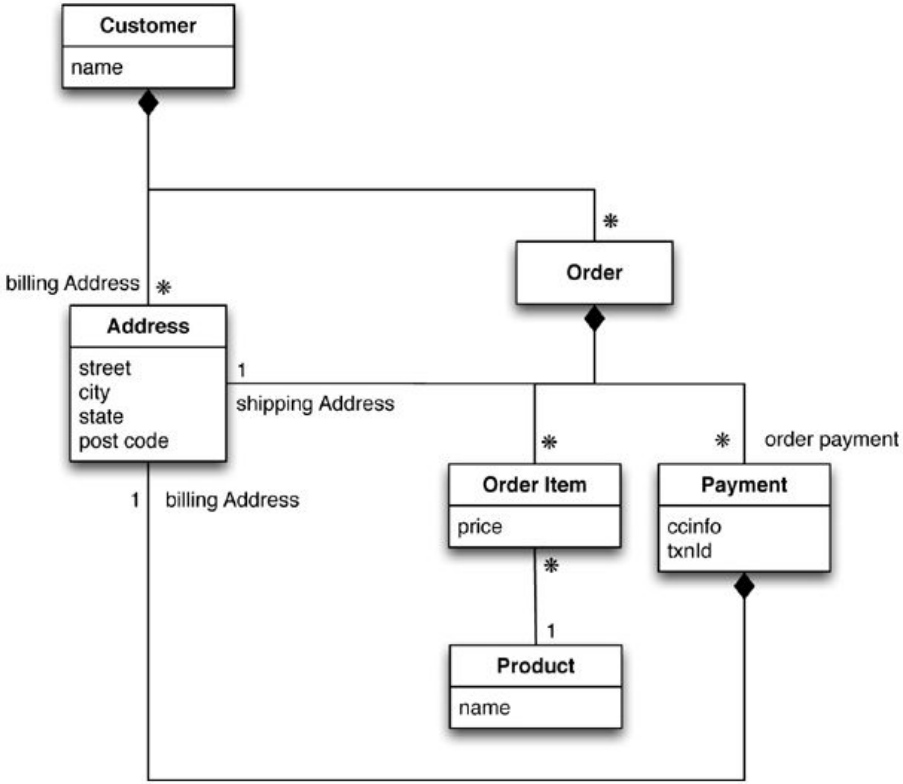
Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft



Aggregate Model (NoSQL)

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
  }]
}
}
```

Aggregate-Oriented vs Aggregate-Ignorant Models

Aggregate-oriented systems

(store data together)

- data grouped into aggregates
- one aggregate = one unit of storage
- retrieved in one read
- no joins needed
- optimized for specific queries

Examples:

document, key–value, column-family

Aggregate-ignorant systems

(store data separately)

- data split across tables
- relationships via references
- joins reconstruct objects
- flexible querying across data
- optimized for ad-hoc queries

Examples:

relational databases (SQL)

One Size Does Not Fit All

Early expectations:
Each new data model was supposed to **replace** the previous one

What actually happened:
Every model introduced a **new capability**, not a replacement

Gartner: *"Big Data" is high volume, high velocity, and/or high variety information assets that require new forms of processing for enhanced decision making, insight discovery and process optimization.*



Social media and networks
(all of us are generating data)



Mobile devices
(tracking all objects)



Sensor networks
(measuring all kinds of data)

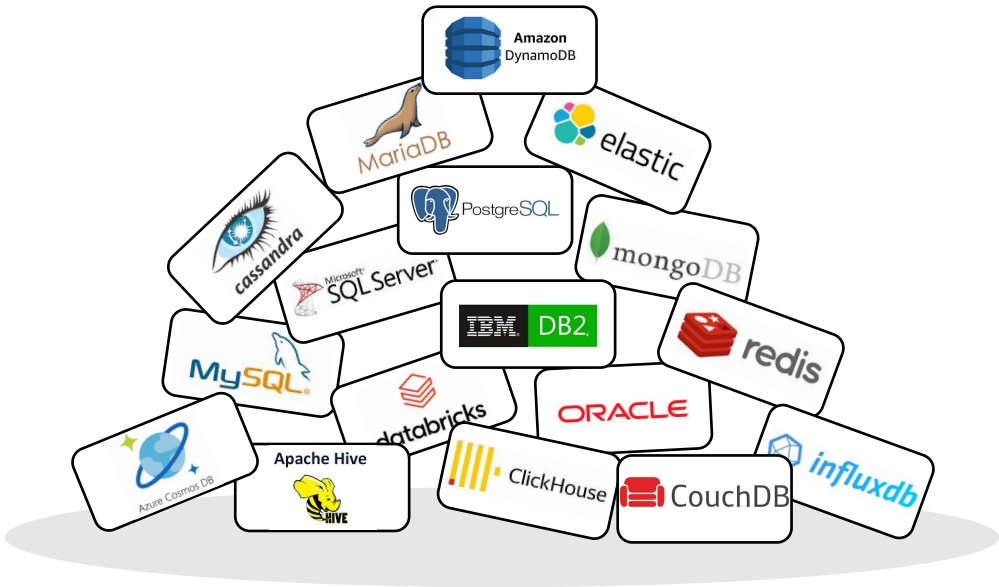


Scientific instruments
(collecting all sorts of data)

Today’s Database Systems

DB-Engines

- **427** database systems in total
- Almost **60%** of the most popular ones offer **multi-model features**



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

The End of Relational Databases?

- Relational databases are NOT going away
  - mature and reliable
  - rich ecosystem
  - strong for many workloads
- But they are now one option among many
  - No single database fits all applications
  - Polyglot persistence



Multi-model databases try to combine approaches.

References

- Ling Liu, M. Tamer Özsu: Encyclopedia of Database Systems. Springer 2009
- https://docs.oracle.com/cd/B19306_01/appdev.102/b14260/adobjxmp.htm
- http://nosql-database.org/
- Pramod J. Sadalage – Martin Fowler: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence
- Eric Redmond – Jim R. Wilson: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement
- Sherif Sakr – Eric Pardede: Graph Data Management: Techniques and Applications
- Shashank Tiwari: Professional NoSQL

Lecture 02

Distributed Storage & Data Processing

NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Big Data Ecosystem – Technology Overview

- Distributed storage
  - File systems - HDFS
  - Object storage - Amazon S3, MinIO
- Distributed processing
  - MapReduce
  - Apache Spark
- Distributed databases
  - NoSQL, multi-model, NewSQL, …
- Search & indexing
  - Elasticsearch

Distributed Storage

- Why distributed storage?
  - Data volumes exceed capacity of a single machine
  - Need for scalability and fault tolerance
  - Data generated continuously (logs, sensors, web, media)
  - Processing must run close to data
- Key idea: data stored across many machines and accessed as a single logical system
- Core design principles
  - Horizontal scaling on commodity hardware
  - Failure is normal ⟶ replication and redundancy
  - High-throughput access to large files
  - Parallel access from many nodes

Apache Hadoop



- What is Apache Hadoop?
  - Open-source framework for distributed storage and processing of large data
  - Designed to run on clusters of commodity hardware
  - Inspired by Google File System and MapReduce
- Why Hadoop emerged
  - Rapid growth of web and log data
  - Limits of single-machine databases
  - Need for scalable and fault-tolerant infrastructure
- Core idea: Store and process large datasets across many machines as one system

http://hadoop.apache.org/

Main Hadoop Components

- HDFS → distributed storage layer
- YARN → resource management and scheduling
- MapReduce → distributed batch processing engine
- Hadoop Common → shared libraries and utilities

First widely adopted open-source Big Data platform and foundation of modern distributed data systems

Hadoop Ecosystem (Selected Projects)

- Storage & serving
  - HBase — column-family distributed database
- SQL & analytics
  - Hive — SQL-like analytics over HDFS
- Coordination
  - ZooKeeper — distributed coordination service
- Serialization / data format
  - Avro — schema-based data serialization

Hadoop evolved into a large ecosystem of specialized tools for storage, processing and analytics

HDFS (Hadoop Distributed File System)

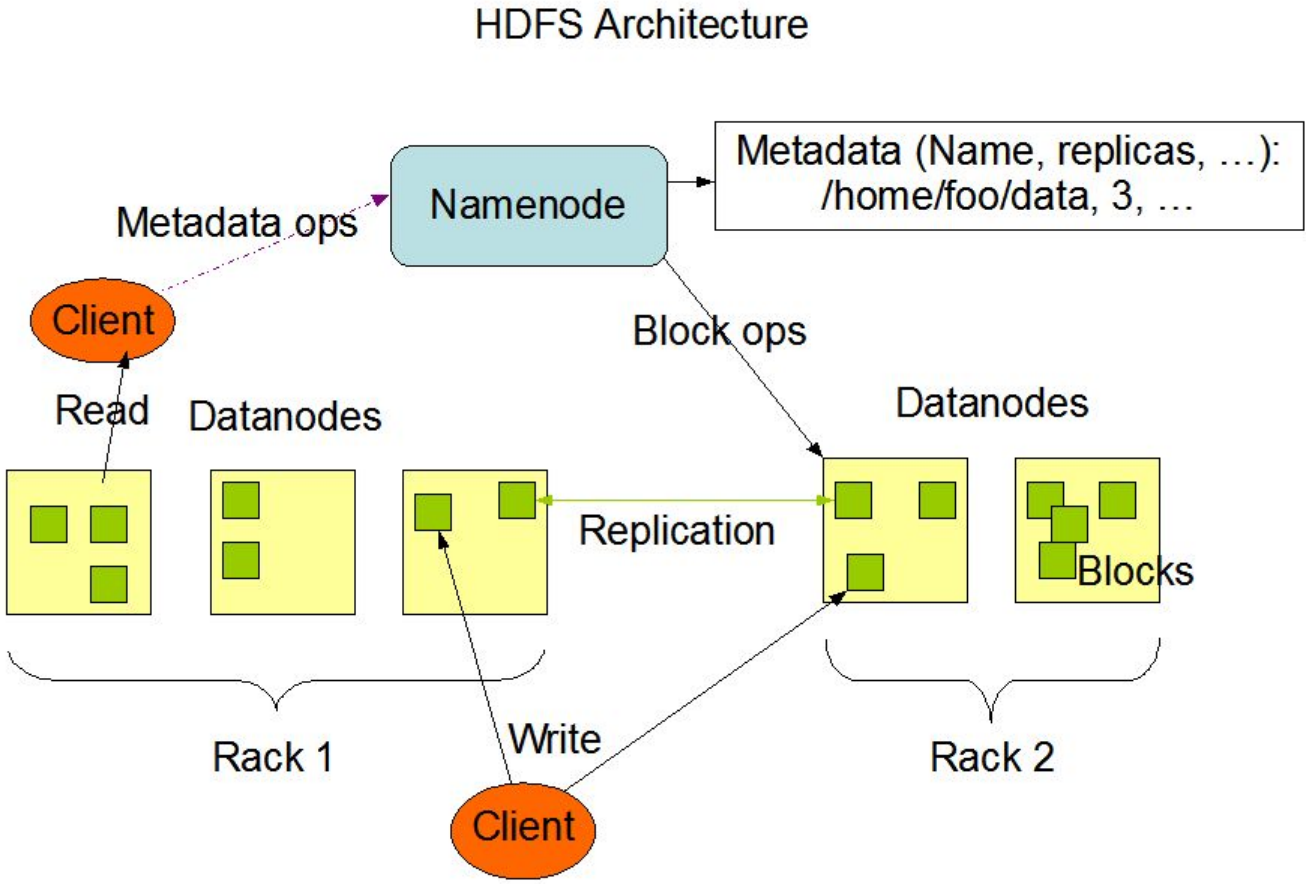


- Abstraction - distributed file system
  - Behaves like a standard file system
- Namespace
  - Hierarchical structure (directories + files)
  - Paths (e.g., `/data/logs/file.txt`)
- Typical operations
  - `mkdir, ls, put / get, mv, rm`
- Access methods
  - Command line (`hdfs dfs …` / `hadoop fs …`)
  - Java API (and other language bindings)
  - Web interface (for browsing & monitoring)

Same interface as a file system — different internal implementation

HDFS - Architecture

- Storage layer of the Hadoop ecosystem
- Master-slave model
  - NameNode ⟶ metadata management
  - DataNodes ⟶ store actual data blocks
- Data model
  - Files split into large blocks (128 MB+)
  - Blocks distributed across cluster nodes
  - Each block replicated (default: 3 copies)


HDFS Architecture
Namenode
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Client
Rack 2

HDFS - Fault Tolerance

- Design philosophy: “Failure is the norm, not the exception.”

- Why?
  - Clusters = thousands of machines
  - Hardware failures are statistically inevitable
- System assumption
  - At any time, some component is failing
- System response
  - Fault detection
  - Automatic re-replication
  - Background recovery

HDFS - Data Characteristics

- Access pattern
  - Streaming data access
  - High throughput preferred
- Update model
  - Files typically written once
  - Appends possible, random updates rare
- Typical usage
  - Log data
  - Web-scale datasets
  - Analytical processing inputs
- Implication: Optimized for large-scale analytics, not interactive data access

HDFS - How NameNode Works?

- Metadata in memory
  - NameNode stores file system metadata in memory
  - Namespace: files, directories, permissions
  - Mapping of files to blocks and DataNodes
- Persistent metadata on disk
  - **FsImage** → snapshot of file system metadata
  - **EditLog** → record of all metadata changes
- Startup & recovery
  - On startup: FsImage loaded into memory
  - EditLog replayed to reconstruct latest state
- Key idea: Separation of metadata snapshot and change log ensures reliability and recovery

HDFS - How DataNode Works?

- Storage role
  - Stores actual data blocks on local disks
  - Each block stored as a local file
  - One DataNode typically per cluster node
- Responsibilities
  - Serves read/write requests from clients
  - Creates, deletes and replicates blocks
  - Executes instructions from NameNode
- Cluster communication
  - Sends periodic **heartbeats** to NameNode
  - Reports stored blocks

Processing Distributed Data

- Now we can store massive data across many machines
  - But how can we process them?
- Challenges
  - Data distributed across cluster
    - Too large for a single machine
  - Moving data over network is expensive
  - Failures during computation are common
- Need: A programming model for distributed data processing

Google

MapReduce – Basic Idea

- A programming model and execution framework
- Origin: Google (Dean & Ghemawat, 2004)
  - Designed for processing web-scale data
  - Inspired Hadoop MapReduce and modern data engines
- Core principle: **divide and conquer**
  - Split data into parts
  - Process parts in parallel
  - Combine partial results
- Framework handles
  - Data distribution
  - Parallel execution
  - Fault tolerance
- Programmer defines only the computation logic

MapReduce – Functions Map and Reduce

- Function **Map** $(k_1, v_1) \rightarrow list(k_2, v_2)$
  - Processes input data
  - Produces intermediate key–value pairs
  - Runs in parallel on many nodes
- Function **Reduce** $(k_2, list(v_2)) \rightarrow (k_2, possibly\ smaller\ list(v_2))$
  - Receives all values with the same key
  - Aggregates or combines them
  - Produces final results

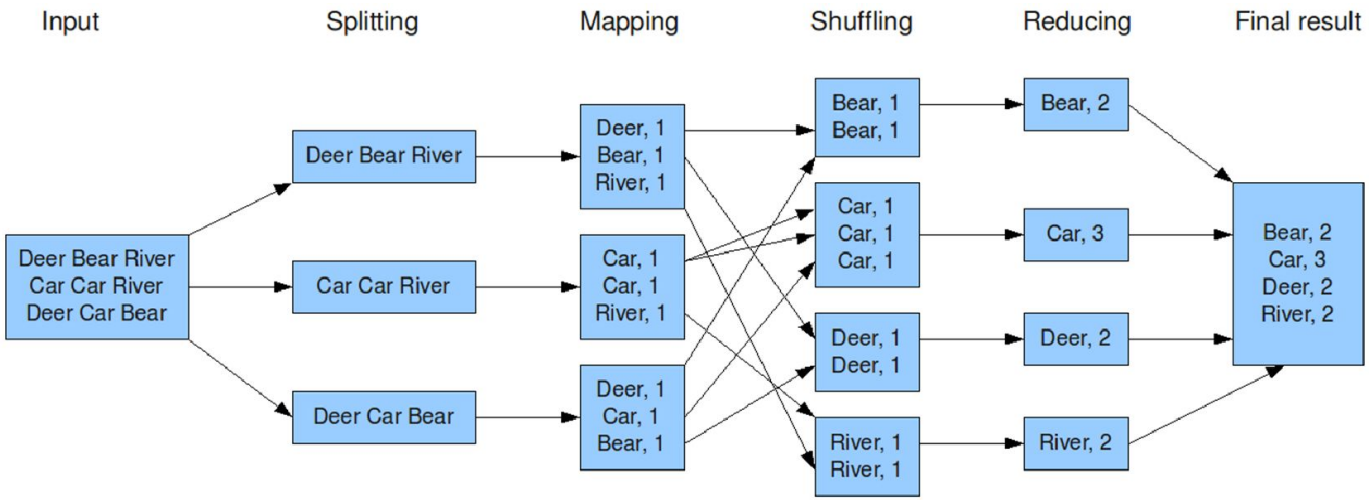
Key idea: Map transforms data ⟶ Reduce aggregates results

Example: Word Frequency

```
map(String key, String value):
  // key: document name
  // value: document contents
for each word w in value:
  EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
int result = 0;
for each v in values:
  result += ParseInt(v);
Emit(key, AsString(result));
```

Example: Word Frequency



Why MapReduce Was Revolutionary

Before MapReduce:

- Distributed computing was complex
- Manual parallelization
- Manual fault tolerance

MapReduce provided:

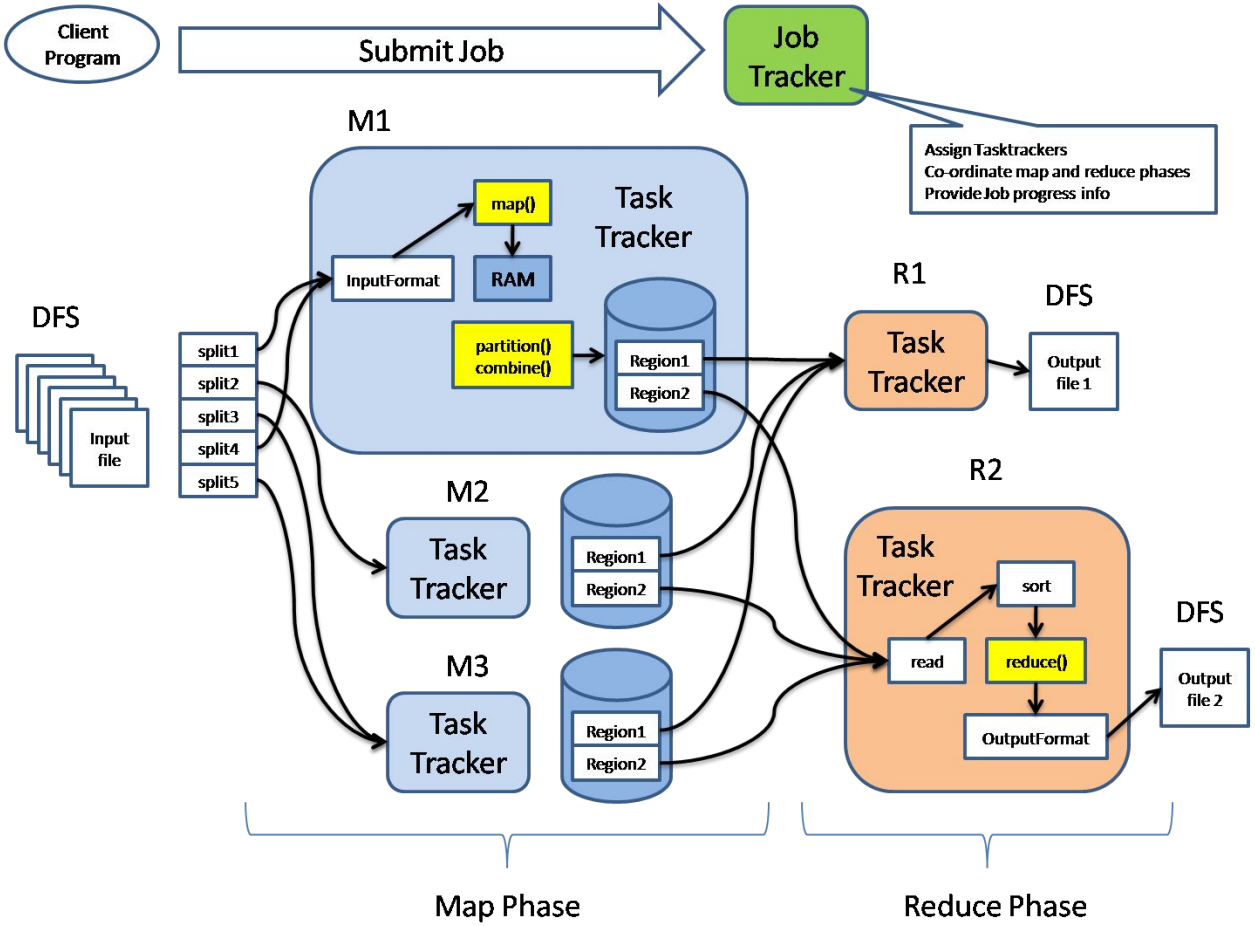
- Simple programming model
- Automatic parallelization
- Built-in fault tolerance
- Scalable processing on clusters

Hadoop MapReduce



- Open-source implementation of the MapReduce model within the Hadoop ecosystem
- Role
  - Distributed batch processing engine
  - Runs on top of HDFS
- Characteristics
  - Processes large datasets
  - Disk-based intermediate storage
  - Designed for high-throughput batch jobs
- JobTracker – master node
  - Schedules jobs and tasks, monitors execution
- TaskTracker – worker nodes
  - Execute map and reduce tasks, report status to JobTracker

Historical importance: First widely adopted Big Data processing framework


Client Program
Submit Job
Job Tracker
Assign Tasktrackers
Co-ordinate map and reduce phases
Provide Job progress info
M1
map()
Task Tracker
InputFormat
RAM
partition()
combine()
Region1
Region2
DFS
Input file
split1
split2
split3
split4
split5
R1
Task Tracker
DFS
Output file 1
M2
Task Tracker
Region1
Region2
R2
Task Tracker
sort
read
reduce()
OutputFormat
DFS
Output file 2
M3
Task Tracker
Region1
Region2
Map Phase
Reduce Phase

Hadoop MapReduce – Execution Roles

- **Mapper**
  - Processes input
  - Produces intermediate key–value pairs
  - Runs in parallel across cluster
- **Partitioner**
  - Determines which reducer receives each key
  - Ensures same keys go to the same reducer!
  - Default: hash(key)
- **Reducer**
  - Receives grouped values for each key
  - Aggregates or combines them
  - Produces final output
- Combiner (optional)
  - Local mini-reducer on mapper node
  - Reduces amount of data sent during shuffle
- Stragglers
  - Slow tasks that delay job completion
  - Re-execution of slow tasks

MapReduce – Criticism

- David DeWitt & Michael Stonebraker (2008)
  - Database community critique of MapReduce
- Main arguments
  - Step backwards from declarative query languages (SQL)
  - Lacks schema and data independence
  - Uses brute-force scans instead of indexes
  - Missing DBMS features
    - indexes
    - transactions
    - query optimization

Key message: MapReduce is powerful for large-scale batch processing, but not a replacement for database systems.

Note: Who is Michael Stonebraker (*1943)?



- Computer scientist and database researcher
  - Professor at UC Berkeley and MIT
- Turing Award recipient (2014)
  - “Nobel Prize of computing”
  - For concepts and practices underlying modern database systems
  - 2016 – Sir Tim Berners Lee
    - For inventing the WWW
- Major contributions
  - Ingres (relational DBMS)
  - Postgres ⟶ PostgreSQL
  - Vertica (column-oriented DBMS)

End of MapReduce?

- MapReduce was a breakthrough for large-scale distributed processing
- But… limitations emerged
  - High latency (disk-based processing)
  - Rigid two-stage model (map → reduce)
  - Inefficient for iterative algorithms
  - Poor support for interactive analytics
- Industry shift
  - Move toward faster, more flexible engines
  - In-memory processing
  - Real-time and interactive analytics

Result: New generation of distributed processing systems

From MapReduce to Apache Spark

- MapReduce enabled large-scale distributed processing
- But modern data workloads required more flexibility and speed
- New requirements
  - Faster processing (lower latency)
  - Interactive data analysis
  - Iterative algorithms (machine learning)
  - Unified support for batch and streaming

- Key idea: Keep data in memory and support general execution graphs

- Apache Spark – a unified engine for distributed data processing

Separation of Compute & Storage

- Traditional Hadoop model:
  - Storage (HDFS) and compute tightly coupled
  - Data stored locally on cluster nodes
    - We cannot scale storage and compute separately
- Modern architecture:
  - Storage independent of compute
  - Object storage (S3, MinIO)
  - Compute clusters scale independently
- Benefits
  - Elastic scaling
  - Better resource utilization
  - Cloud-native design
- Key idea: Compute and storage become separate layers

Data live in storage, compute comes and goes

From Distributed File Systems to Object Storage

- Separating compute and storage → a storage layer that is:
  - scalable
  - shared by many compute clusters
  - independent of specific machines
- Traditional HDFS:
  - tied to compute cluster
  - hard to scale independently
  - difficult to share across systems
- Modern solution: **object storage**
  - Shared persistent storage
  - Access via network API
  - Independent scaling
  - Cloud-native architecture
- Used by: Spark, analytics systems, machine learning, …

Amazon S3 Model



- S3 (Simple Storage Service) = scalable distributed storage layer
  - Cloud service
  - Not a database (no SQL, indexes, transactions. constraints…)
- Data model:
  - **Bucket** (container, namespace)
  - **Object** = key (like path inside a bucket) + data (bytes) + metadata (attributes)
  - **Logical prefixes** (like folders, but not real)
    - The storage is flat
- API operations:
  - `PUT, GET, LIST, DELETE`
- Objects are immutable
  - Updating = overwrite whole object
  - No in-place updates

https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

Data Layout in Object Storage



- Objects are flat → structure is logical (prefix-based)
- Example:
  - `data/raw/transactions.csv`
  - `data/processed/daily_revenue/`
  - `data/analytics/`
- Partitioning via prefixes:
  - `data/processed/region=EU/date=2024-01-01/`
- Key idea:
  - Layout = query optimization
  - No schema → design in paths

MinIO



- High-performance object storage compatible with the Amazon S3 API
  - Stores data as objects in buckets
  - Accessible over network via HTTP API
  - Can run locally or in cloud
- Why we use it?
  - Fully S3-compatible
  - Lightweight and easy to run
  - Suitable for experiments and development
  - Same API as real cloud object storage
- How it is accessed
  - Web interface (browser)
  - Command line client (`mc`)
  - Python (`boto3`)
  - Apache Spark (`s3a:// connector`)

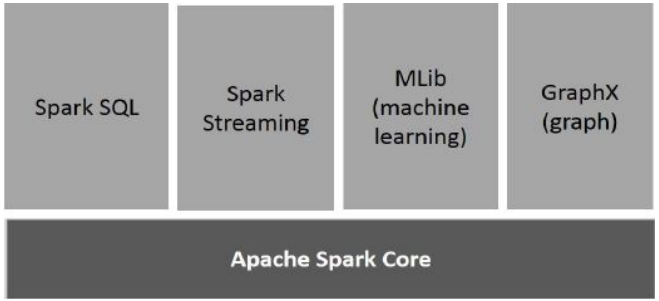
https://www.min.io/

MinIO - Basic Operations

- Upload object:
  `PUT data/raw/users.csv`
- Download object:
  `GET data/raw/users.csv`
- List objects:
  `LIST data/raw/`
- Delete object:
  `DELETE data/raw/users.csv`

Apache Spark (2014)

- Unified platform for large-scale data processing across distributed clusters
- Provides
  - Batch processing
  - SQL analytics (Spark SQL)
  - Machine learning (MLib)
  - Graph processing (GraphX)
  - Stream processing (Spark Streaming)

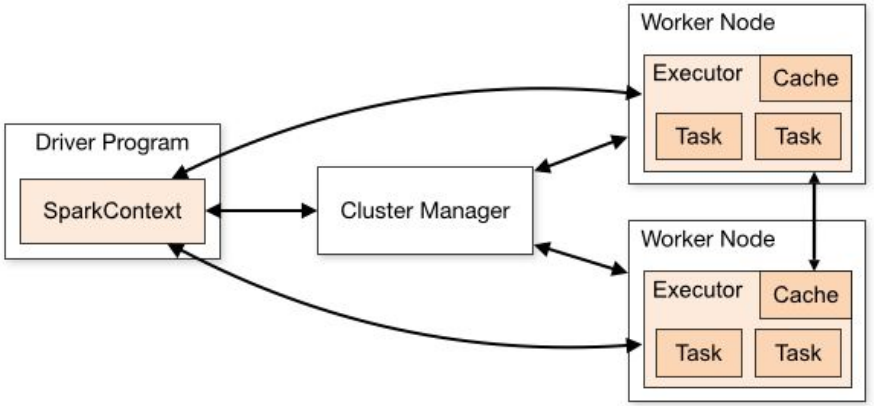


Key feature: General distributed execution engine for diverse data workloads

MapReduce vs Apache Spark

Aspect	MapReduce	Apache Spark
Processing model	Rigid map → reduce pipeline	General DAG execution engine
Intermediate data	Written to disk	In-memory when possible
Performance	Higher latency	Faster, low-latency processing
Workloads	Batch processing	Batch, SQL, streaming, ML, graph
Interactivity	Not interactive	Supports interactive analytics
Flexibility	Limited programming model	Flexible distributed computations
Typical role today	First-generation batch engine	Unified modern data processing engine

Spark Application



- Driver Program
  - Runs user's main function
  - Creates SparkSession / SparkContext
  - Builds execution plan
  - Sends tasks to cluster
- Executors (on worker nodes)
  - Run tasks in parallel
  - Store data in memory or disk
  - Return results to driver
- Cluster Manager
  - Allocates resources
  - Launches executors

Driver → Executors → Tasks → Results

Spark Application

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
.appName("MyApp") \
.getOrCreate()
df = spark.read.csv("data.csv", header=True)
```

- **Spark session**
  - Entry point to Spark functionality.
  - Creates connection to cluster and initializes execution environment.
- Typical steps
  - Create SparkSession
  - Load data
  - Apply transformations
  - Execute actions
- In practice
  - Most work done via DataFrame / Spark SQL API

Resilient Distributed Dataset (RDD)

- Basic distributed data structure in Spark
  - Collection of elements partitioned across cluster
  - Processed in parallel
- Properties
  - Immutable
  - Fault-tolerant
  - Can be cached in memory

- Role today
  - Low-level API
  - Foundation of Spark execution engine

DataFrame

- Distributed table with schema
- Similar to:
  - relational table
  - pandas DataFrame
- Features
  - Structured data
  - Optimized execution
  - SQL support

- Role today
  - Main abstraction in modern Spark

Operations

Transformations

- Create a new dataset from an existing one
  - Lazily
- Build execution plan
- Do NOT run immediately

Actions

- Trigger execution and return results.
- Execute the computation
- Return result to driver

```
df.select("name")
df.filter(df.age > 18)
df.groupBy("age").count()
```

```
df.show()
df.count()
df.collect()
```

Important: Nothing happens until an action is called

Transformations

- Basic transformations
  - **`map`**`()` – apply given function to each element
  - **`filter`**`()` – keep only selected elements
  - **`union`**`()` – combine given datasets
  - **`distinct`**`()` – remove duplicates
- Key-based transformations
  - **`reduceByKey`**`()` – aggregate values by key
  - **`groupByKey`**`()` – group values by key
  - **`sortByKey`**`()` – sort by key

Important: Executed only when an action is called

Actions

- Return values to the driver
  - `count()` – number of rows
  - `collect()` – return all data
  - `first()` / `take(n)` – preview data
- Display results
  - `show()` – print table
- Write output
  - `write()` – save to file (CSV, JSON, Parquet, …)

Important:

- Actions start execution
- All previous transformations are executed together
- Result is either returned or written

Execution Hierarchy

- Action starts a **job**
- Job is divided into **stages**
- Stages consist of parallel **tasks**
  - Shuffle usually creates stage boundaries

Distributed Output

- Spark writes datasets as directories of part files
- Not as one monolithic output file
- Reflects parallel execution
- Post-processing may be needed for single-file export

Shuffle Operations

- Some operations require data redistribution
- Examples
  - `groupBy()`
  - `join()`
  - `reduceByKey()`
  - `orderBy()`
- What happens?
  - Data move across executors
  - Network + disk I/O
  - Expensive operation

Key idea: Shuffle is the main cost in distributed data processing

Closures

```
threshold = 18
df.filter(df.age > threshold).show()
```

- When Spark executes operations on worker nodes, it sends the function and referenced variables
  - Each worker has its own copy
  - This is called a **closure**
- Changes inside tasks do NOT modify driver variables
  - Key idea: Code must be self-contained for distributed execution

Accumulators

- Variables used for aggregated updates from worker nodes to the driver
- Properties
  - Workers can update
  - Only driver can read final value
  - Used for counters, statistics, debugging

```
counter = spark.sparkContext.accumulator(0)

def f(row):
    global counter
    counter += 1

df.foreach(f)
```

Broadcast Variables

Without broadcast, every chef receives their own copy of the recipe book for each dish they prepare. With broadcast, each kitchen gets one shared recipe book that all chefs reuse.

- Normally:
  - Variables are sent to workers with each task
- With broadcast:
  - Data are sent once to each worker and reused by all tasks
- Used for large read-only data

```
rates = {"CZ": 25.0, "EUR": 1.0}
rates_b = spark.sparkContext.broadcast(rates)

def convert(row):
    return row.country, rates_b.value[row.country]

df.rdd.map(convert).collect()
```

Spark SQL

- Module for processing structured data in Spark.
  - Abstraction: **DataFrame** = distributed table with schema
- Provides
  - SQL queries
  - DataFrame API
  - Integration with Python
- Why important
  - Familiar relational model
  - Declarative queries
  - Scalable processing of large datasets

Key idea: Spark becomes a distributed SQL engine

File Formats for Analytics

- **CSV**:
  - row-based
  - human-readable
  - slow for analytics
  - Example: exported transactions, logs, data exchange
- **Parquet**:
  - columnar
  - compressed
  - efficient for aggregation
  - Example: daily revenue analysis, BI queries, data lake tables
- Key idea: Format affects performance

```
tr_id,timestamp,user_id,region,product_cat,amount
1,2024-01-01 10:00:00,1001,EU,books,25.0
2,2024-01-01 10:05:00,1002,US,electronics,199.0
3,2024-01-01 10:10:00,1003,EU,books,30.0
```

```
transaction_id: [1, 2, 3]
timestamp: [t1, t2, t3]
user_id: [1001, 1002, 1003]
region: [EU, US, EU]
category: [books, electronics, books]
amount: [25.0, 199.0, 30.0]
```

Partitioning by Column

- Large datasets are often stored in separate folders/prefixes
  - According to selected attribute values
- Example:
  - e.g., `region=EU/, region=US/, region=APAC/`
- Why?
  - Queries often filter by region or date
  - Spark can read only relevant parts
  - Less data scanned = faster analytics
- Trade-off: too many partitions may create too many small files
  - Spark can reduce output file count using `coalesce()`

Basic Examples

Typical workflow:

- Load data into DataFrame

```
df = spark.read.csv("people.csv", header=True)
df.show()
```

- Create temporary view

```
df.createOrReplaceTempView("people")
```

- Run SQL queries

```
r1 = spark.sql("SELECT name FROM people WHERE age > 18")
r2 = spark.sql("SELECT age, COUNT(*) FROM people GROUP BY age")
r3 = spark.sql("SELECT * FROM people ORDER BY age DESC")
```

- Work with results

```
r1.show()
r2.filter(r2.cnt > 10).show()
r3.show()
```

SQL and DataFrame API

- Two ways to work with structured data in Spark
- Both approaches produce the same execution plan
- When to use what?

SQL

- familiar relational queries
- joins, aggregations
- analysts & DB mindset

```
spark.sql(
"SELECT name, age "
"FROM people "
"WHERE age > 18"
).show()
```

DataFrame API

- programmatic pipelines
- integration with Python
- complex transformations

```
df.filter(df.age > 18) \
    .select("name", "age") \
    .show()
```

DataFrame API – Common Operations (PySpark)

- Selection & columns
  - **`select`**`()` – choose columns
  - **`withColumn`**`()` – create/modify column
  - **`drop`**`()` – remove column
- Filtering: **`filter`**`()` / **`where`**`()` – filter rows
- Aggregation
  - **`groupBy`**`()` – grouping
  - **`agg`**`()` – aggregate functions (count, avg, sum, …)
- Joins: **`join`**`()` – join DataFrames
- Sorting: **`orderBy`**`()` / **`sort`**`()` – sorting
- Output
  - **`show`**`()` – display results
  - **`write`**`()` – save data (CSV, Parquet, …)

https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/dataframe.html

Spark + Object Storage (MinIO)

Compute and storage are separated

- Spark does not require local storage.
  - Reads and writes data via storage connectors
- Architecture
  - **Compute layer:**
    - Apache Spark cluster
    - Executes distributed jobs
  - **Storage layer:**
    - MinIO (S3-compatible object storage)
    - Stores raw and processed data
- Connection:
  - `s3a:// protocol` (Hadoop S3 connector)

```
df = spark.read.csv("s3a://data/raw/users.csv",
     header=True
)
df.write.mode("overwrite").parquet("s3a://data/output/")
```

References

- Apache Hadoop: http://hadoop.apache.org/
- Hadoop: The Definitive Guide, by Tom White, 2nd edition, Oreilly's, 2010
- Hadoop Map/Reduce Tutorial
  - http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html
- David DeWitt and Michael Stonebraker: Relational Database Experts Jump The MapReduce Shark
- Spark Overview
  - https://spark.apache.org/docs/latest/index.html
- Apache Spark Examples
  - https://spark.apache.org/examples.html
- A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets
  - https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Lecture 03



In-Memory & Embedded Data Stores

NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

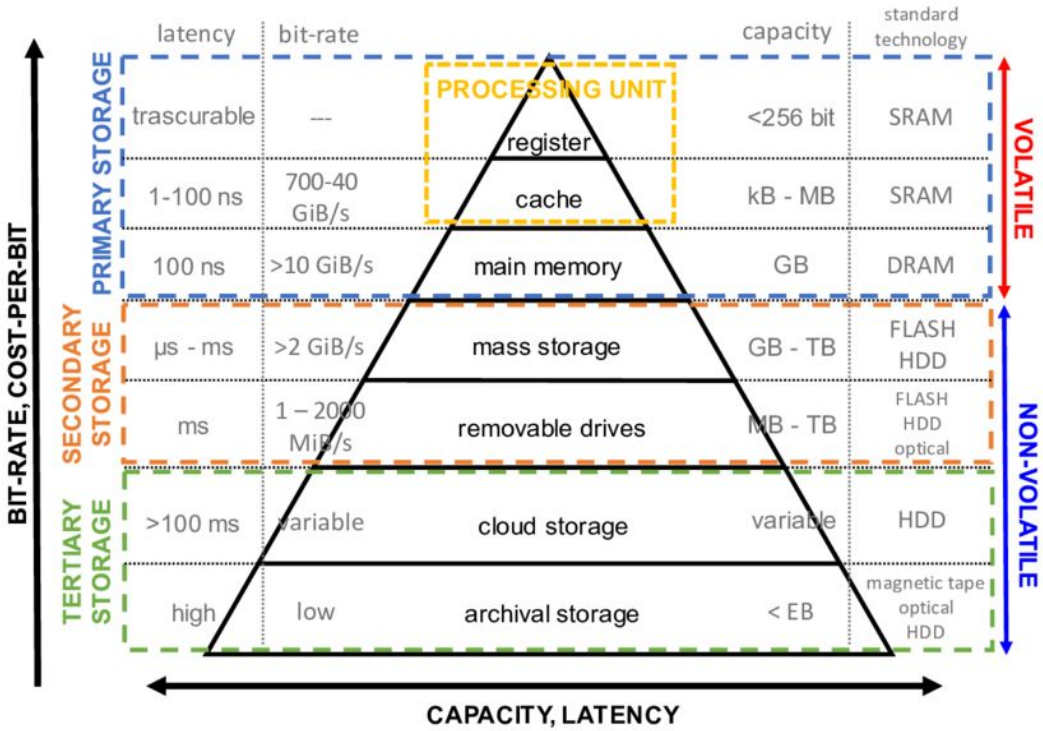
irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Why Memory Changes Architecture?

- The latency hierarchy:
  - L1 cache ~ 1 ns
  - RAM ~ 100 ns
  - SSD ~ 100 µs
  - HDD ~ 10 ms

Accessing disk instead of memory can be 100 000× slower



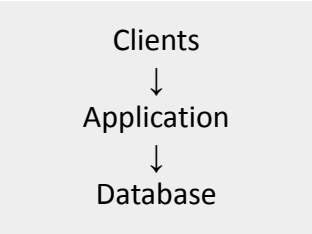
https://doi.org/10.13140/RG.2.2.20451.89124

Architectural Implication

- Data placement determines system behavior
  - Disk-based systems → I/O-bound
  - Memory-based systems → CPU-bound
  - Distributed systems → network-bound

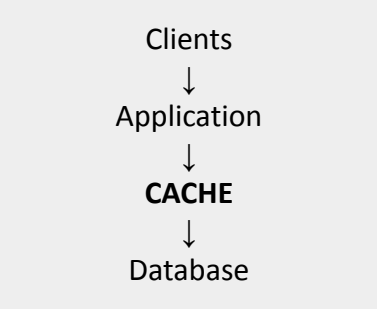
Why Databases Become Bottlenecks?

- Typical architecture



- Problem:
  - many repeated reads
  - database CPU becomes bottleneck
  - disk IO becomes bottleneck
- Solution idea: memory layer

- Caching layer



- Benefits:
  - faster reads
  - lower DB load
  - better scalability

Why Caching Works?

Clients
↓
Application
↓
CACHE
↓
Database

- Observations:
  - Most workloads have **temporal locality**
    - The recently used data will soon be used again
  - Most workloads access a small subset of data very frequently
- Example: product page, user profile, session data
- Idea: Store frequently accessed data in memory = caching

1. Read cache
2. If miss → read DB
3. Store in cache

```
Hit ratio = hits / (hits + misses)
```

- High hit ratio → effective cache
- Low hit ratio → wasted memory

Most workloads follow a Zipf-like distribution

Two Orthogonal Dimensions

Storage Model

Model	Description	Example
In-memory	data stored primarily in RAM	Redis
Disk-based	data stored on disk	PostgreSQL

Deployment Model

Model	Description	Example
Embedded	runs inside the application	SQLite
Server-based	separate database server	PostgreSQL

Some systems combine both properties

In-Memory Databases

- **Primary storage is RAM**
- Extremely low latency (microseconds)
- Often used as key-value stores
- Persistence is optional/asynchronous

Embedded Databases

- DBMS is **embedded in** the **application**
- No separate server process
- Application communicates via **direct library calls**

















In-Memory Databases

- Advantages
  - Very fast reads and writes
  - Suitable for real-time workloads
  - Efficient for high-throughput systems
- Disadvantages
  - RAM is expensive
  - Risk of data loss without persistence
  - Often limited dataset size
- Typical uses:
  - caching layers
  - session storage
  - streaming analytics
  - leaderboards / counters

Embedded Databases

- Advantages
  - Zero configuration
  - Very small footprint
  - No network overhead
  - Easy deployment
- Disadvantages
  - Limited concurrency
  - Usually single-node
  - Not ideal for distributed systems
- Typical uses:
  - mobile apps
  - desktop applications
  - IoT devices
  - local caches

In-memory describes where data is stored; embedded describes how the database is deployed

Embedded Databases in Practice

- 
  - Most widely deployed database
  - Single-file database
  - No server process
  - Runs directly inside the application
    - It is accessed through a library
- Typical environments: mobile apps, web browsers, desktop applications, IoT devices

Embedded databases run inside the application process

Redis



- Open-source database
  - First release 2009
- Written in C
  - Clients for many languages
- **Key-value store** with advanced data structures
  - Atomic operations
- Primary **in-memory dataset**
  - Persistence: dumping the dataset to disk periodically / appending each command to a log
- Typical roles:
  - **caching**, session storage, real-time analytics, …

http://redis.io/

Keys and Values



- Keys are binary safe = any binary sequence can be a key
- The stored value can be any object
  - strings, hashes, lists, sets and sorted sets
- Can do range, diff, union, intersection, … operations
  - **Atomic** operations
  - Not usual, not required for key-value stores

Strings



- Binary safe = any binary sequence
  - e.g., a JPEG image
- Max length: 512 MB
- Typical use: counters, cached objects, session tokens
- Operations:
  - Set/get the string value of a key: GET/SET, SETNX (set if not set yet)
  - String-operation: APPEND, STRLEN, GETRANGE (get a substring), SETRANGE (change a substring)
  - Integer-operation: INCR, INCRBY, DECR, DECRBY
    - When the stored value can be interpreted as an integer
  - Bit-operation: GETBIT, BITCOUNT, SETBIT

Strings - Examples




```
> SET count 10
OK
> GET count
"10"
> INCR count
(integer) 11
> DECRBY count 10
(integer) 1
> DEL count
(integer) 1          // returns the number of keys removed
```

Typical use: counters and cached values

Hash

- A structured object stored under a single key
  - Multiple field-value pairs
  - Max number of field-value pairs: $2^{32} - 1$
- Example: a user with fields name, surname, age, …
- Operations:
  - HSET key field value (set a value to the field of a specified key), HMSET (set multiple fields)
  - HGET (get the value of a hash field), HMGET, HGETALL (get all fields and values in a hash)
  - HKEYS (get all fields), HVALS (get all values)
  - HDEL (delete one or more hash fields), HEXISTS, HLEN (number of fields in a hash)

Hash - Examples

```
> HSET users:sara id 3
(integer) 1
> HGET users:sara id
"3"
> HMSET users:sara login sara group students
OK
> HMGET users:sara login id
1) "sara"
2) "3"
> HDEL users:sara group
(integer) 1
> HGETALL users:sara
1) "id"
2) "3"
3) "login"
4) "sara"
```

Typical use: storing objects or user profiles

List



- Lists of strings, sorted by insertion order
- Possible to push new elements on the head (on the left) or on the tail (on the right)
- A key is removed from the key space if a list operation will empty the list (= value for the key)
- Max length: $2^{32} - 1$ elements
  - 4,294,967,295 = more than 4 billion of elements per list
- Accessing elements
  - Very fast near the extremes of the list (head, tail)
  - Slow accessing the middle of a very big list
    - O(N) operation
- Typical use: queues, recent activity

List - Operations



- Add element(s) to the list:
  - LPUSH (to the head)
  - RPUSH (to the tail)
  - LINSERT (inserts before or after a specified element)
  - LPUSHX (push only if the list exists, do not create if not)
- Remove element(s): LPOP, RPOP, LREM (remove elements specified by a value)
- LRANGE (get a range of elements), LLEN (get length), LINDEX (get an element at index)
- BLPOP, BRPOP remove an element or block until one is available
  - Blocking version of LPOP/RPOP

List - Examples




```
> LPUSH animals dog
(integer) 1       // number of elements in the list
> LPUSH animals cat
(integer) 2
> RPUSH animals horse
(integer) 3
> LRANGE animals 0 -1 // -1 = the end
1) "cat"
2) "dog"
3) "horse"
> RPOP animals
"horse"
> LLEN animals
(integer) 2
```

Example: queue-like behavior

Set

Redis

- Unordered collection of **non-repeating** strings
- Possible to add, remove, and test for existence of members in O(1)
- Max number of members: $2^{32} - 1$
- Operations:
  - Add element: SADD, remove element: SREM
  - Classical set operations: SISMEMBER, SDIFF, SUNION, SINTER
  - The result of a set operation can be stored at a specified key (SDIFFSTORE, SINTERSTORE, ...)
  - SCARD (element count), SMEMBER (get all elements)
  - Operations with a random element: SPOP (remove and return random element), SRANDMEMBER (get a random element)
  - SMOVE (move element from one set to another)

Set - Examples




```
> SADD friends:Lisa Anna
(integer) 1
> SADD friends:Dora Anna Lisa
(integer) 2
> SINTER friends:Lisa friends:Dora
1) "Anna"
> SUNION friends:Lisa friends:Dora
1) "Lisa"
2) "Anna"
> SISMEMBER friends:Lisa Dora
(integer) 0
> SREM friends:Dora Lisa
(integer) 1
```

Example: social graph operations

Sorted Set

- Non-repeating collection of strings
  - Every member is associated with a **score**
    - Used in order to make the set ordered
  - From the smallest to the greatest
    - May have repeated values
- Then lexicographical order
- Possible to add, remove, or update elements in O(log N)
- Typical use: leaderboards, ranking
- Operations:
  - Add element(s): ZADD, remove element(s): ZREM, increment the score of a member: ZINCRBY
  - Number of elements in a set: ZCARD
  - Elements with a score in a specified range: ZCOUNT (count), ZRANGEBYSCORE (get the elements)
  - Set operations (store result at a specified key): ZINTERSTORE, ZUNIONSTORE , …

Sorted Set - Example

```
> ZADD articles 1 Anna 2 John 5 Tom
(integer 3)
> ZCARD articles
(integer) 3
> ZCOUNT articles 3 10 // members with score 3-10
(integer) 1
> ZINCRBY articles 1 John
"3"              // returns new John's score
> ZRANGE articles 0 -1 // outputs all members
1) "Anna"     // sorted according score
2) "John"
3) "Tom"
```

Typical use: leaderboards

Redis as Data Structure Server



type	use case
string	cache objects
list	ordered data
set	membership, tags
sorted set	ranking
hash	structured objects

Redis can be used not only as a cache, but also as a data layer for various applications.

Time to Live



- Allows Redis keys to expire automatically
- Important
  - TTL is defined on keys
  - Expiration removes the entire object

```
> SET cookie:google hello
OK
> EXPIRE cookie:google 30
(integer) 1
> TTL cookie:google
(integer) 23
> GET cookie:google
"hello"
> TTL cookie:google
(integer) -1
> GET cookie:google
(nil)
```

TTL allows cached data to expire automatically

Caching vs Persistence

- Cache
  - session data
  - API responses
  - recent objects
- Persistent storage
  - orders
  - payments
  - user accounts
- Important:
  - cache → performance
  - database → durability

Caching	Persistence
temporary data	durable data
improve performance	ensure durability
data may disappear	data must survive failures

Performance-Driven Design

- Modern systems often redesign data structures for performance
  - Trade consistency and normalization for performance

Pattern	Idea
Caching	store frequently accessed data in memory
Denormalization	duplicate data to reduce joins
Precomputation	compute results in advance
Materialized views	store query results

These techniques are widely used in large-scale web systems

Caching Patterns

- Different strategies for interaction between cache and database

Pattern	Idea
cache-aside	application loads cache on miss
write-through	writes go through cache
write-back	cache writes to DB asynchronously



Cache-Aside (Lazy loading)

- The application loads data into the cache only when needed
- Flow:
  - Cache hit: App → Redis → data
  - Cache miss: App → Redis (miss)
    → Database → Redis store
- Characteristics
  - Database is the source of truth
  - Cache populated on demand
    - First request = cache miss
    - Later requests = cache hit
- Typical use:
  - Product pages, user profiles, API responses
- Most popular

```
value = cache.get(key)

if value == null:
     value = database.load(key)
     cache.set(key, value)

return value
```

Write-Through Cache

- All writes go through the cache before reaching the database
- Characteristics
  - Cache always contains fresh data
  - Simpler reads
  - Writes are slower (two writes)
- Typical use
  - Systems requiring strong consistency
  - Financial or transactional systems

```
cache.write(key, value)
database.write(key, value)
```

```
value = cache.get(key)
```

Ensures cache and database remain consistent

Write-Back Cache (Write-Behind)

- Writes go only to cache, database is updated later asynchronously
- Characteristics
  - Very fast writes
  - Database load reduced
  - Risk of data loss if cache fails
- Typical use
  - High-throughput systems
  - Analytics pipelines
  - Buffering writes

```
cache.write(key, value)
async database.write(key, value)
```

```
value = cache.get(key)
```

Risk of data loss if cache fails before data is persisted

Cache Invalidation (Stale Data Problem)

- Problem: Cache may return outdated (stale) data
- Two strategies:
  - TTL (time-based expiration)
    - Simple
    - Data may be stale until expiration
  - Explicit invalidation
    - On update: DELETE cache key
    - Stronger consistency
- Pattern: UPDATE database → DEL cache key

Cache stores only part of the data…

- Memory is limited
- Example:
  - Dataset: 2 TB
  - RAM available for cache: 64 GB
- ⟶ only a small fraction of data fits into memory
- Typical observation: A small subset of data generates most requests
- Caching focuses on hot data
  - hot data ⟶ frequently accessed
  - cold data ⟶ rarely accessed
- Goal: keep frequently accessed data in memory

Caching relies on skewed access patterns

What happens when cache is full?

- The system must remove some items = **cache eviction**
- Common eviction policies:
  - LRU = remove least recently used item
  - LFU = remove least frequently used item
  - FIFO = remove oldest item
- Example:

```
Cache size = 3

A B C
access A
insert D

Evicted item → B or C (least recently used)
```

Eviction policies approximate which data will be used again

Cache vs Redis

- Cache = architectural concept
  - Temporary storage for frequently accessed data
  - Reduces database load
  - Improves latency
- Examples of cache systems:
  - Redis
  - Memcached
  - in-process cache (application memory)
- Important: Cache is a role in the architecture, not a specific technology

Cache is an architectural role
Redis is one possible implementation

Redis as a Cache



- Redis is commonly used as a **cache implementation**
- Why?
  - In-memory storage → very low latency
  - Simple key-value access
  - Supports TTL (automatic expiration)
  - Scalable and distributed
- Redis stores:
  - Frequently accessed objects
  - Session data
  - Temporary results
  - Counters / leaderboards
- Database (e.g., PostgreSQL) remains the source of truth
  - Cache can be deleted anytime
  - Data must always exist in the database

Cache key design

- Keys usually encode the object type
- Key design principles:
  - Encode query in key
    - e.g. `product:1, product:1:json, product:1:keywords`
    - Key = materialized query
  - Consistent naming: `entity:id[:view]`
  - Avoid:
    - High variability keys
    - Unbounded key space
- Benefits:
  - Easy lookup
  - Avoids collisions
  - Easier cache invalidation

Use consistent naming conventions for cache keys

Anti-pattern: Key Explosion



- Problem:
  - Too many unique keys
  - Low reuse ⟶ low hit ratio
- Example:
  - `search:price_gt:101`
  - `search:price_gt:102`
  - `search:price_gt:103`
- ⟶ Almost no cache hits
- Solutions:
  - Do not cache high-variability queries
  - USe Bucket values
    - e.g., `0-100, 100-200`

References

- Eric Redmond – Jim R. Wilson: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement
- Pramod J. Sadalage – Martin Fowler: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence
- Karl Seguin: The Little Redis Book http://openmymind.net/2012/1/23/The-Little-Redis-Book/
- https://redis.io/

Lecture 03

Introduction to NoSQL Data Stores

NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Modern Data Landscape

- Modern applications work with:
  - massive volumes of data
  - distributed systems
  - real-time processing
  - flexible schemas

- Traditional relational databases are not always the best fit

  → New database architectures emerged

From SQL to NoSQL

Traditional databases:

- relational model
- fixed schema
- ACID transactions
- strong consistency

NoSQL databases:

- flexible data models
- horizontal scalability
- distributed architectures
- often relaxed consistency

Why NoSQL Appeared

- Main motivations:
  - web-scale systems
  - large datasets
  - distributed architectures
  - performance and scalability
- Examples:
  - Google
  - Amazon
  - Facebook
- These companies needed databases that scale across thousands of machines

NoSQL Data Model - Aggregates

- Relational model
  - Data normalized
  - Many tables
  - JOIN to reconstruct objects
- NoSQL model
  - Data stored together
  - One aggregate = one unit
  - No joins required

Aggregate = data that is stored and accessed together

Relational Model: Normalized Data

Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

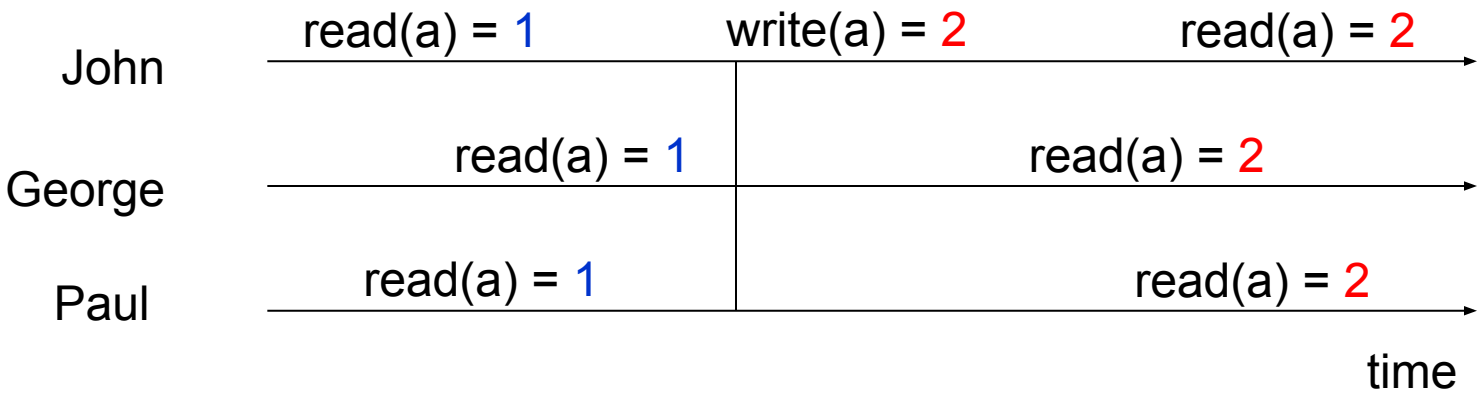
Aggregate Model (NoSQL)

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
  }]
}
}
```

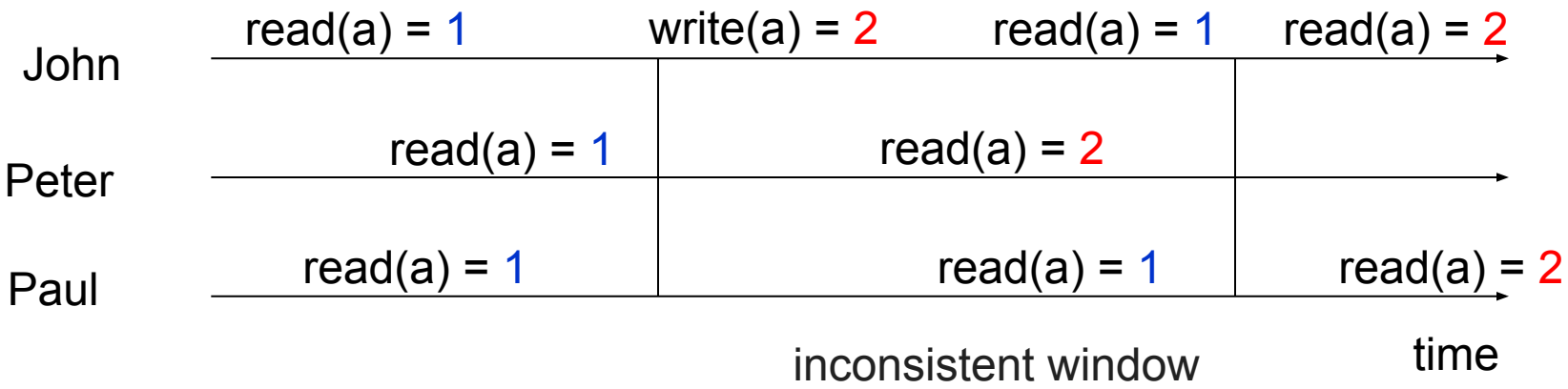
Trade-Offs

- ACID - too strict, too inefficient in a distributed system
  - Atomicity, Consistency, Isolation, Durability
- BASE
  - Basically Available
    - The system works basically all the time
    - Partial failures can occur, but without total system failure
  - Soft state
    - The system is in flux and non-deterministic
    - Changes occur all the time
  - Eventual consistency
    - The system will be in some consistent state at some time in future

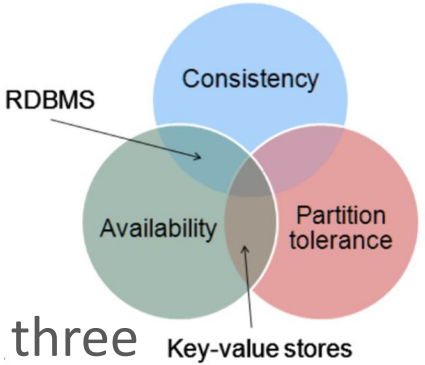
Strong Consistency



Eventual Consistency



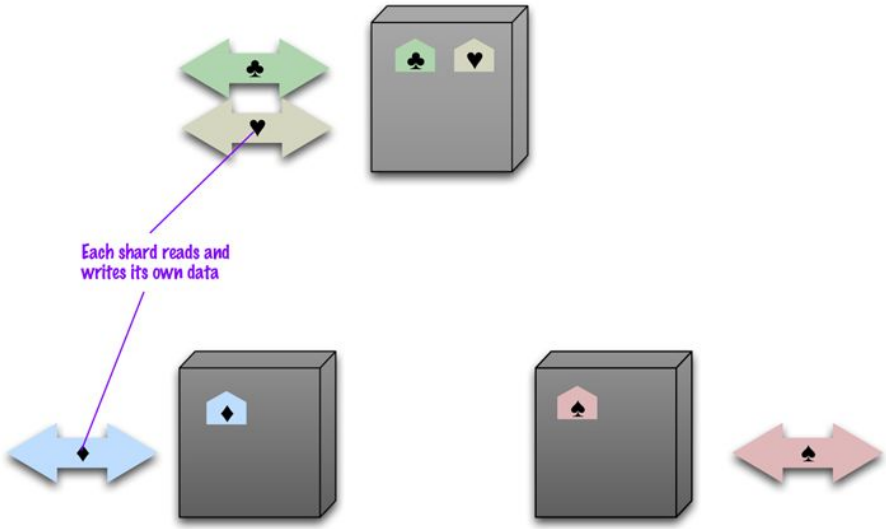
CAP Theorem (Brewer's theorem)



- In a distributed system it is impossible to guarantee all three properties simultaneously:
  - Consistency (C) - all nodes see the same data at the same time
  - Availability (A) - every request receives a response
  - Partition tolerance (P) - the system continues working despite network failures
- During a network partition a system must choose between Consistency OR Availability
  - Partition tolerance is naturally required in distributed systems

Sharding

- Sharding = splitting data across multiple machines
- Benefits:
  - horizontal scalability
  - increased storage capacity
  - parallel query processing
- Challenges:
  - data distribution strategy
  - cross-shard queries
  - rebalancing when nodes are added
- Common techniques:
  - range partitioning
  - hash partitioning

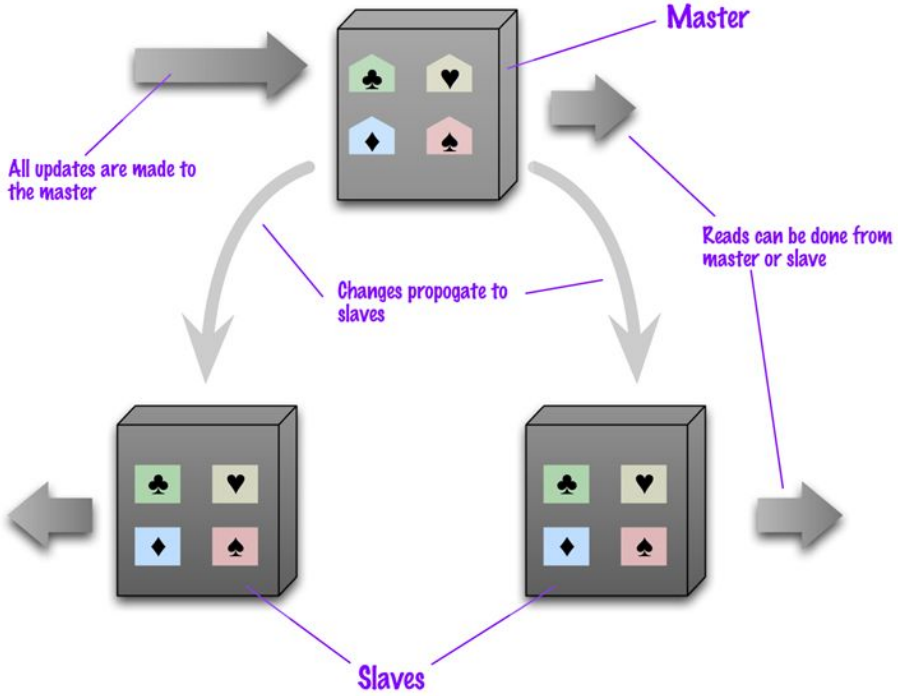


Replication

- Replication = storing multiple copies of the same data
- Benefits:
  - fault tolerance
  - high availability
  - faster reads
- Challenges:
  - keeping replicas consistent
  - conflict resolution
  - replication lag
- Replication factor: N = number of copies
  - Common default: N = 3

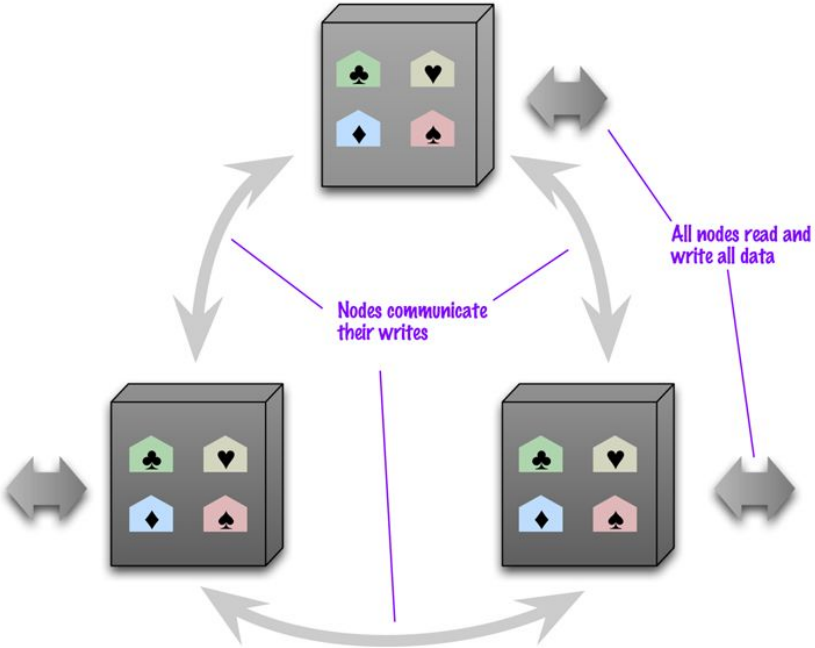
Master-Slave Replication

- One node acts as the master.
  - accepts all write operations
  - propagates changes to replicas
- Replicas (slaves):
  - receive updates from the master
  - often serve read requests
- Advantages
  - simple design
  - easy consistency control
- Limitations
  - master is a potential bottleneck
  - master failure requires failover

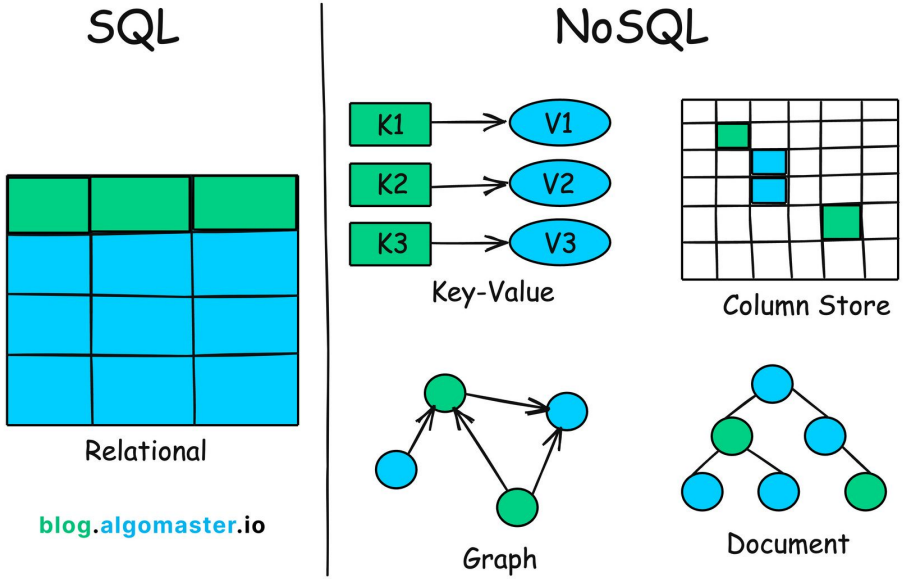


Peer-to-Peer Replication

- All nodes are equal
  - any node can accept reads and writes
  - data is replicated across nodes
  - no single master
- Characteristics
  - no single point of failure
  - high availability
  - distributed coordination required
- Challenges
  - conflict resolution
  - eventual consistency
  - more complex protocols



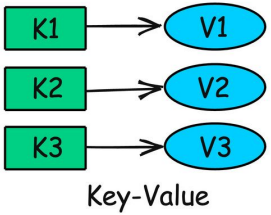
Types of NoSQL Databases



Type	Example
Key-value	Redis riak
Document	mongoDB.
Column-family	Apache CASSANDRA
Graph	neo4j

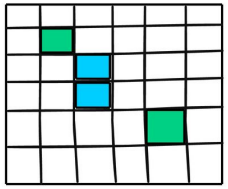
These systems differ mainly in how data is modeled and accessed

Key-Value Systems



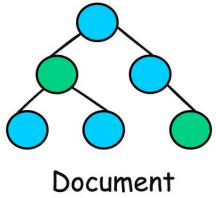
- The simplest NoSQL data stores
- Conceptually similar to a hash table
- Like a table in RDBMS with two columns, such as ID and VALUE
  - **ID** column being the key
  - **VALUE** column storing the value
    - A BLOB that the data store just stores
- Basic operations:
  - **Get** the value for the key
  - **Put** a value for a key
  - **Delete** a key from the data store
- Simple = great performance, easily scaled
- Simple = not for complex queries, aggregation needs

Column-Family Systems



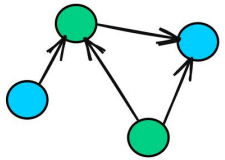
- Data organized into rows and column families
  - Inspired by Google BigTable
- A row key identifies a record
- Columns grouped into column families
  - **Columns within a row can differ**
  - New columns can be added dynamically
- Data often stored with timestamps
- Basic operations:
  - Read a row by key
  - Insert or update columns
  - Delete columns or rows

Document Systems



- **Store data as documents** (JSON, BSON, XML)
  - Documents are self-describing hierarchical structures
  - Documents are typically grouped into collections
  - Natural representation of application objects
- Flexible schema - fields may differ between documents
- Basic operations:
  - Insert/delete a document
  - Retrieve documents by query
  - Update document fields
- **Rich query capabilities**
  - Complex joins are not typical

Graph Systems



- Designed to **store entities and relationships**
- Data represented as:
  - Nodes (entities)
  - Edges (relationships)
  - Properties (attributes)
- Optimized for graph traversal
  - Find neighbors of a node, traverse relationships, search paths between nodes
- Efficient for highly connected data
  - Natural modeling of relationships
  - Powerful graph queries
- Distribution of large graphs is difficult

Schemaless Data Model

- Many NoSQL databases support flexible or dynamic schemas
  - Records in the same collection do not have to share identical structure
- Advantages
  - flexible data model
  - easier evolution of applications
  - suitable for rapidly changing data
- Challenges
  - weaker data validation
  - application must handle data structure
  - harder to enforce consistency

Polyglot Persistence

- Modern applications often use multiple database technologies
  - Different systems are optimized for different workloads
- Examples
  - Relational databases → transactions, structured data
  - Key-value stores → caching, fast lookup
  - Document databases → flexible data structures
  - Graph databases → relationship-heavy data

Instead of one universal database, choose the right database for the right task

References

- http://nosql-database.org/
- Pramod J. Sadalage – Martin Fowler: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence
- Eric Redmond – Jim R. Wilson: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement
- Sherif Sakr – Eric Pardede: Graph Data Management: Techniques and Applications
- Shashank Tiwari: Professional NoSQL

Lecture 03



Key-Value Stores

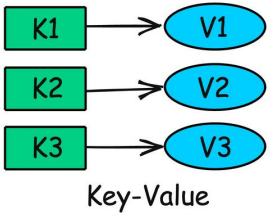
NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Key-Value Stores



- The simplest NoSQL data stores
- Conceptually similar to a hash table
- Like a table in RDBMS with two columns, such as ID and VALUE
  - **ID** column being the key (typically string or numeric)
  - **VALUE** column storing the value
    - A BLOB that the data just stores (text, JSON, binary data, image, session state, ...)
- Basic operations:
  - `Get(key)` the value for the key
  - `Put(key,value)` a value for a key
  - `Delete(key)` a key from the data store
- Simple = great performance, easily scaled
- Simple = not for complex queries, aggregation needs

RocksDB

riak

Redis

MEMCACHED

Suitable Use Cases

- Caching
  - result cache, page cache, object cache
- Session management
  - web sessions, authentication state
- User preferences / profiles
  - simple per-user settings
- Shopping carts
  - key = user/cart id
- Metadata / lookup tables
  - fast access by identifier
- Temporary state

Querying and Key Design

- We primarily query by the key
- If the key is not known:
  - some systems allow listing all keys (expensive)
  - some systems support search over values via an external index (the data must be indexed first - e.g., Riak Search)
- How to design the key?
  - generated automatically
  - provided by the application or user
  - derived from timestamps or other data
- Typical use cases: session data (session ID as key), shopping carts (user ID), user profiles (user ID)
- Some systems support key expiration (TTL)

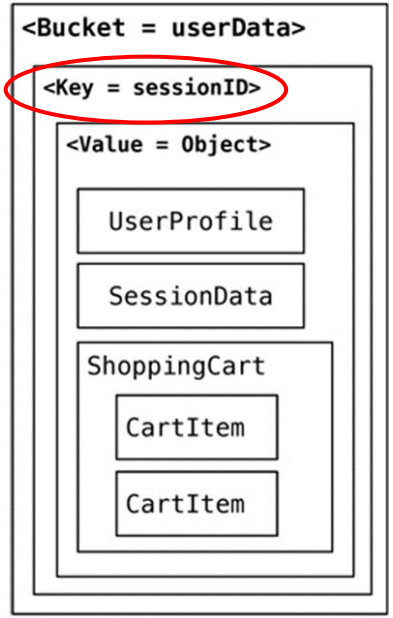
Riak



- A distributed key-value store
- Designed for:
  - scalability
  - fault tolerance
  - high availability
- Suitable for workloads with simple access by key
  - Built-in MapReduce support
- Riak extends the basic key-value model with:
  - replication
  - partitioning
  - quorum-based reads and writes
  - conflict resolution

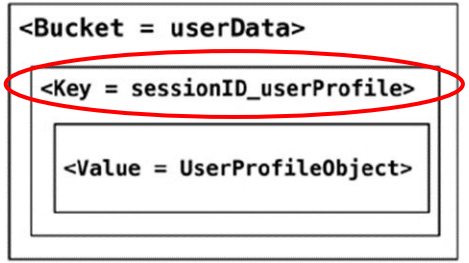
Riak Data Model

- **Key** identifies one object within the **bucket**
- **Value** stores the actual data
  - text, JSON, binary object
- Buckets may also define shared properties
  - e.g., replication or conflict-handling settings



riak

Oracle	Riak
database instance	Riak cluster
table	bucket
row	key-value
rowid	key



Working with Riak over HTTP



- Riak provides an HTTP API
- Objects are addressed by bucket + key
- Typical operations:
  - `GET` – read an object
  - `PUT` – create or replace an object
  - `DELETE` – remove an object
- Values are content-agnostic, but a content type should be specified
  - e.g. `application/json`

```
# PUT
curl -X PUT \
  -H "Content-Type: application/json" \
  -d '{"name":"Alice","city":"Prague"}' \
  http://localhost:8098/buckets/users/keys/alice

# GET
curl -i \
  http://localhost:8098/buckets/users/keys/alice

# DELETE
curl -X DELETE \
  http://localhost:8098/buckets/users/keys/alice
```

Riak Links



- Riak can store **links between objects**
- A link connects one object to another object
  - Identified by bucket + key
- Links can express simple relationships between related data
  - e.g., user → shopping cart, order → customer
- Useful for navigation between objects
- But links are not a replacement for relational joins
  - Limited querying capabilities
  - Application logic is still often needed

Object: users/alice
Links:
→ carts/alice
→ profiles/alice

Links add navigation, but not rich query semantics

Creating Links

```
curl -X PUT \
  -H "Content-Type: application/json" \
  -d '{"name":"Alice","email":"alice@example.com"}' \
  http://localhost:8098/buckets/users/keys/alice

curl -X PUT \
  -H "Content-Type: application/json" \
  -d '{"items":["book","pen"],"total":350}' \
  http://localhost:8098/buckets/carts/keys/alice

curl -X PUT \
  -H "Content-Type: application/json" \
  -H 'Link: </buckets/carts/keys/alice>; riaktag="cart"' \
  -d '{"name":"Alice","email":"alice@example.com"}' \
  http://localhost:8098/buckets/users/keys/alice
```

- Create user
- Create cart
- Add link (with tag “cart”) from user to cart

Link Walking

```
curl -i \
  http://localhost:8098/buckets/users/keys/alice

curl -i \
  http://localhost:8098/buckets/users/keys/alice/carts,cart,1
```

/buckets/<bucket>/keys/<key>/<target_bucket>,<tag>,<keep>

- Classical GET user
- GET cart of user accessed via link (with tag “cart”)
- The result is object `carts/alice`

Riak Search



- Riak Search adds search over **indexed content**
- It supports querying by fields inside stored values
  - Not only by bucket + key
- Riak Search 2.0 is based on Solr integration
  - Distributed full-text engine
  - Requires the data to be indexed first
- Support for various mime types
  - JSON, XML, plain text, …
- Exact match queries, scoring and ranking for most relevant results

Riak Search - Examples



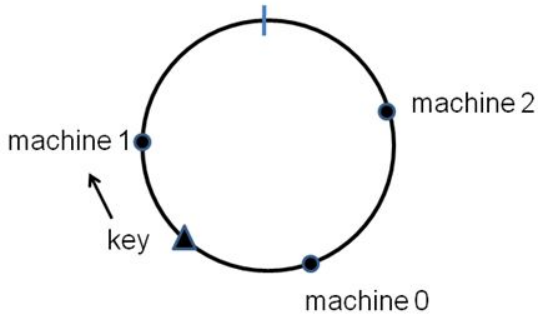
- Exact match: `city_s:Prague`
- Boolean queries: `city_s:Prague AND age_i:[25 TO *]`
- Wildcard / prefix: `name_s:A*`
- Range queries: `age_i:[20 TO 30]`
- Full-text search: `description_t:"wireless headphones"`

More on Internals of riak

Consistent Hashing

- Minimizes reshuffling of keys when a hash-table data structure is rebalanced
  - Slots are added/removed
- Hash function maps the keys to a circle
  - Each node in the cluster is responsible for an interval of hashes (slot) in the circle
  - Only `k/n` keys need to be remapped on average
    - `k` = number of keys
    - `n` = number of intervals (slots)

  instead of almost all in most other hashing types

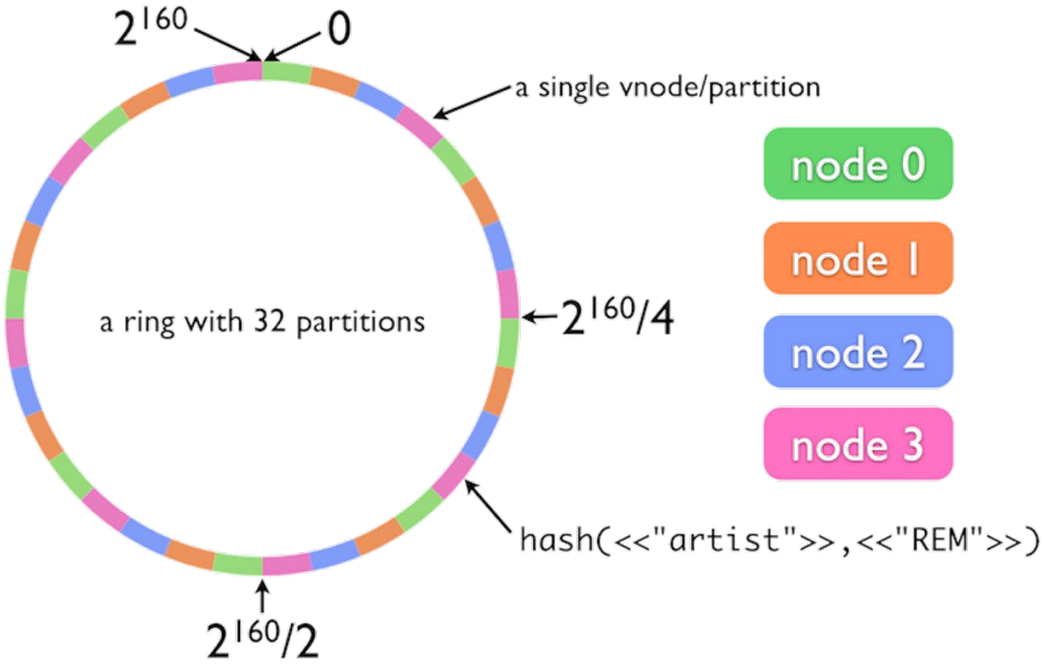


Sharding and Replication



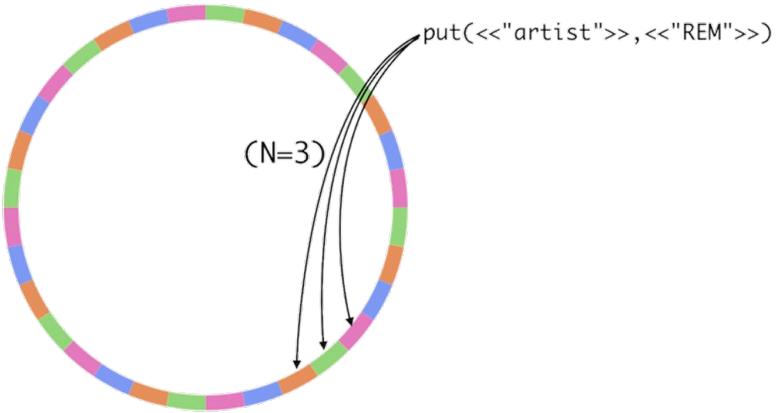
- Center of any cluster: 160-bit integer space (**Riak ring**) which is divided into equally-sized partitions
- Each physical node in the cluster is responsible for `1/(number of physical nodes)` of the ring
- Physical nodes run virtual nodes (vnodes)
  - vnode is responsible for storing a separate portion of the keys
- Nodes can be added and removed from the cluster dynamically
- Riak will redistribute the data accordingly
- Example:
  - A ring with 32 partitions
  - 4 physical nodes
  - 8 vnodes per node

Riak Ring


2^160
0
a single vnode/partition
node 0
node 1
node 2
node 3
a ring with 32 partitions
2^160/4
hash(<<"artist">>,<<"REM">>)
2^160/2

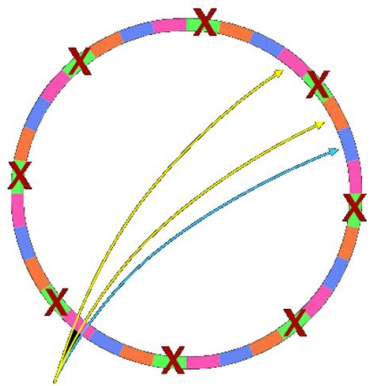
Replication

- Each object is stored on `N` nodes
  - Example: `N = 3`
- Replicas are placed on consecutive partitions on the ring
- Benefits:
  - fault tolerance
  - availability



Hinted Handoff

- Used when a target replica node is temporarily unavailable
  - Riak stores the replica on another reachable node
- The temporary node keeps a hint which node should eventually receive the replica
- When the failed node recovers the data is handed off to the correct node



Hinted handoff means “store now, move later”

Consistency vs Availability



- Quorum Reads and Writes
  - Replication factor: N
  - Write parameter: W
  - Read parameter: R
- Write succeeds if W replicas confirm
- Read queries R replicas
- R + W > N ⟶ read and write sets overlap ⟶ stronger consistency
- Otherwise ⟶ higher availability ⟶ possible stale reads

$W > N/2$

$R + W > N$

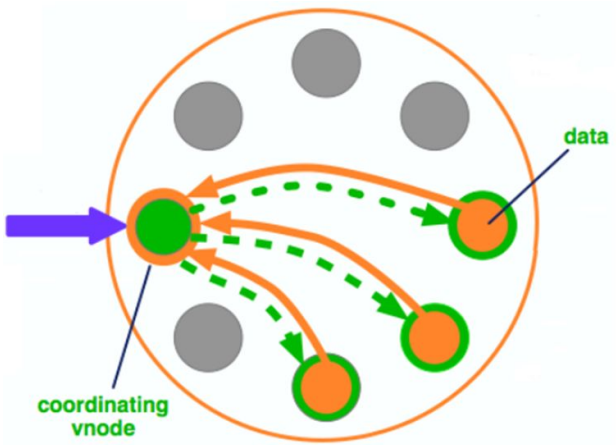
Example:

- N = 3
- W = 2
- R = 2

Request Anatomy

riak

- Each node can be a coordinating vnode = node responsible for a request
- Steps:
  - Finds the vnode for the key according to hash
  - Finds vnodes where other replicas are stored – next N-1 nodes
  - Sends a request to all vnodes
  - Waits until enough requests (R/W) returned the data
    - To fulfill the read/write quorum
  - Returns the result to the client



Coordinator + quorum = scalable distributed read/write

Eventual Consistency



- Not ACID, but BASE (**B**asically **A**vailable, **S**oft state, **E**ventually consistent)
- Eventual Consistency
  - Replicas may temporarily differ
  - Updates are propagated asynchronously
  - Over time replicas converge
- Trade-off: availability vs consistency

Conflicts and Siblings

riak

- Conflicting updates
  - Concurrent writes may create multiple versions
  - Riak stores them as **siblings**
    - No update is lost
- Resolution:
  - automatic (e.g., last-write-wins)
  - application-level
- **Vector Clocks**
  - Track causal relationships between versions
  - Detect:
    - newer version
    - concurrent updates
  - Used for conflict detection

```
a85hYGBgzGDKBVIcR4M2cgczH7HPYEpkzGNlsP/VfYYvCwA=
```

References

- Eric Redmond – Jim R. Wilson: **Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement**
- Pramod J. Sadalage – Martin Fowler: **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**
- Riak https://riak.com/
- Eric Redmond: A Little Riak Book http://littleriakbook.com/

Lecture 04



Document-Oriented & Semi-Structured Data

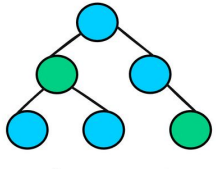
NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Document Stores



- **Store data as documents** (JSON, BSON, XML)
  - Documents are self-describing hierarchical structures (aggregates)
  - Documents are typically grouped into collections
  - Natural representation of application objects
- Flexible schema - fields may differ between documents
- Basic operations:
  - Insert/delete a document
  - Retrieve documents by query
  - Update document fields
- **Rich query capabilities**
  - Complex joins are not typical

RavenDB

ArangoDB

CouchDB

mongoDB.

XML vs JSON

```xml
<user id="42">
  <name>Alice</name>
  <email>alice@example.com</email>
  <roles>
    <role>admin</role>
    <role>editor</role>
  </roles>
</user>
```

```json
{
  "id": 42,
  "name": "Alice",
  "email": "alice@example.com",
  "roles": ["admin", "editor"]
}
```

Observations:

- XML uses tags
- JSON uses key–value pairs
- JSON is more compact

XML (Extensible Markup Language)

- Tree-structured document format
- Tag-based representation
- Supports attributes and nested elements
- Designed for document exchange and interoperability
- Strong ecosystem (XSD, XPath, XQuery)

JSON

- Lightweight hierarchical data format
- Based on key–value pairs and arrays
- Designed for data interchange in web applications
- Native to JavaScript and widely used in APIs

Common properties

- hierarchical structure
- semi-structured data
- human-readable text format
- widely supported by programming languages

Structural Differences

Feature	XML	JSON
Data model	Document tree	Object / array structure
Syntax	Tags with opening/closing elements	Key-value pairs
Attributes	Yes	No (use fields instead)
Arrays	Multiple elements	Explicit arrays
Expresiveness	High	Low
Human readability	Moderate	High

- JSON → native for **document databases**
- XML → older **XML databases**

Suitable Use Cases for Document Stores

Event Logging

- Many different applications want to log events
  - Type of data being captured keeps changing

Content Management Systems, Blogging Platforms

- Managing user comments, user registrations, profiles, web-facing documents, …

Web Analytics or Real-Time Analytics

- Parts of the document can be updated
- New metrics can be easily added without schema changes

E-Commerce Applications

- Flexible schema for products and orders
- Evolving data models without expensive data migration

Query-Driven Document Modeling

- Document design should reflect dominant application queries
  - Main decision: embed vs reference
- Optimize for:
  - what is read together
  - what changes frequently
  - what is aggregated often
- Idea: Document schema is not arbitrary, it is shaped by access patterns

Schema-on-Write vs Schema-on-Read

Key idea

- Schema-on-write: structure first → then store data
- Schema-on-read: store data first → interpret structure later

```
CREATE TABLE Orders(
  order_id INT,
  customer VARCHAR(100),
  price DECIMAL
);
```

Data must follow this schema before insertion

Different structures can coexist; the interpretation happens during query processing

```
{ "order_id": 101, "customer": "Alice", "price": 20 }
{ "order_id": 102, "customer": "Bob", "price": "20 EUR" }
{ "order_id": 103, "customer": "Alice", "items": ["book","pen"] }
```

Schema-on-Write vs Schema-on-Read

	Schema-on-Write	Schema-on-Read
Principle	Data must conform to schema before being stored	Schema is applied when data is read
Validation	Strict validation during ingestion	Minimal validation during ingestion
Typical systems	Relational DBMS	Data lakes, most NoSQL systems
Flexibility	Low	High
Query performance	Usually faster (optimized structure)	Can be slower (interpretation at query time)
Data evolution	Schema changes required	Multiple structures can coexist

mongoDB



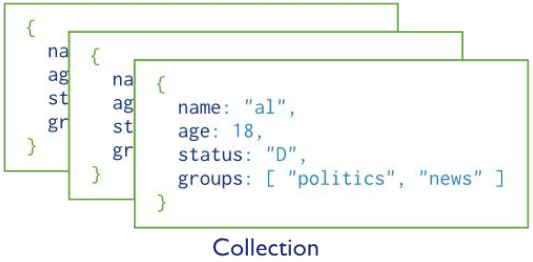
- Initial release: 2009
- Written in C++
  - Open-source
- Cross-platform
- JSON documents
  - Dynamic schemas
- Features:
  - High performance – indices
  - High availability – replication + eventual consistency + automatic failover
  - Automatic scaling – automatic sharding across the cluster
  - MapReduce support

http://www.mongoDB.org/

Terminology

Oracle	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id
join	DBRef

Terminology in Oracle and mongoDB



- Each mongoDB instance has multiple databases
- Each database can have multiple collections
- When we store a document, we have to choose database and collection

Documents

- Use JSON
- Stored as BSON
  - Binary representation of JSON
- Have maximum size: 16MB (in BSON)
  - Not to use too much RAM
  - GridFS tool divides larger files into fragments
- Restrictions on field names:
  - `_id` is reserved for use as a primary key
    - Unique in the collection
    - Immutable
    - Any type other than an array
  - The field names cannot start with the `$` character
    - Reserved for operators
  - The field names cannot contain the `.` character
    - Reserved for accessing fields

https://www.mongodb.com/docs/manual/reference/limits/

BSON (Binary JSON)

- Binary-encoded serialization of JSON documents
  - Allows embedding of documents, arrays, JSON simple data types + other types (e.g., date)
- Purpose
  - efficient storage
  - fast traversal of documents

```
{
  "name": "Alice",
  "age": 30
}
```

```
16 00 00 00         total document size
02                  type: string
6e 61 6d 65 00      field name: "name"
06 00 00 00         string length
41 6c 69 63 65 00   value: "Alice"
10                  type: int32
61 67 65 00         field name: "age"
1e 00 00 00         value: 30
00                  end of document
```

MongoDB stores JSON-like documents, but internally uses BSON

http://bsonspec.org/

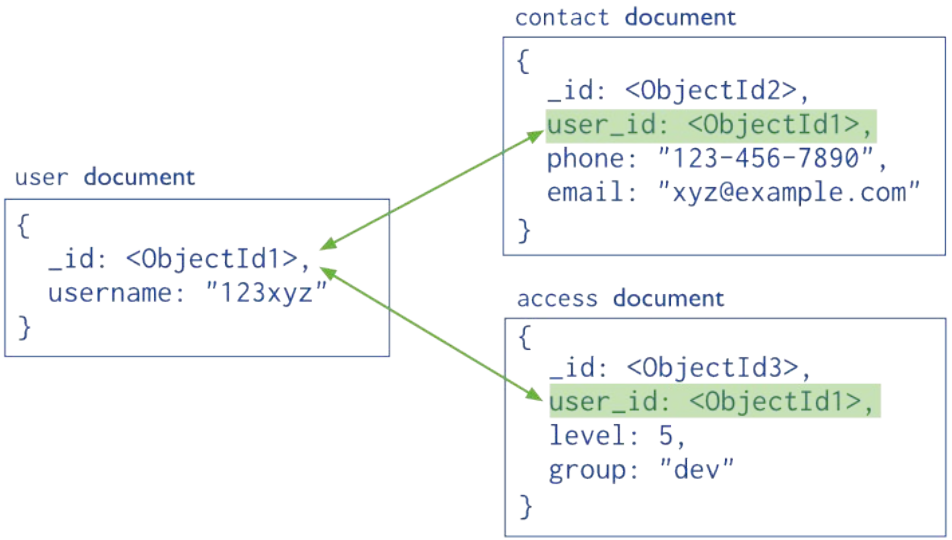
Data Model

mongoDB

- Documents have flexible schema
  - Collections do not enforce structure of data
  - In practice the documents are similar
- Challenge: Balancing
  - the needs of the application
  - the performance characteristics of database engine
  - the data retrieval patterns
- Key design decision: references vs. embedded documents
  - Factors
    - access patterns
    - update frequency
    - data size

References

- Including links / references from one document to another
  - More flexibility than embedding
- Use normalized data models:
  - When embedding would result in duplication of data not outweighted by read performance
  - To represent more complex many-to-many relationships
  - To model large hierarchical data sets
- Disadvantages:
  - Can require more roundtrips to the server (follow up queries)



Embedded Data

- Related data in a single document structure
  - Documents can have subdocuments
  - Applications may need to issue less queries
- Denormalized data models
- Allow to manipulate related data in a single database operation
- Provides:
  - Better performance for read operations
  - Ability to retrieve/update related data in a single database operation
- Disadvantages:
  - Documents may significantly grow
  - Only one “view” of the data



Embedding vs Referencing – Trade-off

	Embedding	Referencing
Idea	Related data stored inside one document	Related data stored in separate documents
Read performance	Fast reads (single query)	May require multiple queries or joins
Data redundancy	Duplication of data	Normalized data
Updates	Updating duplicated data may be expensive	Update in one place
Typical usage	Aggregate objects	Shared entities

When to Use Embedding vs Referencing

- Use **Embedding** when
  - data is naturally hierarchical
  - data is usually read together
  - updates are rare
  - document size stays reasonable
- Typical use cases
  - orders with items
  - blog posts with comments
  - user profile with addresses

```
order
 └ items[]
    └ product snapshot
```

- Use **Referencing** when
  - entities are shared across many documents
  - data changes frequently
  - relationships are many-to-many
  - documents would become too large
- Typical use cases
  - customers
  - products
  - large hierarchical structures

```
orders → customer_id
orders → product_id
```

When to Use Embedding vs Referencing

Embedding

- one collection
- customer data are inside the order document

```
db.orders.find(
  {
    "customer.given_name": "James",
    "customer.family_name": "Thompson"
  },
  { _id: 0 }
)
```

Referencing

- customer data are stored separately
- requires an extra step

```
db.people.find(
  {
    given_name: "James",
    family_name: "Thompson"
  },
  { _id: 1 }
)

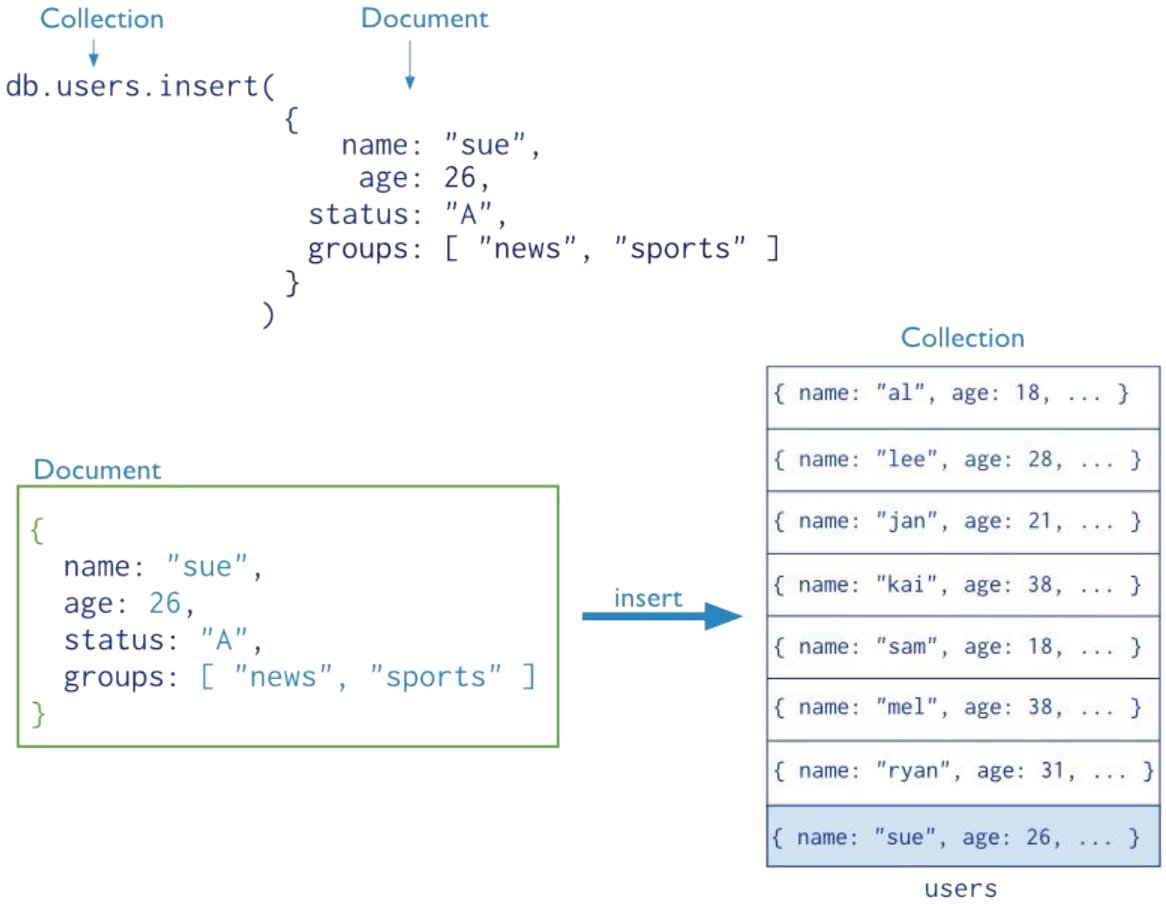
db.orders_flat.find(
  { customer_id: 5595 },
  { _id: 0 }
)
```

Data Modification

- CRUD operations
  - Create - insert new document
  - Read - retrieve documents using queries
  - Update - modify fields in existing documents
  - Delete - remove documents
- Operations always target a collection of documents

MongoDB operations modify whole documents or their fields

```
db.orders.insertOne({...})
db.orders.find({...})
db.orders.updateOne({...})
db.orders.deleteOne({...})
```


Collection
Document
db.users.insert(
{
name: "sue",
age: 26,
status: "A",
groups: ["news", "sports"]
}
)
Document
{
name: "sue",
age: 26,
status: "A",
groups: ["news", "sports"]
}
insert
Collection
{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }
users

Example – Insert / Update / Delete

```
db.inventory.insertOne({
    type: "misc",
    item: "card",
    qty: 15
})
```

insertOne() → one network request per document
insertMany() → batch insert (more efficient)

```
db.inventory.insertMany([
{ type: "food", item: "apple", qty: 50, price: 0.5 },
{ type: "food", item: "banana", qty: 80, price: 0.3 },
{ type: "book", item: "notebook", qty: 40, price: 5.0
},
{ type: "misc", item: "pen", qty: 100, price: 1.2 }
])
```

```
db.inventory.updateOne(
    { item: "card" },
    { $set: { qty: 20 } }
)
```

```
db.inventory.deleteOne(
    { item: "card" }
)
```

MongoDB modifies documents, not rows

MongoDB Update Operators

Operator	Purpose	Example
$set	set value of (or add) a field	update price
$inc	increment numeric value	increase quantity
$push	add element to array	add tag
$pull	remove element from array	remove tag

```
{
  item: "notebook",
  qty: 40,
  tags: ["school", "paper"]
}
```

```
db.inventory.updateOne(
  { item: "notebook" },
  { $inc: { qty: 10 } }
)
```

```
db.inventory.updateOne(
  { item: "notebook" },
  { $set: { price: 5.0 } }
)
```

```
db.inventory.updateOne(
  { item: "notebook" },
  { $pull: { tags: "paper" } }
)
```

```
db.inventory.updateOne(
  { item: "notebook" },
  { $push: { tags: "office" } }
)
```

Update operators modify fields without replacing the entire document

updateOne vs updateMany vs replaceOne

Operation	Description
updateOne()	updates the first matching document
updateMany()	updates all matching documents
replaceOne()	replaces the entire document

```
{
  item: "notebook",
  qty: 40,
  price: 5
}
```

```
db.inventory.updateOne(
  { item: "notebook" },
  { $set: { price: 6 } }
)
```

```
db.inventory.updateMany(
  { type: "food" },
  { $inc: { qty: 10 } }
)
```

```
db.inventory.replaceOne(
  { item: "notebook" },
  { item: "notebook", qty: 50 }
)
```

MongoDB Query Structure

```
{
  item: "apple",
  type: "food",
  qty: 50,
  price: 0.5
}
```

- Basic query syntax:

  `db.collection.find(filter, projection)`

- Parameters
  - `collection` - collection to query
  - `filter` - conditions selecting documents
  - `projection` - fields returned in the result

```
db.inventory.find(
  { type: "food" },
  { item: 1, qty: 1 }
)
```

```
db.inventory.find(
  { type: "food" },
  { item: 1, qty: 1, _id: 0 }
)
```

Querying Nested Documents

- MongoDB allows querying fields inside nested documents using dot notation
- Note:
  - customer.given_name → nested document
  - items.0.product → array

```
{
  _id: 101,
  customer: {
    given_name: "Alice",
    family_name: "Smith"
  },
  total: 120
}
```

```
db.orders.find(
  { "customer.given_name": "Alice" }
)
```

Querying Arrays

```
{
  item: "orange",
  tags: ["fruit", "food", "citrus"]
}
```

```
db.inventory.find(
  { tags: "fruit" }
)
```

```
db.inventory.find(
  { "tags.0": "fruit" }
)
```

```
{
  item: "notebook",
  memos: [
    { by: "shipping", memo: "on time" },
    { by: "billing", memo: "paid" }
  ]
}
```

Query matches if at least one array element satisfies the condition

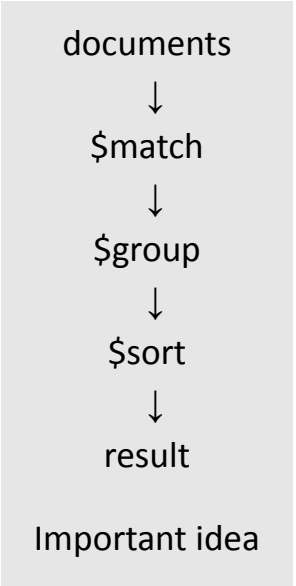
Query	Meaning
{ tags: "fruit" }	array contains value
{ tags: ["fruit","food","citrus"] }	exact array match

```
db.inventory.find(
  { "memos.by": "shipping" }
)
```

MongoDB Aggregation Pipeline

- Aggregation framework processes documents in multiple stages
  - Pipeline = sequence of transformations
  - Similar to SELECT → WHERE → GROUP BY → ORDER BY
- Each stage transforms the data and passes it to the next stage

```
db.orders.aggregate([
  { $match: {...} },
  { $group: {...} },
  { $sort: {...} }
])
```



Common Aggregation Stages

Stage	Purpose	SQL analogy
$match	filter documents	WHERE
$project	select/rename fields	SELECT
$group	aggregate values	GROUP BY
$sort	order results	ORDER BY
$limit	restrict number of results	LIMIT
$unwind	flatten array	JOIN / explode

Example Aggregation

```
{
  order_id: 1,
  items: [
    { product_id: 10, quantity: 2 },
    { product_id: 11, quantity: 1 }
  ]
}
```

Compute total quantity sold per product

```
db.orders.aggregate([
  { $unwind: "$items" },
  {
    $group: {
      _id: "$items.product_id",
      total_qty: { $sum: "$items.quantity" }
    }
  },
  { $sort: { total_qty: -1 } }
])
```

```
{ "_id": 10, "total_qty": 120 }
{ "_id": 11, "total_qty": 95 }
```

Aggregation pipeline processes documents step by step

find() ≈ SELECT ... WHERE
aggregate() ≈ SELECT ... GROUP BY ...

`find()` **vs** `aggregate()`

	find()	aggregate()
Purpose	retrieve documents	transform and analyze data
Complexity	simple queries	multi-stage processing
Operations	filtering, projection	grouping, sorting, reshaping
Typical use	application queries	analytics and reports

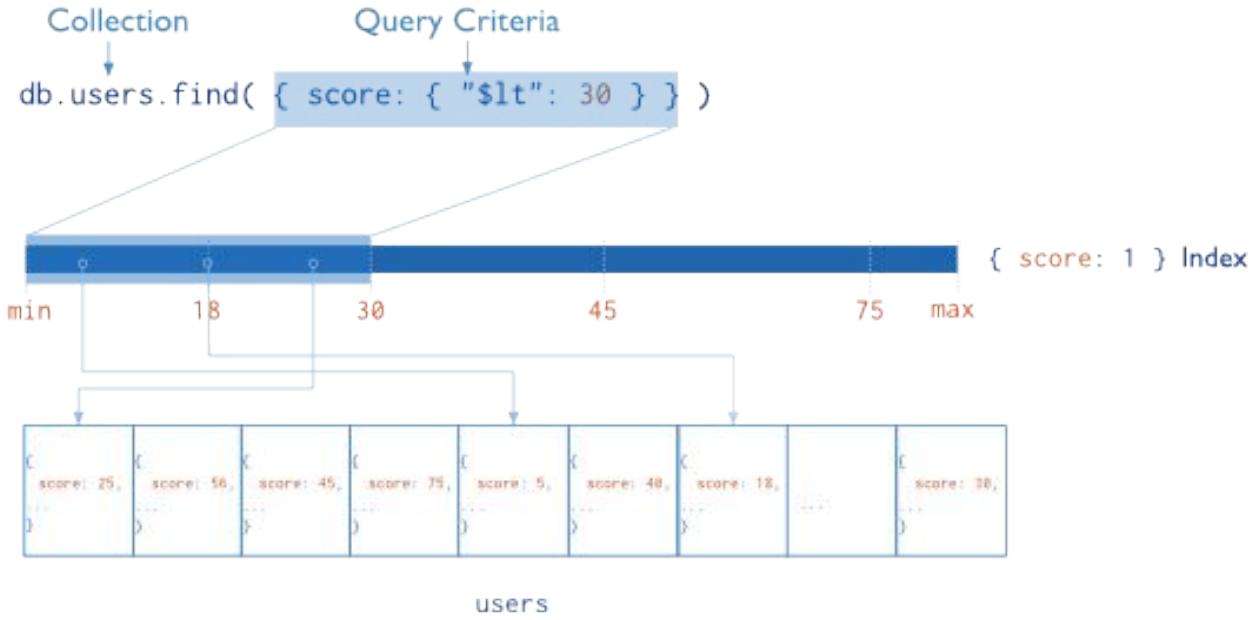
```
db.orders.find({ region: "EU-CZ" })
```

```
db.orders.aggregate([
  { $match: { region: "EU-CZ" } },
  { $group: { _id: "$customer_id", total: { $sum: "$total_price" } } }
])
```

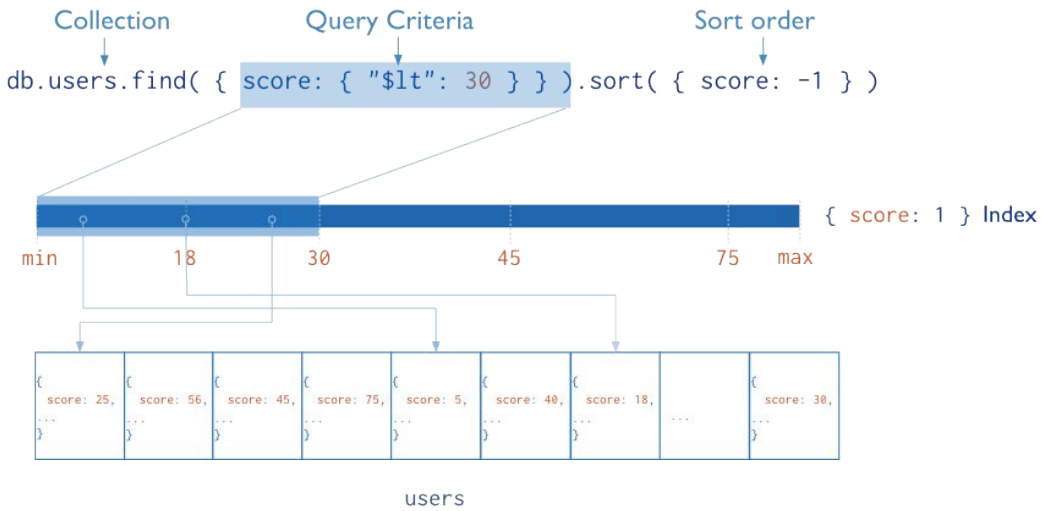
Indices

- Without indices:
  - MongoDB scans all documents (operation called COLLSCAN)
- With index
  - MongoDB searches using index structure (operation called IXSCAN)
- Indices store a portion of the collection's data set in an easy to traverse form
  - Stores the value of a specific field or a set of fields ordered by the value of the field
  - B-tree like structures
- Defined at collection level
- Purpose:
  - To speed up common queries
  - To optimize the performance of other operations in specific situations

Indices – Example

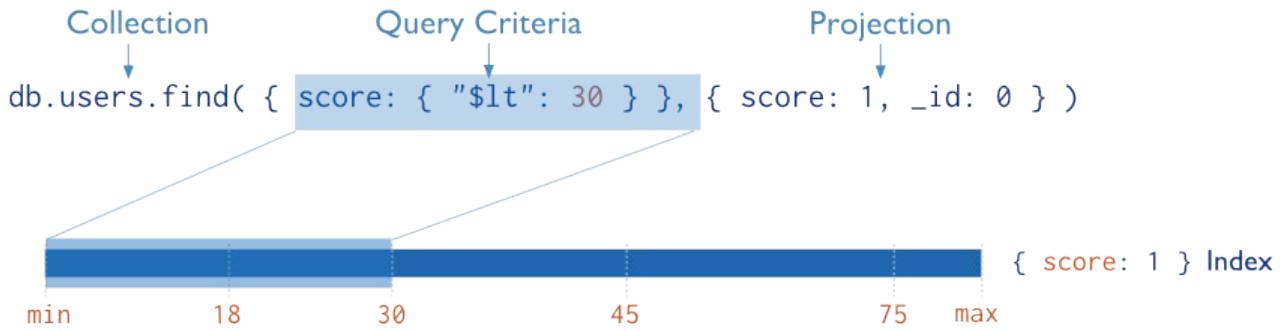


Indices – Sorted Results



- The index stores `score` values in ascending order
- mongoDB can traverse the index in either ascending or descending order to return sorted results (without sorting)

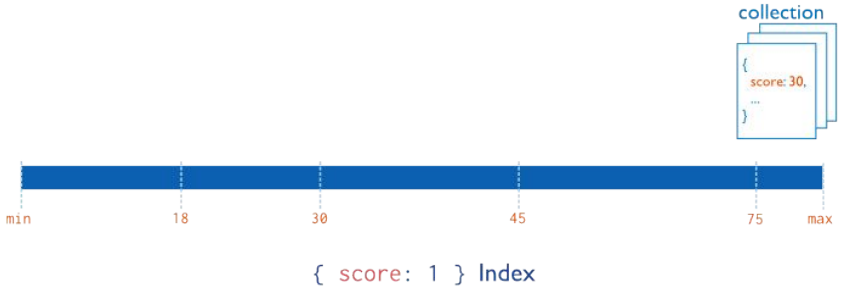
Indices – Covered Results



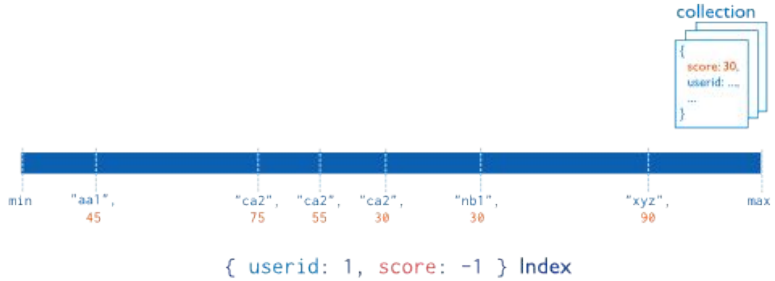
- mongoDB does not need to inspect data outside of the index to fulfil the query

Index Types

- **Default `_id`**
  - Exists by default
    - If applications do not specify `_id`, it is created automatically
  - Unique by default
- **Single Field**
  - User-defined indices on a single field of a document
- **Compound**
  - User-defined indices on multiple fields
- **Multikey index**
  - To index the content stored in arrays
  - Creates separate index entry for every element of the array



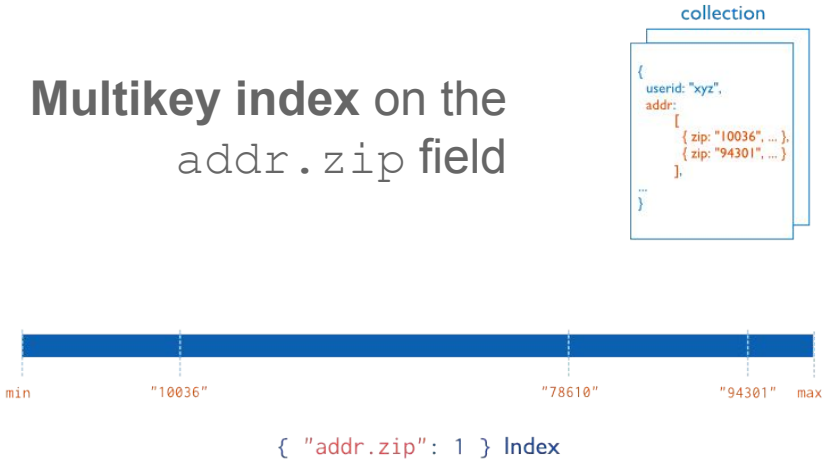
Single field index on the `score` field (ascending).



Compound index on the `userid` field (ascending) and the `score` field (descending).

- sorts first by `userid` and then, within each `userid` value, sort by `score`

Multikey index on the `addr.zip` field



Index Types

- **Geospatial Field**
  - 2d indexes = use planar geometry when returning results
    - For data representing points on a two-dimensional plane
  - 2sphere indexes = use spherical (Earth-like) geometry to return results
    - For data representing longitude, latitude
- **Text Indexes**
  - Searching for string content in a collection
- **Hash Indexes**
  - Indexes the hash of the value of a field
  - Only support equality matches (not range queries)

Indexes

```
db.people.createIndex( { "phone-number": 1 } )
```

- Creates a single-field index on the `phone-number` field of the `people` collection

```
db.products.createIndex( { item: 1, category: 1, price: 1 } )
```

- Creates a compound index on the `item`, `category`, and `price` fields

```
db.accounts.createIndex( { "tax-id": 1 }, { unique: true } )
```

- Creates a unique index
  - Prevents applications from inserting documents that have duplicate values for the inserted fields

```
db.collection.createIndex( { _id: "hashed" } )
```

- Creates a hashed index on `_id`

Indexes + `explain()`

- Without index
  - Likely COLLSCAN
  - Many examined documents
  - Possible extra sort stage
- After index
  - IXSCAN
  - Fewer examined documents
  - Sort can be index-supported

```
db.orders.aggregate([
  { $match: { region: "EU-CZ" } },
  { $sort: { order_date: -1 } },
  { $limit: 20 }
]).explain("executionStats")

db.orders.createIndex({
  region: 1,
  order_date: -1 })
```

MongoEngine

- Object Document Mapper for Python
  - Maps collections and embedded documents to classes
- Improves readability of application code
- Same MongoDB logic underneath
  - ODM does not remove modeling trade-offs

```
class Customer(EmbeddedDocument):
    id = IntField(required=True)
    given_name = StringField()
    family_name = StringField()

class Order(Document):
    meta = {"collection": "orders"}
    region = StringField()
    order_date = DateTimeField()
    customer = EmbeddedDocumentField(Customer)

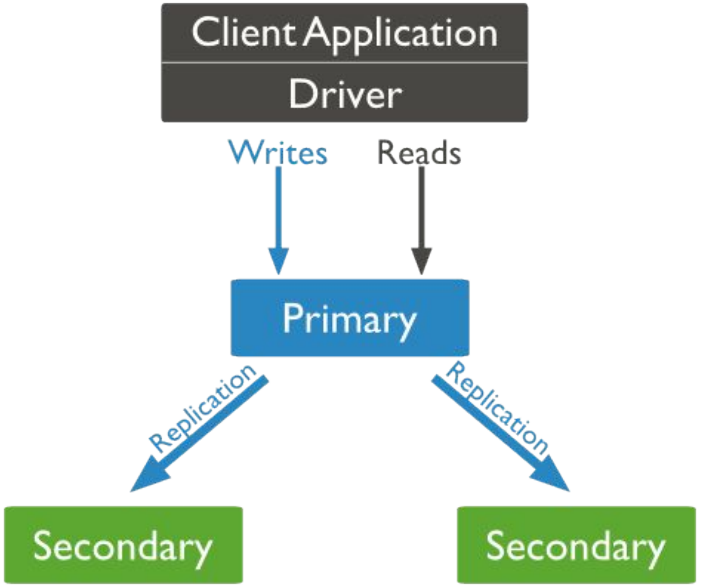
Order.objects(region="EU-CZ")
     .order_by("-order_date") [:20]
```

```
db.orders.find(
  { region: "EU-CZ" }
)
.sort({ order_date: -1 })
.limit(20)
```

More on Internals of 

Replication

- Master/slave replication
- Replica set = group of instances that host the same data set
  - **primary** (master) – receives all write operations
  - **secondaries** (slaves) – apply operations from the primary so that they have the same data set



Replication Steps

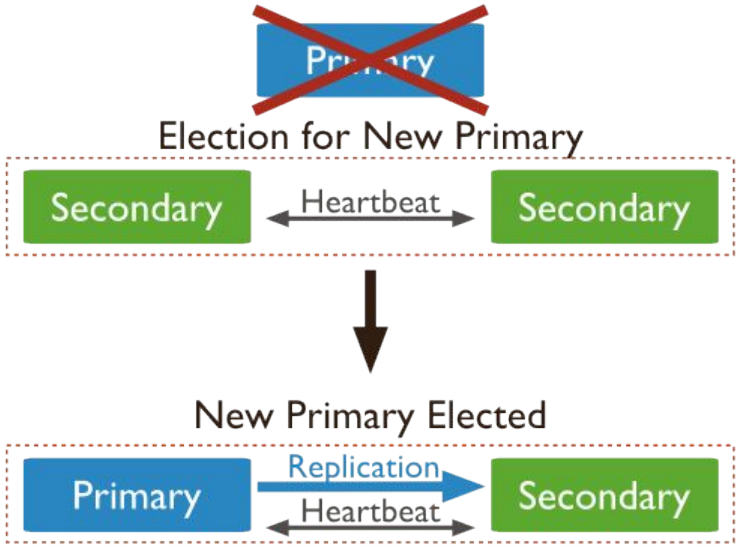
- **Write:**
  1. mongoDB applies write operations on the primary
  2. mongoDB records the operations to the primary's oplog
  3. Secondary members replicate oplog + apply the operations to their data sets
- **Read:** All members of the replica set can accept read operations
  - By default, an application directs its read operations to the primary member
    - Guaranties the latest version of a document
    - Decreases read throughput
  - Read preference mode can be set

Replication – Read Preference Mode

Read Preference Mode	Description
`primary`	operations read from the current replica set primary
`primaryPreferred`	operations read from the primary, but if unavailable, operations read from secondary members
`secondary`	operations read from the secondary members
`secondaryPreferred`	operations read from secondary members, but if none is available, operations read from the primary
`nearest`	operations read from the nearest member (= shortest ping time) of the replica set, irrespective of the member's type

Replica Set Elections

- Replica set can have at most one primary
- If the current primary becomes unavailable, an election determines a new primary
- Note:
  - Elections need some time
    - Approx. 1 minute
  - No primary ⇒ no writes



Replica Set Elections – Influencing Factors

- **Heartbeat** (ping)
  - Every 2s sent to each other
  - No response for 10s ⇒ node is inaccessible
- **Priority comparisons**
  - Higher priority = preferred to be voted
  - Members with priority = 0
    - Cannot become primary (not eligible)
    - Cannot trigger election, but can vote
  - The current primary has the highest priority and is within 10s of the latest oplog entry ⇒ OK
  - A higher-priority member catches up to within 10s of the latest oplog entry of the current primary ⇒ elections
    - The higher-priority node has a chance to become primary
- **Connections**
  - A node cannot become primary unless it can connect to a majority of the members

Replica Set Elections – Mechanism

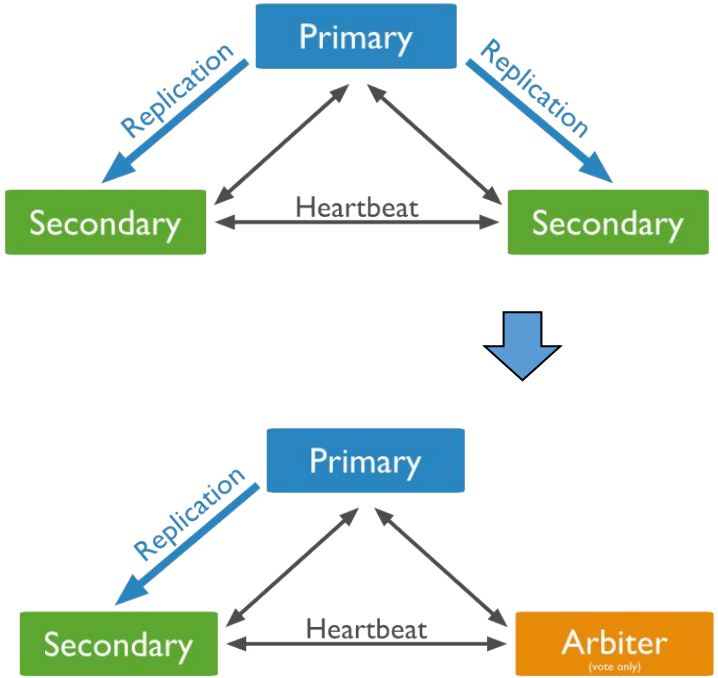
- Replica sets hold an election any time there is no primary:
  - Initiation of a new replica set
  - A secondary loses contact with a primary
  - A primary steps down
- A primary will step down:
  - After receiving the `replSetStepDown` command
    - Forces a primary to become a secondary
  - If one of the current secondaries is eligible for election and has a higher priority
  - If it cannot contact a majority of the members of the replica set

Replica Set Elections – Mechanism

- The replica set elects an eligible member with the highest priority value as primary
  - By default, all members have a priority of 1
    - Can be adjusted
- The first member to receive the majority of votes becomes primary
  - By default, all members have 1 vote
    - Can be disabled = non-voting members
      - Hold copies of data
      - Can become primary
    - Not recommended to set more than 1 (better use priority)
- All members of a replica set can veto an election, e.g.,
  - If the member seeking an election is not up-to-date with the most recent operation accessible in the replica set
  - If the member seeking an election has a lower priority than another member in the set that is also eligible for election
  - …

Replication – Arbiters

- Arbiter
  - A special node
  - Does not maintain a data set
    - Does not require dedicated hardware
  - Cannot be a primary
  - Exists to vote in elections
    - For replicas with even number of members

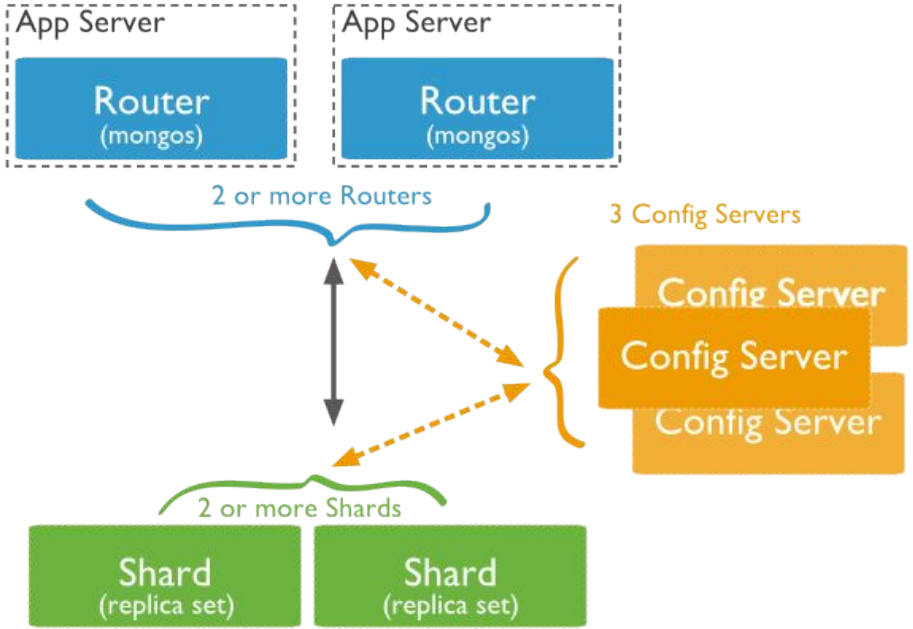


Replication – Secondaries

- A secondary can be configured as:
  - **Priority 0** – to prevent it from becoming a primary in an election
    - e.g., a standby
  - **Hidden** – to prevent applications from reading from it
    - Just replicates the data for special usage
    - Can vote in elections
  - **Delayed** – to keep a running “historical” snapshot
    - For recovery from errors like unintentionally deleted databases

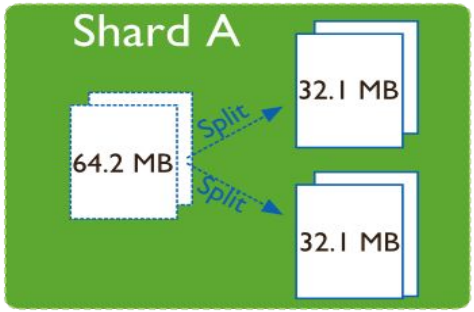
Sharding

- Supported through sharded clusters
  - **Shards** – store the data
    - Each shard is a replica set
  - **Query routers** – interface with client applications
    - Direct operations to the appropriate shard(s) + return the result to the user
    - More than one ⇒ to divide the client request load
  - **Config servers** – store the cluster's metadata
    - Mapping of the cluster's data set to the shards
    - Recommended number: 3



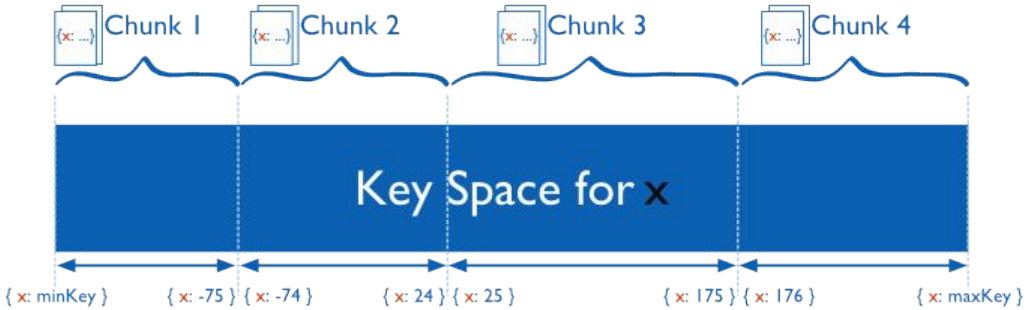
Data Partitioning

- Partitions a collection's data by the shard key
  - Indexed (possibly compound) field that exists in every document in the collection
    - Immutable
  - Divided into chunks distributed across shards
    - **Range-based partitioning**
    - **Hash-based partitioning**
  - When a chunk grows beyond the chunk size, it is split
    - Small chunks ⇒ more even distribution at the expense of more frequent migrations
    - Large chunks ⇒ fewer migrations



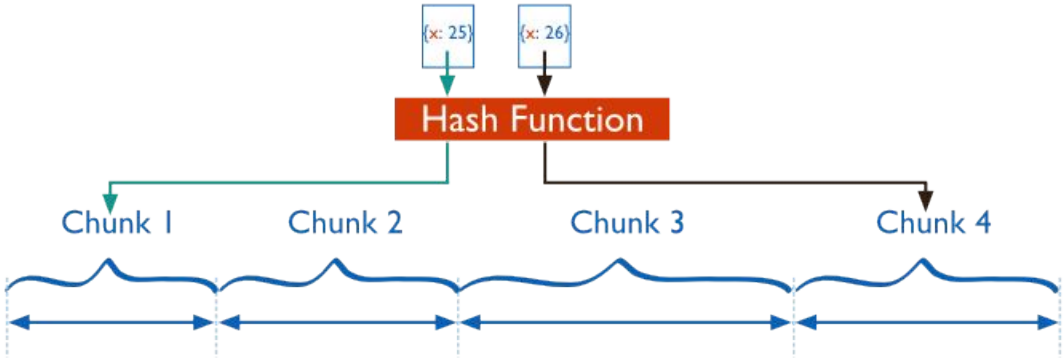
Range-Based Partitioning

- Each value of the shard key falls at some point on line from negative infinity to positive infinity
- The line is partitioned into non-overlapping chunks
- Documents with “close” shard key values are likely to be in the same chunk
  - More efficient range queries
  - Can result in an uneven distribution of data



Hash-Based Partitioning

- Computes a hash of a field's value
  - Hashes form chunks
- Ensures a more random distribution of a collection in the cluster
  - Documents with "close" shard key values are unlikely to be a part of the same chunk
  - A range query may need to target most/all shards



Transactions

- Write operations are atomic at the level of a single document
  - Including nested documents (sufficient for many cases, but not all)
- When a single write operation modifies multiple documents, it is not atomic
  - Other operations may interleave
- Isolation of a single write operation that affects multiple documents
  - No client sees the changes until the operation completes or errors out
  - `db.foo.update( { field1 : 1 , $isolated : 1 }, { $inc : { field2 : 1 } } , { multi: true } )`

mongoDB Enterprise



- Commercial edition of mongoDB
- Includes:
  - **Advanced Security** – Kerberos authentication
  - **Management Service** – a suite of tools for managing mongoDB deployments
    - Monitoring, backup capabilities, helping users optimize clusters, …
  - **Enterprise Software Integration** – SNMP support to integrate mongoDB with other tools
  - **Certified OS Support** – has been tested and certified on Red Hat/CentOS, Ubuntu, SuSE and Amazon Linux
  - …

https://www.mongoDB.com/products/mongoDB-enterprise

References

- Eric Redmond – Jim R. Wilson: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement
- Pramod J. Sadalage – Martin Fowler: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence
- Tiny mongoDB Browser Shell: https://mongoplayground.net/
- mongoDB Manual: http://docs.mongoDB.org/manual/

Lecture 05



Wide-Column & Time-Series Databases

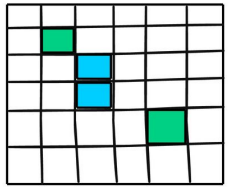
NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

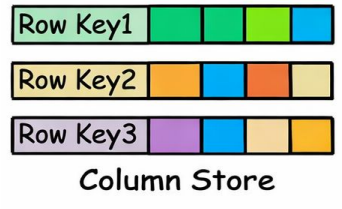
irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Wide-Column Stores



- Also: column-family, columnar, …
- Store data in rows with flexible columns
  - Data are organized into **column families**
  - Each row may contain a different set of columns
  - Suitable for sparse and large-scale data
- Schema **flexibility** – columns may differ between rows
- Basic operations:
  - Insert/delete a row
  - Retrieve rows by key
  - Update column values
- Efficient for large-scale access
  - Fast writes and lookups by primary key
  - Complex joins are not typical



Google BigTable

ScyllaDB

APACHE HBASE

cassandra

Another Meaning of Column-Oriented

- Stores data tables as columns rather than as rows
- Maps data to rowIds = a closer structure to an index
  - More efficient query operations (some of them)
    - E.g. Find all people with name Jones
  - Retrieving a whole row may be slower

RowId	EmpId	Lastname	Firstname	Salary
001	10	Smith	Joe	40000
002	12	Jones	Mary	50000
003	11	Johnson	Cathy	44000
004	22	Jones	Bob	550



```
001:10,Smith,Joe,40000;
002:12,Jones,Mary,50000;
003:11,Johnson,Cathy,44000;
004:22,Jones,Bob,55000;
```



```
Jones:002,004
```



```
10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;
```

Typical Use Cases

Event logging and telemetry

- High write throughput, append-heavy data

User profiles and activity data

- Rows may have different sets of attributes

Content platforms

- Denormalized data grouped by access patterns

Apache Cassandra

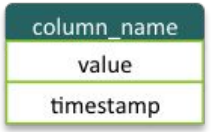


- Distributed wide-column data store
  - Peer-to-peer architecture
  - Designed for scalability and high availability
- Originally developed at Facebook
- Initial release: 2008
  - Apache Licence
- Written in: Java
  - Cross-platform
- Operations:
  - **CQL** (Cassandra Query Language)
  - Integration with Hadoop ecosystem

http://cassandra.apache.org/

Terminology

- **Column** = name–value pair with timestamp metadata
  - Timestamp is used for versioning / conflict resolution)
- **Row** = a collection of columns identified by a row key
- **Column family** = a collection of similar rows
  - Rows do not have to have the same columns



RDBMS	Cassandra
database instance	cluster
database	keyspace
table	column family
row	row
column (same for all rows)	column (can be different per row)

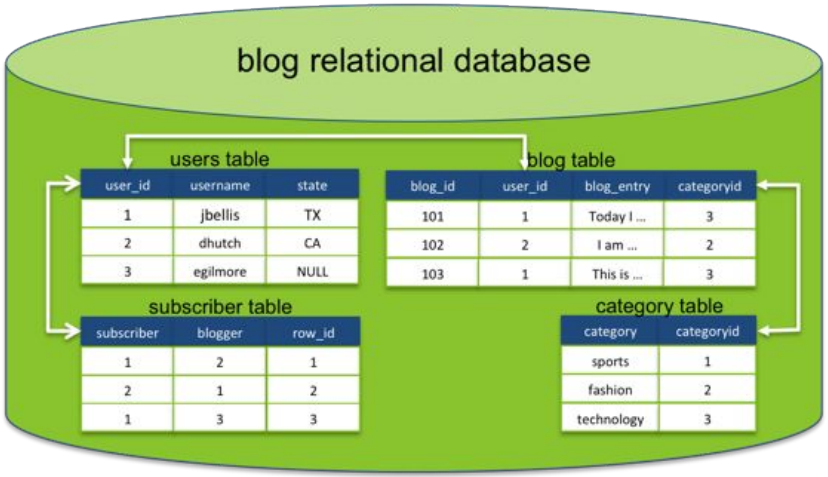


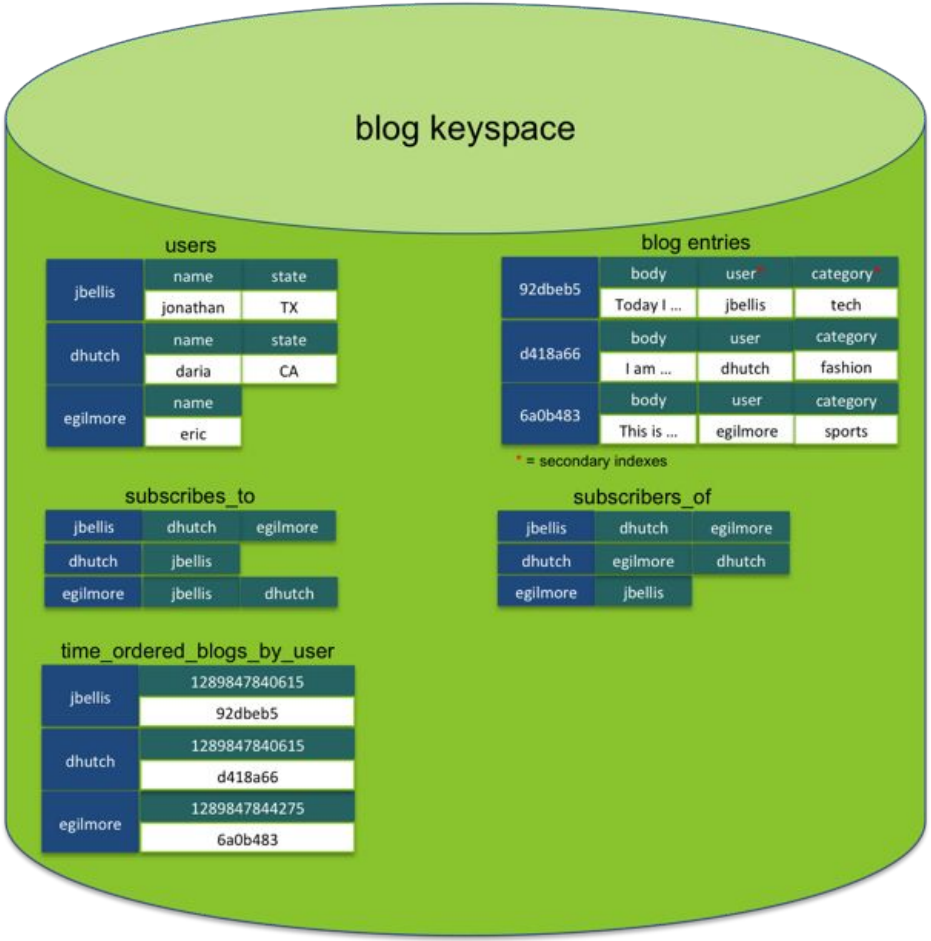
Data Model – Example

- Column:
  - `firstName : Martin`
  - `lastName : Fowler`
- Row:
  - `Row key: martin-fowler`
  - `Columns: firstName, lastName, location`
- Column Family
  - `martin-fowler → firstName, lastName, location`
  - `pramod-sadalage → firstName, lastName, lastVisit`

Column-families vs. Relations

- Flexible schema
  - Columns do not have to be defined in advance
  - Rows may contain **different sets of columns**
  - The schema can evolve over time
- Access-pattern-oriented design
  - Column families group data accessed together
  - **Denormalization** is common
  - Modeling is driven by expected queries
- Limited relational features
  - **No formal foreign keys**
  - Joins are usually not supported at query time
  - Secondary indexes and precomputed views may help


blog relational database
users table
user_id username state
1 jbellis TX
2 dhutch CA
3 egilmore NULL
blog table
blog_id user_id blog_entry categoryid
101 1 Today I ... 3
102 2 I am ... 2
103 1 This is ... 3
subscriber table
subscriber blogger row_id
1 2 1
2 1 2
1 3 3
category table
category categoryid
sports 1
fashion 2
technology 3

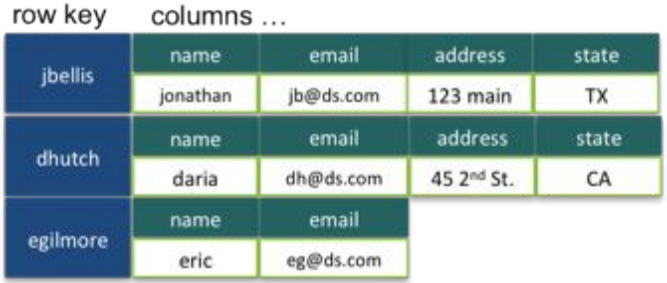

blog keyspace
users
jbellis name state
jonathan TX
dhutch name state
daria CA
egilmore name
eric
blog entries
92dbeb5 body user* category*
Today I ... jbellis tech
d418a66 body user category
I am ... dhutch fashion
6a0b483 body user category
This is ... egilmore sports
* = secondary indexes
subscribes_to
jbellis dhutch egilmore
dhutch jbellis
egilmore jbellis dhutch
subscribers_of
jbellis dhutch egilmore
dhutch egilmore dhutch
egilmore jbellis
time_ordered_blogs_by_user
jbellis 1289847840615
92dbeb5
dhutch 1289847840615
d418a66
egilmore 1289847844275
6a0b483

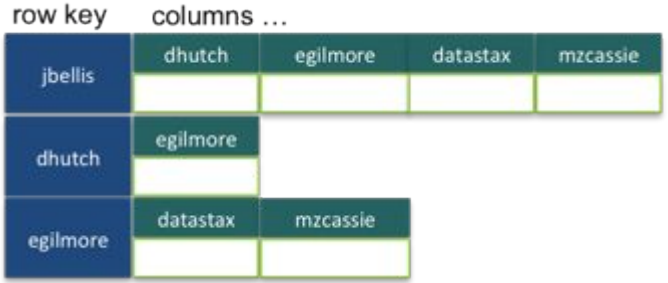
Column-families



- **Static column families**
  - Similar to a relational table
  - Rows usually share the same set of columns
  - Not all columns have to be present in every row
- **Dynamic column families**
  - Use application-defined column names
  - Support flexible and evolving row structure
- Materialized / query-oriented rows
  - Precomputed result sets can be stored in a single row
  - A row may represent a snapshot of data matching a query
  - Similar to a materialized view

Static vs. Dynamic Column Families

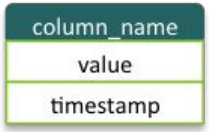




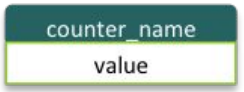
Example: subscribers of a user's blog

Columns

- A column is the smallest data unit
  - Name, value, and timestamp



- Columns are stored within a row identified by a row key
- Special column features
  - **TTL** (time to live)
    - a column value may expire automatically after a specified time
  - **Counter** columns
    - support incremental updates
    - useful for counts such as page views or events



Super Columns (Legacy Concept)

- A super column groups multiple columns under one name
  - Historically used to represent nested data
  - In modern Cassandra/CQL rarely used
- Similar needs are usually handled by tables or collection types



Cassandra Query Language (CQL)



- SQL-like syntax
  - CREATE, ALTER, INSERT, UPDATE, DELETE, DROP, TRUNCATE
- Table-oriented data model
  - Since CQL 3, schema is expressed as tables (not column families)
- Simpler query model than SQL
  - no joins
  - limited subqueries
  - queries are driven mainly by the primary key and table design

CQL Data Types



`ascii`	ASCII character string
`bigint`	64-bit signed long
`blob`	Arbitrary bytes (no validation)
`boolean`	true or false
`counter`	Counter column (64-bit long)
`decimal`	Variable-precision decimal
`double`	64-bit IEEE-754 floating point
`float`	32-bit IEEE-754 floating point
`int`	32-bit signed int
`text`	UTF8 encoded string
`timestamp`	A timestamp
`uuid`	A UUID in standard UUID format
`varchar`	UTF8 encoded string
`varint`	Arbitrary-precision integer
…	…

User Defined Types (UDT)



- Structured nested data inside a single column
- Alternative to joins / normalization
- Typical in denormalized read mode
- Frozen types
  - Cassandra by default allows partial updates
  - `frozen` - one atomic blob
    - Entire value is read/written at once
    - Cannot update individual fields inside
    - For: UDT in collections, complex embedded objects

```
CREATE TYPE offer_udt (
  region text,
  price decimal,
  currency text,
  valid_from timestamp,
  valid_to timestamp
);

CREATE TABLE title_page_by_id (
  title_id text PRIMARY KEY,
  name text,
  offers list<frozen<offer_udt>>
);
```

Working with a Keyspace



- A keyspace is the top-level namespace in Cassandra
- It defines how data are replicated across the cluster
  - In production, replication is usually configured more carefully than in simple examples

```
CREATE KEYSPACE test
WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': 3
};

USE test;
…
DROP KEYSPACE test;
```

Working with a Table – Primary Key



- A table defines columns and a primary key
- The primary key determines how data are partitioned and ordered
  - **Partition key** - how data are distributed across the cluster
  - **Clustering column** - how rows are ordered within a partition
- Table design should reflect the expected access patterns

```
CREATE TABLE users (
  user_id uuid PRIMARY KEY,
  first_name text,
  last_name text,
  email text
);
```

```
CREATE TABLE messages (
  user_id uuid,
  message_time timestamp,
  message text,
  PRIMARY KEY (user_id, message_time)
);
```

- `user_id` = partition key
- `message_time` = clustering column

Time-Series Pattern



- Stores a sequence of timestamped events or measurements
  - Groups data by **entity into one partition**
  - Uses **time** as a **clustering column**
  - Supports efficient reads over a time interval

```
CREATE TABLE measurements_by_sensor_day (
    sensor_id text,
    day date,
    ts timestamp,
    value double,
    PRIMARY KEY ((sensor_id, day), ts)
) WITH CLUSTERING ORDER BY (ts DESC);
```

- `(sensor_id, day)` = **partition key**
  - Rows for one sensor and one day are stored together
- `ts` = **clustering column**
  - Within the partition, rows are ordered by time

Working with a Table – Column Expiration



- Cassandra can automatically expire data after a specified time
- TTL is useful for sessions, logs, caches, or temporary records

```
INSERT INTO clicks (userid, url, date, name)
VALUES (
  3715e600-2eb0-11e2-81c1-0800200c9a66,
  'http://apache.org',
  '2013-10-09',
  'Mary'
) USING TTL 86400;
```

```
UPDATE clicks USING TTL 86400
SET name = 'Mary'
WHERE userid =
   3715e600-2eb0-11e2-81c1-0800200c9a66
AND url = 'http://apache.org';
```

```
SELECT TTL(name) FROM clicks
WHERE userid = 3715e600-2eb0-11e2-81c1-0800200c9a66
AND url = 'http://apache.org';
```

Working with a Table – Collections



- Collection types
  - **set** — a set of unique values
  - **list** — an ordered list of values
  - **map** — key–value pairs
- Notes
  - Collections are useful for storing small groups of related values
  - Each collection element is internally stored separately
  - Collection elements may have their own TTL

Working with a Table – Set



- A set stores unique unordered values
  - Duplicate values are ignored automatically
  - Set elements are not indexed by position
- Useful for small groups of tags, categories, or identifiers

```
CREATE TABLE users (
  id uuid PRIMARY KEY,
  emails set<text>
);
```

```
UPDATE users
SET emails = emails + {'c@example.com'} WHERE id = ...;

UPDATE users
SET emails = emails - {'a@example.com'} WHERE id = ...;

SELECT emails FROM users WHERE id = ...;

SELECT * FROM users
WHERE emails CONTAINS 'b@example.com';
```

Working with a Table – List



- A list stores ordered values
  - The same value may appear multiple times

```
ALTER TABLE users ADD top_places list<text>;

UPDATE users SET top_places = [ 'rivendell', 'rohan' ] WHERE user_id = 'frodo';

UPDATE users SET top_places = [ 'the shire' ] + top_places WHERE user_id = 'frodo';

UPDATE users SET top_places = top_places + [ 'mordor' ] WHERE user_id = 'frodo';

UPDATE users SET top_places[2] = 'riddermark' WHERE user_id = 'frodo';

DELETE top_places[3] FROM users WHERE user_id = 'frodo';

UPDATE users SET top_places = top_places - ['riddermark' ] WHERE user_id = 'frodo';
```

Working with a Table – Map



- A map stores key–value pairs
  - Elements are accessed by key, not by position

```
ALTER TABLE users ADD  todo map<timestamp, text>;

UPDATE users SET  todo = {
  '2012-09-24 00:00:00': 'enter Mordor',
  '2012-10-02 12:00:00': 'throw ring into Mount Doom'
} WHERE user_id = 'frodo';

UPDATE users SET  todo['2012-10-02 12:00:00'] = 'throw my precious into Mount Doom'
WHERE user_id = 'frodo';

INSERT INTO users (user_id, todo) VALUES ('frodo', {
  '2013-09-22 12:01:00': 'birthday wishes to Bilbo',
  '2013-10-01 18:00:00': 'check into the Inn of the Prancing Pony'
});

DELETE  todo['2012-09-24 00:00:00'] FROM users WHERE user_id = 'frodo';
```

Working with a Table

- DROP TABLE — removes the table and all its data
- TRUNCATE — removes all rows but keeps the table schema
- CREATE INDEX — creates a secondary index on a column
- DROP INDEX — removes the index

```
DROP TABLE users;

TRUNCATE users;

CREATE INDEX user_last_name_idx ON users (last_name);

DROP INDEX user_last_name_idx;
```

Querying

```
SELECT select_expression
FROM keyspace_name.table_name
WHERE relation [AND relation ...]
[GROUP BY column_name [, column_name ...]]
[ORDER BY clustering_column ASC | DESC]
[LIMIT n]
[ALLOW FILTERING]
```

- Cassandra supports simple queries
- Queries are most efficient when they follow the primary key
    - ALLOW FILTERING may work, but is usually a warning sign
- WHERE conditions are limited compared to SQL
- ORDER BY is only supported within the clustering-key order

```
SELECT * FROM users WHERE user_id = 'frodo';

SELECT * FROM emp WHERE emp_id IN (130, 104);

SELECT * FROM timeline
WHERE userid = 123
ORDER BY posted_month DESC;
```

Model tables for the queries you need, not the other way around

Querying - `select_expression`

- column list
- DISTINCT
- COUNT(*)
- aliases (AS)
- TTL(column_name)
- WRITETIME(column_name)

```
SELECT select_expression
FROM keyspace_name.table_name
WHERE relation [AND relation ...]
[GROUP BY column_name [, column_name ...]]
[ORDER BY clustering_column ASC | DESC]
[LIMIT n]
[ALLOW FILTERING]
```

Querying - `relation`

- Simple conditions
  - column = value
  - column <, >, <=, >= value
  - column IN (...)

```
SELECT select_expression
FROM keyspace_name.table_name
WHERE relation [AND relation ...]
[GROUP BY column_name [, column_name ...]]
[ORDER BY clustering_column ASC | DESC]
[LIMIT n]
[ALLOW FILTERING]
```

- Advanced option
  - TOKEN(...) is useful for scanning partitions by token range, e.g. in parallel batch processing
- Key point: Query conditions are more limited than in SQL

Querying - `GROUP BY`

- Groups rows by selected columns
- Only groupings induced by primary-key columns are allowed
- Supported aggregate functions:
  - COUNT
  - MIN
  - MAX
  - SUM
  - AVG

```
SELECT select_expression
FROM keyspace_name.table_name
WHERE relation [AND relation ...]
[GROUP BY column_name [, column_name ...]]
[ORDER BY clustering_column ASC | DESC]
[LIMIT n]
[ALLOW FILTERING]
```

- Note: If a selected column is neither grouped nor aggregated, Cassandra returns an arbitrary first encountered value

Querying - ALLOW FILTERING

- Explicitly permits a query that may require scanning many rows (non-filtering queries)
  - Such queries may have unpredictable performance
  - A warning sign, not the preferred design

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

```
SELECT select_expression
FROM keyspace_name.table_name
WHERE relation [AND relation ...]
[GROUP BY column_name [, column_name ...]]
[ORDER BY clustering_column ASC | DESC]
[LIMIT n]
[ALLOW FILTERING]
```

Querying - ALLOW FILTERING

```
CREATE TABLE users (
    username text PRIMARY KEY,
    firstname text,
    lastname text,
    birth_year int,
    country text
);

CREATE INDEX ON users (birth_year);

SELECT firstname, lastname
FROM users
WHERE birth_year = 1981;

SELECT firstname, lastname
FROM users
WHERE birth_year = 1981 AND country = 'FR'
ALLOW FILTERING;
```

cqlengine

- Python object mapper for Apache Cassandra
  - Cassandra tables → Python classes
- Enbes to query data using model objects
  - Instead of raw CQL
- Improves readability of application code
- Does not remove Cassandra modeling constraints

```
from cassandra.cqlengine import columns
from cassandra.cqlengine.models import Model

class MeasurementsBySensorDay(Model):
    __keyspace__ = "test"
    __table_name__ = "measurements_by_sensor_day"

    sensor_id = columns.Text(partition_key=True)
    day = columns.Date(primary_key=True, partition_key=True)
    ts = columns.DateTime(primary_key=True,
clustering_order="DESC")
    value = columns.Double()

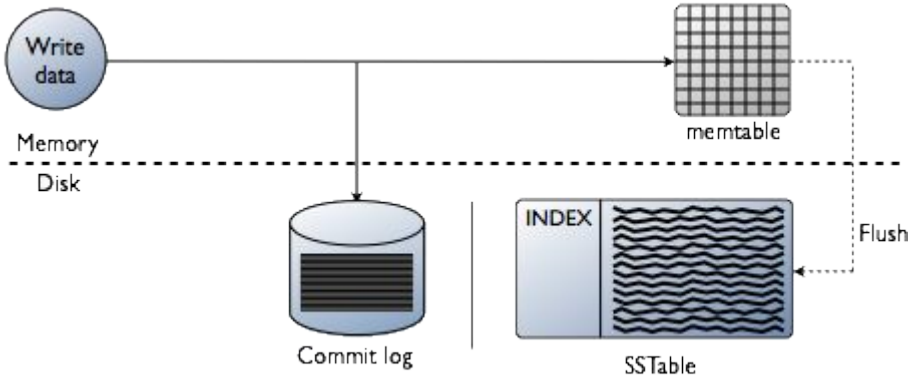
rows = (
    MeasurementsBySensorDay.objects(
        sensor_id="sensor-17",
        day=date(2026, 4, 8)
    )
    .filter(ts__gte=datetime(2026, 4, 8, 12, 0, 0))
    .order_by("-ts")
    .limit(10)
)
```

```
SELECT * FROM measurements_by_sensor_day
WHERE sensor_id = 'sensor-17'
AND day = '2026-04-08'
AND ts >= '2026-04-08 12:00:00'
ORDER BY ts DESC
LIMIT 10;
```

More on Internals of



Writes



- A write is atomic at the row level
- When a write occurs:
  - Data are written to the **memtable** in memory
  - The write is appended to the **commit log** on disk
  - When the memtable reaches a threshold, it is scheduled for flush
  - The memtable is flushed to an **SSTable** on disk
  - The corresponding commit log data can then be removed
- Key point:
  - The commit log provides durability
  - The memtable makes writes fast

Writes

- Writes
  - Memtables and SSTables are maintained per table
  - SSTables are **immutable**
  - A **row** may be spread **across multiple SSTables**
    - Reads may need to combine data from:
      - multiple SSTables
      - unflushed memtables
- SSTable lookup structures
  - Partition index — maps primary keys to row positions in the data file
  - Partition summary — a sampled structure used to speed up lookup
    - By default 1 primary key out of every 128

Write Example

```
write (k1, c1:v1)
write (k2, c1:v1 c2:v2)
write (k1, c1:v4 c3:v3 c2:v2)
```

Memtable:

```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```

Data are stored in sorted order
Column names are not repeated

Commit log:

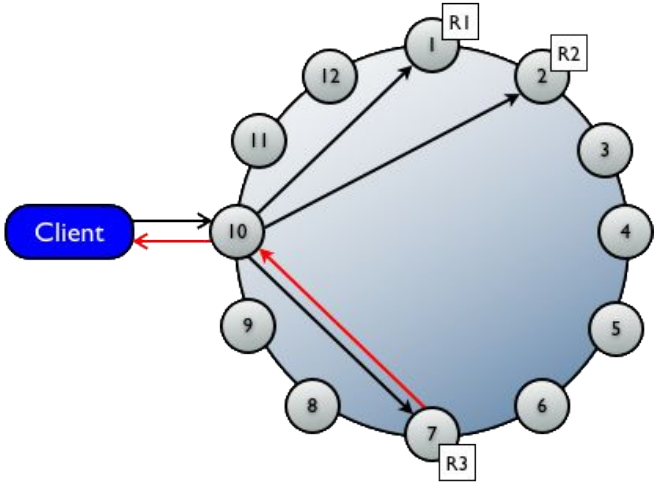
```
k1, c1:v1
k2, c1:v1 c2:v2
k1, c1:v4 c3:v3 c2:v2
```

SSTable:

```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```

After flushing the memtable to disk

Write Request



- On write
  - A client can contact any node
  - That node acts as the **coordinator**
    - The coordinator forwards the write to the replicas that own the row
  - The **write consistency level** determines how many replicas must acknowledge success
  - A replica acknowledges success after writing to the commit log and memtable
- Example: With replication factor 3 and consistency level ONE, the coordinator returns success after the first successful replica response

Read Request

- On read:
  - A **coordinator** sends read requests to replicas according to the **read consistency** level
    - It typically contacts the replicas that currently respond fastest
  - Returned data are compared for consistency
  - If replicas differ, Cassandra uses the most recent version based on timestamps
- Read repair:
  - Replica differences may be repaired in the background

Consistency Levels and Quorums

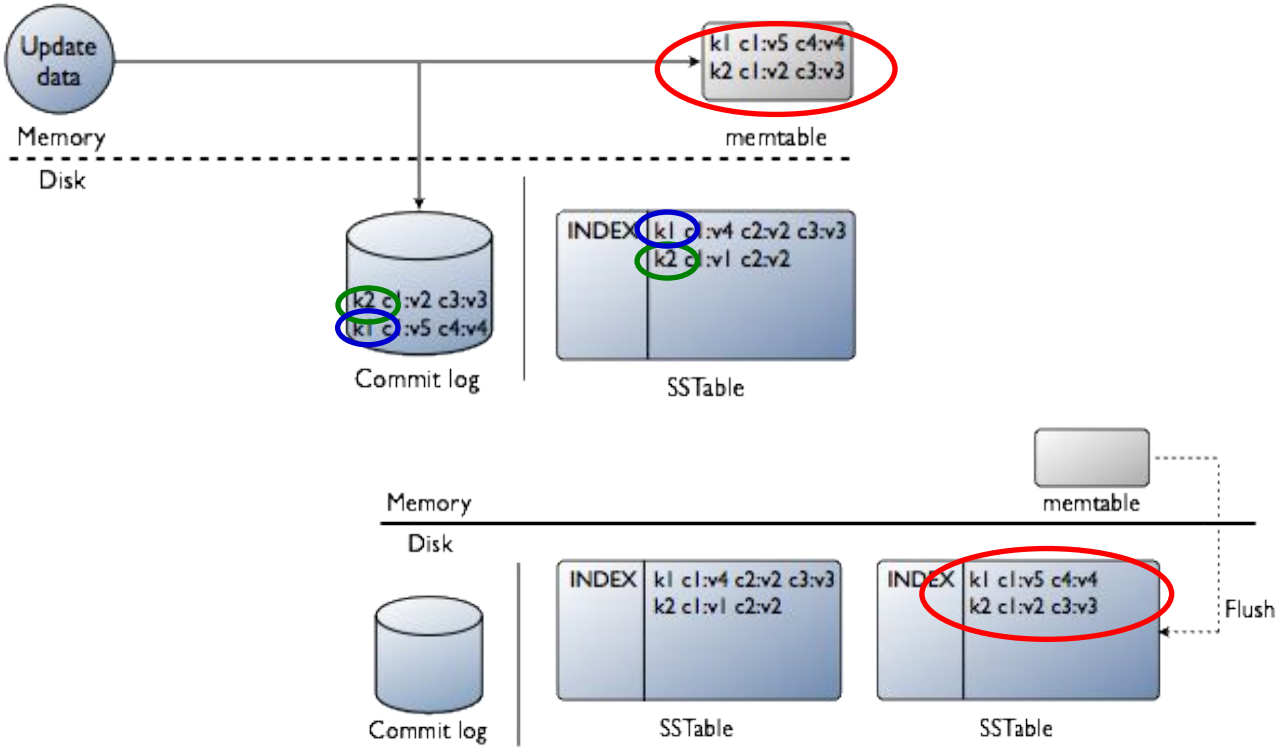
- Cassandra supports different **consistency levels** for writes and reads
  - The minimum number of replicas that must acknowledge a write (W)
  - The minimum number of replicas that must respond to a read (R)
- User-defined: ONE, TWO, QUORUM (a majority of replicas must respond), ALL, …
- For replication factor RF: **QUORUM > RF / 2**
- Example: RF = 3, Write QUORUM = 2, Read QUORUM = 2
  - If **W + R > RF**, the read and write sets must overlap
    - At least one replica participates in both the successful write and the read
    - Read can see the latest successful write

https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlConfigConsistency.html

Updates

- **Insert** and **update** operations are handled similarly
- Cassandra uses **upsert** semantics
  - The operation inserts new data or updates existing data
- Multiple columns can be written at the same time
- Newer values overwrite older ones only if they have a more recent timestamp
- Older versions may remain in SSTables until they are merged by compaction
- Note: Timestamps should be consistent across clients

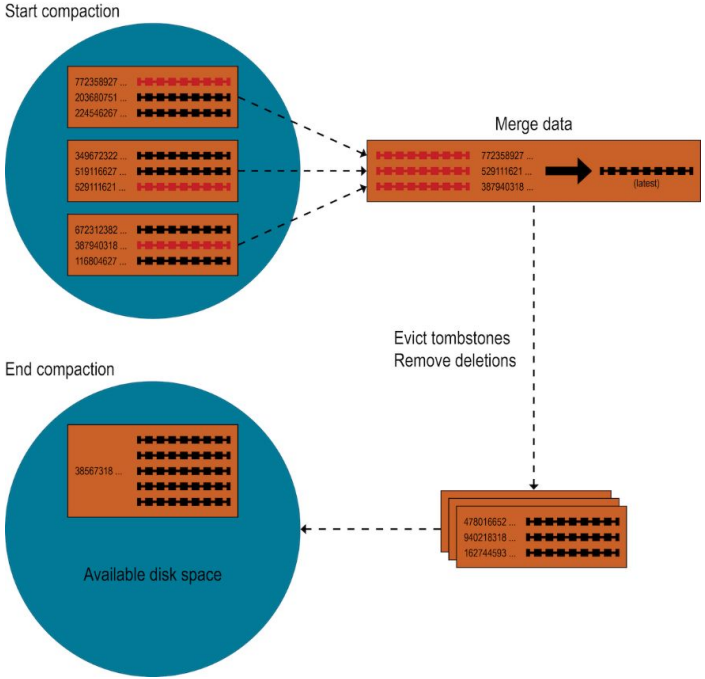
Updates


Update data
Memory
Disk
k1 c1:v5 c4:v4
k2 c1:v2 c3:v3
memtable
k2 c :v2 c3:v3
k1 c :v5 c4:v4
Commit log
INDEX
k1 c :v4 c2:v2 c3:v3
k2 c :v1 c2:v2
SSTable
memtable
Memory
Disk
Flush
Commit log
INDEX
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
SSTable
INDEX
k1 c1:v5 c4:v4
k2 c1:v2 c3:v3
SSTable

Deletes

- Deleting a row means writing deletion markers for its data
  - SSTables are immutable - deleted data are not removed immediately
  - Cassandra writes a **tombstone**
    - a marker indicating that the data were deleted
- Tombstones remain for a configurable **grace period**
- During **compaction**, tombstones and deleted data can be permanently removed
  - Triggered when there are more than `min_threshold` SSTables
- Why this matters
  - Tombstones help propagate deletes to replicas that may have been temporarily unavailable
  - If a node misses a delete for too long, deleted data may reappear
  - Regular repair helps keep replicas consistent

Compaction Process



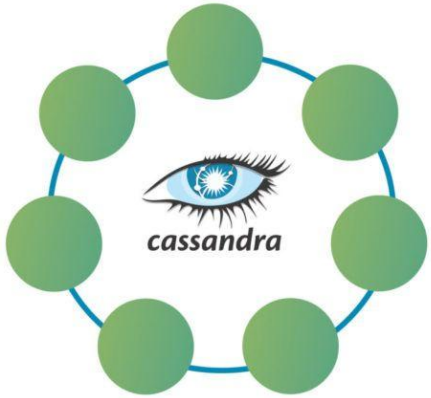
- Cassandra does not update or delete data in place
  - Newer row versions and tombstones = new SSTables
- Over time, SSTables are merged by **compaction**
- Compaction steps
  - Merge rows from multiple SSTables by partition key
  - Keep the most recent values based on timestamps
  - Remove expired tombstones and deleted data when possible
  - Write the merged result into a new SSTable
  - Delete old SSTable files when they are no longer needed
- Key point: Compaction improves read efficiency and reclaims storage space

Architecture

- Peer-to-peer distributed system
  - Coordinator = any node responsible for a particular client operation
- Key components:
  - **Partitioner** – determines how to distribute the data across the nodes
  - **Virtual nodes** – assign data ownership to physical nodes
  - **Replica placement strategy** – determines which nodes to place replicas on
  - **Gossip** – exchanging information across the cluster

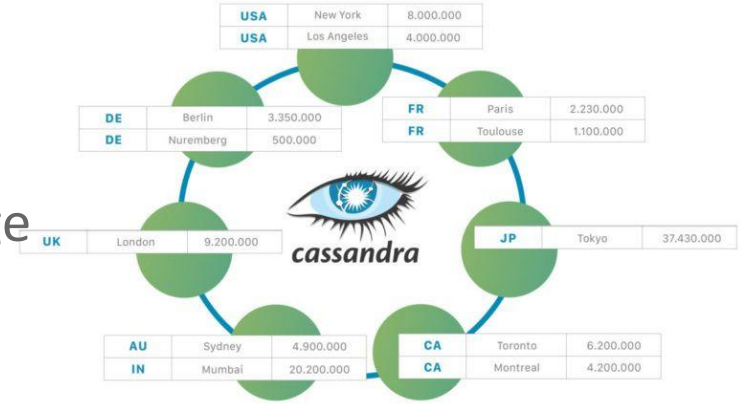
Partitioning

- The **partition key** is transformed into a **token**
- The token is a numeric value produced by the partitioner
  - Hash function
- Tokens determine positions on the logical ring
- Each node is responsible for a range of tokens



COUNTRY	CITY	POPULATION
USA	New York	8.000.000
USA	Los Angeles	4.000.000
FR	Paris	2.230.000
DE	Berlin	3.350.000
UK	London	9.200.000
AU	Sydney	4.900.000
DE	Nuremberg	500.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
FR	Toulouse	1.100.000
JP	Tokyo	37.430.000
IN	Mumbai	20.200.000

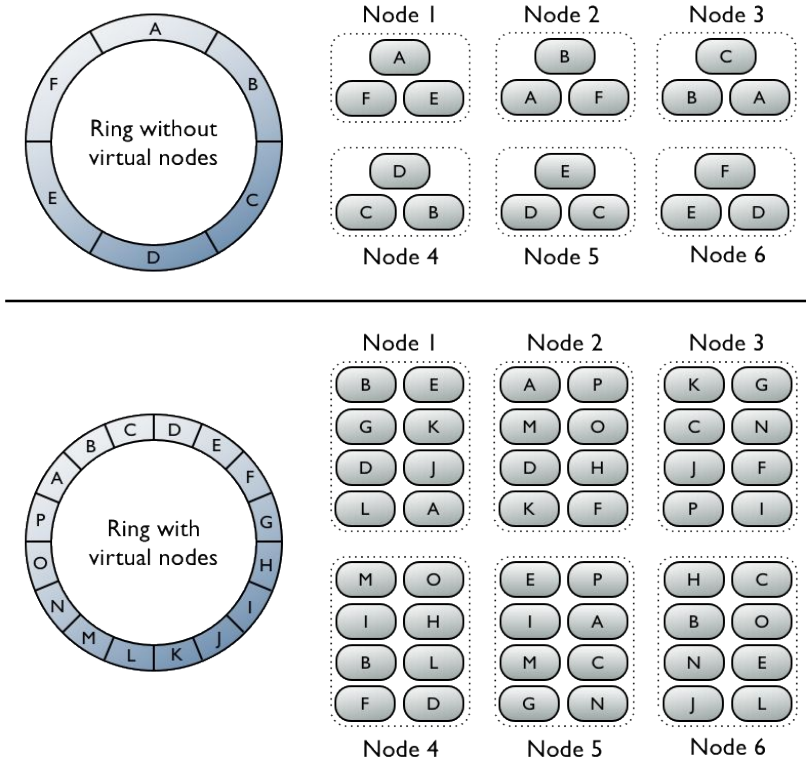
Partition Key



Virtual Nodes

- **One physical node** is responsible for **many virtual nodes**
- Each virtual node owns a small token range
  - Leads to more even data distribution
- It reduces the impact of node addition or failure
- Rebalancing becomes easier

Example: replication factor = 3



Replication

- Cassandra stores each partition on multiple nodes
  - Determined by the **replication factor**
  - All replicas are equally important
- **Replica placement strategies**
  - Tokens form a logical circular space
  - Each node owns one or more token ranges
  - Replicas can be placed on subsequent nodes in the ring
  - This supports scalability and fault tolerance

Replication

SimpleStrategy

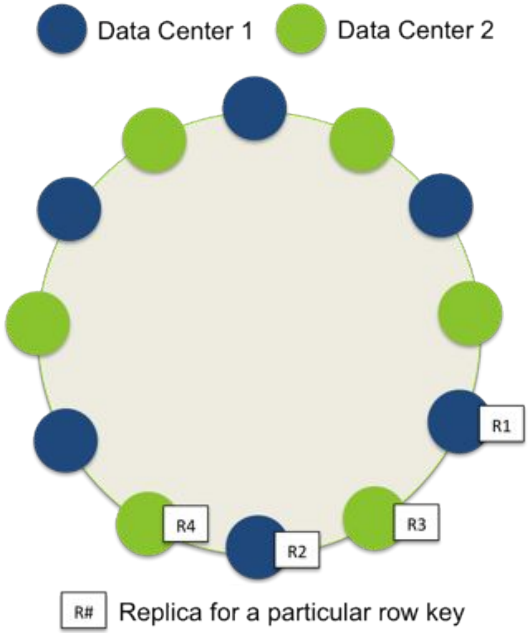
- Places the first replica on a node determined by the partitioner (hash function)
- Additional replicas are placed on the next nodes clockwise in the ring
- For a single data center only

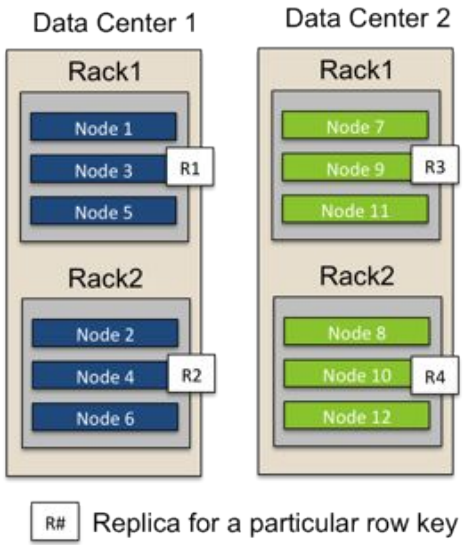
- We can divide the nodes into (optional **racks** forming) **data centers**
    - Collection of related nodes, physical or virtual

Replication

NetworkTopologyStrategy

- Places replicas within a data center
  - We set number of replicas per a data center
- The first replica is placed according to the partitioner
- Additional replicas are placed by walking the ring clockwise until a node in a different rack is found
  - Motivation: nodes in the same rack often fail at the same
- If no such node exists, additional replicas are placed in different nodes in the same rack


Data Center 1
Data Center 2
R1
R2
R3
R4
R#
Replica for a particular row key


Data Center 1
Data Center 2
Rack1
Node 1
Node 3
Node 5
R1
Rack2
Node 2
Node 4
Node 6
R2
Rack1
Node 7
Node 9
Node 11
R3
Rack2
Node 8
Node 10
Node 12
R4
R#
Replica for a particular row key

Gossip

- Gossip process
  - Runs every second
  - Exchanges state messages with up to 3 other nodes in the cluster
  - Enables to detect failures
- Gossiped message:
  - Information about a gossiping node + other nodes that it knows about
  - Acquired:
    - Directly = by direct communication
    - Indirectly = second hand, third hand, …
  - Has a version
    - Older information is overwritten with the most current state

References

- Eric Redmond – Jim R. Wilson: **Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement**
- Pramod J. Sadalage – Martin Fowler: **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**

- Cassandra:
  - Getting Started: https://cassandra.apache.org/doc/latest/getting_started/
  - CQL: https://cassandra.apache.org/doc/latest/cql/index.html

Lecture 06

Graph Data Management

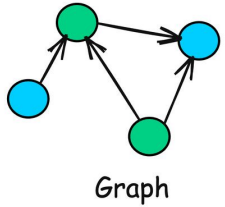
NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Graph Data Management



- Also: property graph databases, graph stores
- Store data as **nodes** and **relationships**
  - Entities, relationships, properties
  - Data are naturally modeled as a graph
- Schema is often flexible (lightweight)
- Typical operations:
  - Find a node or neighborhood
  - Traverse relationships
  - Match structural patterns
  - …
- Joins are replaced by traversals

JanusGraph

NebulaGraph

TigerGraph

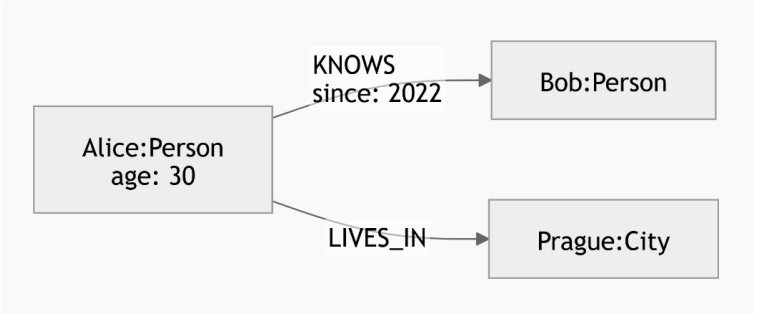
neo4j

Property Graph Model

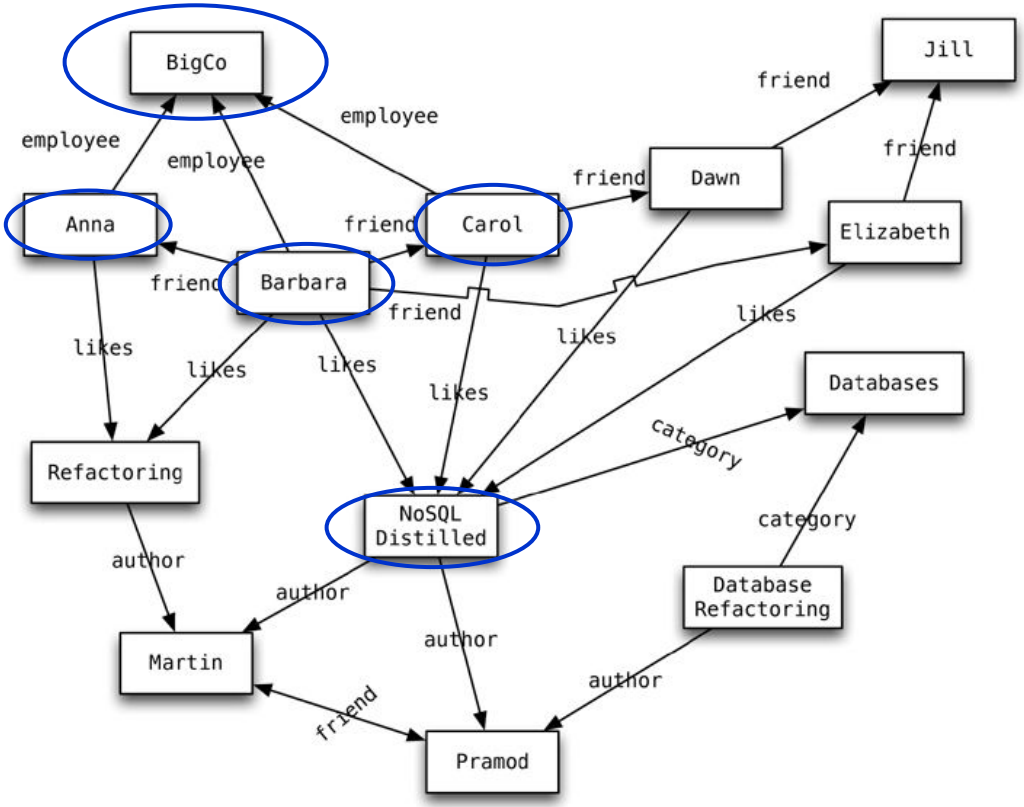
Two variants:
- **Labeled Property Graph** (LPG) – schema-later (Neo4j)
- **Typed Property Graph** (TPG) – schema-first

```
(Alice:Person {age: 30})-[:KNOWS {since: 2022}]->(Bob:Person)
(Alice)-[:LIVES_IN]->(Prague:City)
```

- Node
  - identifier
  - **labels**
    - Group nodes of similar role
    - LPG-specific concept
  - properties
- Relationship
  - directed edge between two nodes
  - **type**
  - properties
- Property
  - key-value pair, e.g. age: 30, since: 2022



Example - Graph Traversal



Relational vs. Graph Databases

Relational DBs	Graph DBs
data in tables	data as nodes and relationships
relationships via foreign keys	relationships stored directly
multi-hop queries often need joins	multi-hop queries use traversals
good for structured tabular data	good for highly interconnected data

Graph databases are especially useful when relationships are central

Person

id	name	age
1	Alice	30
2	Bob	NULL

City

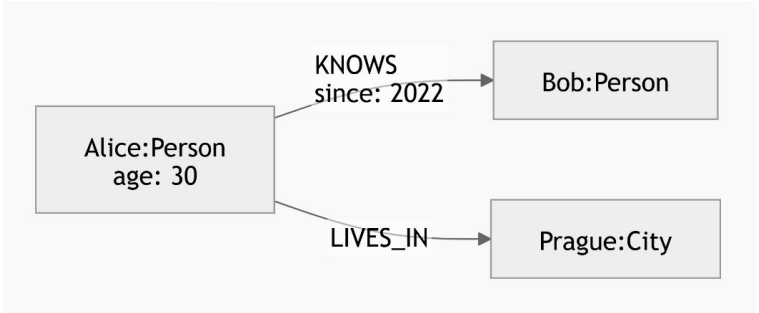
id	name
1	Prague

Knows

person1_id	person2_id	since
1	2	2022

LivesIn

person_id	city_id
1	1

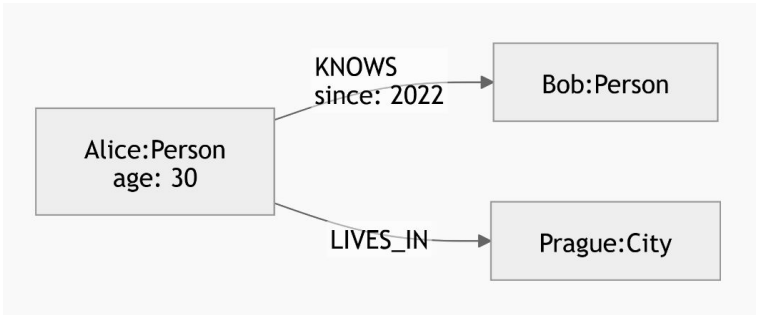


Document vs. Graph Databases

Document DBs	Graph DBs
data stored as documents	data stored as nodes and relationships
good for hierarchical and nested data	good for highly interconnected data
natural representation as a tree	natural representation as a graph
references are possible, but not central	relationships are first-class

```
{
  "name": "Alice",
  "age": 30,
  "lives_in": {
    "name": "Prague"
  },
  "knows": [
    {
      "name": "Bob",
      "since": 2022
    }
  ]
}
```

Graph databases are a better fit when links between entities are important



Wide-Column vs. Graph Databases

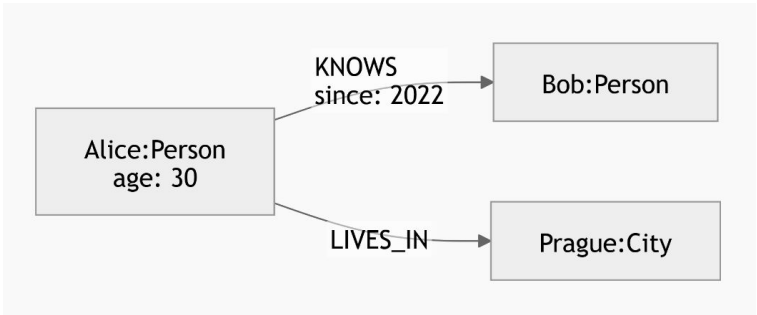
Wide-Column DBs	Graph DBs
data grouped by row key and columns	data stored as nodes and relationships
optimized for lookup by key	optimized for traversal of connected data
good for large-scale sparse data	good for relationship-centric queries
weaker support for explicit relationships	relationships stored directly

Wide-column databases optimize access by key, while graph databases optimize access by relationships

Wide-Column Table

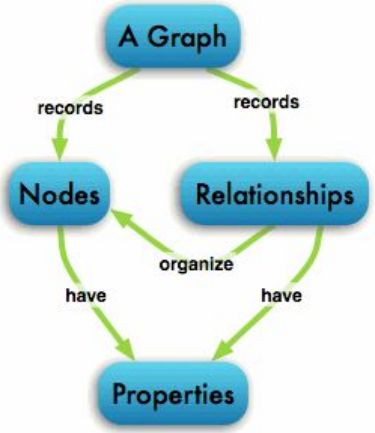
row key	age	lives_in	knows:Bob
Alice	30	Prague	2022

row key	name
Bob	Bob



Neo4j

- One of the best-known property graph databases
  - Open source
  - Written in: Java
    - Cross-platform
- Initial release: 2007
- Data: Nodes, relationships, labels, and properties
- Data model: labeled property graph (schema-optional)
- Queries: mainly using **Cypher**
- Supports official drivers for multiple languages
  - Including Python



https://neo4j.com/

Data Model



- Fundamental units: **nodes** and **relationships**
- Both can have **properties**
  - Stored as key–value pairs
    - Value - simple type or array of simple types
- Relationships are directed
  - Have type, start node, and end node
  - Can be recursive
- In queries, direction can be used or ignored

Missing value is usually represented by an absent property

Type	Description	Value range
boolean		true/false
byte	8-bit integer	-128 to 127, inclusive
short	16-bit integer	-32768 to 32767, inclusive
int	32-bit integer	-2147483648 to 2147483647, inclusive
long	64-bit integer	-9223372036854775808 to 9223372036854775807, inclusive
float	32-bit IEEE 754 floating-point number	
double	64-bit IEEE 754 floating-point number	
char	16-bit unsigned integers representing Unicode characters	u0000 to uffff (0 to 65535)
String	sequence of Unicode characters	

Node Labels/Edge Types



- Nodes can have zero, one, or multiple labels
  - Used for logical grouping and for selecting nodes in queries
  - Examples: `:Person`, `:City`, `:Student`
- Relationships always have exactly one type
  - Examples: `:KNOWS`, `:LIVES_IN`

```
(Alice:Person:Student)-[:LIVES_IN]->(Prague:City)
(Alice)-[:KNOWS]->(Bob:Person)
```

Cypher

- Query language for Neo4j
- Used for querying and updating
- Declarative
  - Describe what to find
  - Inspired by SQL and SPARQL
- Based on **graph patterns**
- Designed to be readable



https://neo4j.com/docs/getting-started/cypher/

Basic Cypher Clauses



- `MATCH` — find a pattern
- `WHERE` — add conditions
- `RETURN` — return results
- `CREATE` — create nodes/relationships
- `SET` — update properties/labels
- `DELETE` / `DETACH DELETE` — remove data
- `REMOVE` — removes label/property
- `WITH` — pass intermediate results

CREATE and SET



```
CREATE (a:Person {name: 'Alice', age: 30})
CREATE (b:Person {name: 'Bob'})
CREATE (c:City {name: 'Prague'})
CREATE (a)-[:KNOWS {since: 2022}]->(b)
CREATE (a)-[:LIVES_IN]->(c)

MATCH (a:Person {name: 'Alice'})
SET a.age = 31
RETURN a
```

… updates an existing property (upsert)

```
MATCH (a:Person {name: 'Alice'})
SET a:Student
RETURN a
```

… adds a new label

Cypher uses patterns to create and update graph data

CREATE



```
CREATE (a:Person {name: 'Alice', age: 30})
       -[:LIVES_IN]->
       (:City {name: 'Prague'})

MATCH (a:Person {name: 'Alice'}),
      (b:Person {name: 'Bob'})
CREATE (a)-[:KNOWS {since: 2022}]->(b)

MATCH (a:Person {name: 'Alice'})
CREATE (a)-[:LIVES_IN]->(c:City {name: 'Brno'})
```

`MATCH` can find existing nodes, while `CREATE` can add both new nodes and new relationships.

MERGE



```
MERGE (a:Person {name: 'Alice'})

MATCH (a:Person {name: 'Alice'}),
      (c:City {name: 'Prague'})
MERGE (a)-[:LIVES_IN]->(c)

MERGE (a:Person {name: 'Alice'})-[:LIVES_IN]->(c:City {name: 'Prague'})

MERGE (a:Person {name: 'Alice', age: 30})

MERGE (a:Person {name: 'Alice'})
SET a.age = 30
```

`CREATE (a:Person {name: 'Alice'})` **creates a new node every time the query is executed,** `MERGE` **only if it does not exist**

(DETACH) DELETE, REMOVE



```
MATCH (a:Person {name: 'Alice'})-[r:KNOWS]->(b:Person {name: 'Bob'})
DELETE r

MATCH (b:Person {name: 'Bob'})
DETACH DELETE b

MATCH (a:Person {name: 'Alice'})
REMOVE a:Student
RETURN a

MATCH (a:Person {name: 'Alice'})
REMOVE a.age
RETURN a
```

A node with existing relationships cannot be deleted by `DELETE` alone

MATCH, WHERE, RETURN

```
MATCH (a:Person)-[:KNOWS]->(b:Person)
WHERE a.name = 'Alice'
RETURN b
```

```
+--------------------------------+
| b                              |
+--------------------------------+
| (:Person {name: 'Bob'})        |
+--------------------------------+
```

```
MATCH (a:Person)-[:KNOWS]->(:Person)-[:LIVES_IN]->(c:City)
WHERE a.name = 'Alice'
RETURN c.name
```

```
+----------+
| c.name   |
+----------+
| Prague   |
| Brno     |
+----------+
```

```
MATCH (a:Person)-[:KNOWS]->(b:Person)
WHERE a.name = 'Alice'
RETURN a.name, b.name AS friend
```

```
+--------+--------+
| a.name | friend |
+--------+--------+
| Alice  | Bob    |
| Alice  | Carol  |
+--------+--------+
```

DISTINCT, ORDER BY, LIMIT, SKIP

```
MATCH (a:Person)-[:KNOWS]->(:Person)-[:LIVES_IN]->(c:City)
WHERE a.name = 'Alice'
RETURN DISTINCT c.name AS city
ORDER BY city ASC
SKIP 1
LIMIT 5
```



Cypher can remove duplicates, sort results, skip selected rows, and limit the output size

Aggregation Functions



```
MATCH (a:Person)-[:KNOWS]->(b:Person)
RETURN a.name AS person, count(b) AS cnt
```

- Grouping is implicit
  - Non-aggregated expressions in `RETURN` define the groups
  - `a.name` is the grouping key
  - `count(b)` is evaluated for each value of the key

```
MATCH (a:Person)-[:KNOWS]->(b:Person)
RETURN count(b) AS number_of_friends
```

WITH



```
MATCH (a:Person)-[:KNOWS]->(b:Person)
WITH b, count(*) AS popularity
WHERE popularity >= 2
MATCH (b)-[:LIVES_IN]->(c:City)
RETURN b.name AS person, popularity, c.name AS city
```

- Used to pass **intermediate results** to the next part of a query
- It allows us to:
  - Carry selected variables forward
  - Rename values with AS
  - Use aggregation results in later steps

FOREACH



```
MATCH (a:Person)-[:KNOWS]->(b:Person)
WHERE a.name = 'Alice'
WITH collect(b) AS friends
FOREACH (f IN friends | SET f.visited = true)
```

- `MATCH` returns multiple rows with different values of `b`
- `collect(b)` combines them into a single list
- `FOREACH` iterates over that list

Useful when we already have a list and want to apply an update to each element

Returning Paths



```
MATCH p = (a:Person)-[:KNOWS]->(b:Person)
RETURN p
```

```
+-------------------------------------------------------------+
| p                                                           |
+-------------------------------------------------------------+
| (:Person {name: 'Alice'})-[:KNOWS]->(:Person {name: 'Bob'}) |
| (:Person {name: 'Bob'})-[:KNOWS]->(:Person {name: 'Carol'}) |
+-------------------------------------------------------------+
```

- `RETURN p` returns a path
  - In Neo4j Browser, the result can be visualized as a graph
    - **Graph view** vs **table view**
- Useful for exploring connected data

Cypher results do not have to be only tables; they can also be returned as paths

MATCH vs. OPTIONAL MATCH



- MATCH requires the pattern to exist
- OPTIONAL MATCH keeps the row even if the pattern is missing
- Similar to inner join vs. left outer join
  - Missing values are returned as null

```
MATCH (a:Person {name: 'Alice'})
OPTIONAL MATCH (a)-[:LIVES_IN]->(c:City)
RETURN a.name AS person, c.name AS city
```

Direction in Patterns



```
MATCH (a)-[:KNOWS]->(b)
RETURN a, b

MATCH (a)-[:KNOWS]-(b)
RETURN a, b
```

- -[]-> uses the stored direction
- -[]- ignores direction in the match
- Useful when direction is not important for the query

Cypher - Advanced Constructs



- More aggregation functions: `sum()`, `avg()`, `min()`, `max()`
- Predicates over lists: `all()`, `any()`, `none()`, `single()`
- Path functions: `nodes(p)`, `relationships(p)`, `length(p)`
- Relationship metadata: `type(r)`
- Node metadata: `labels(n)`, `keys(n)`

… and many others

https://neo4j.com/docs/cypher-manual/current/functions/

Python + Cypher

- Python sends Cypher queries to Neo4j
- Query parameters can be passed from application code
- Results can be processed programmatically
- Useful for scripts, web applications, and data pipelines

```
from neo4j import GraphDatabase

URI = "neo4j://localhost:7687"
AUTH = ("neo4j", "password")

driver = GraphDatabase.driver(URI, auth=AUTH)

driver.execute_query("""
     CREATE (a:Person {name: $name, age: $age})
     CREATE (b:Person {name: $friend})
     CREATE (c:City {name: $city})
     CREATE (a)-[:KNOWS {since: $since}]->(b)
     CREATE (a)-[:LIVES_IN]->(c)
""",
name="Alice",
age=30,
friend="Bob",
city="Prague",
since=2022,
database_="neo4j"
)

driver.close()
```

Python + Cypher

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver(
     "neo4j://localhost:7687",
     auth=("neo4j", "password")
)

records, summary, keys = driver.execute_query(
     """
     MATCH (a:Person)-[:LIVES_IN]->(c:City)
     RETURN a.name AS person, c.name AS city
     """,
     database_="neo4j"
)

for record in records:
     print(record["person"], record["city"])

driver.close()
```

Result: `Alice Prague`

neomodel

```
from neomodel import StructuredNode, StringProperty, RelationshipTo

class Person(StructuredNode):
    name = StringProperty()
    knows = RelationshipTo('Person', 'KNOWS')

alice = Person(name="Alice").save()
bob = Person(name="Bob").save()
alice.knows.connect(bob)
```

Low-level:

- Cypher via driver (full control)
- Explicit queries

High-level:

- OGM (Object Graph Mapping)
  - Similar to ORM
- Work with Python objects instead of queries
  - maps objects ↔ graph
  - less control, more convenience

https://neomodel.readthedocs.io/en/latest/

Path and Traversal



- **Path** = one or more nodes connected by relationships
  - Can be returned directly as a query result
- **Traversal** = visiting nodes by following relationships
  - Follows some rules:
    - relationship type
    - relationship direction
    - traversal order (e.g., BFS / DFS)
  - depth or stopping condition

Traversal Framework



- Not a second query language
  - A framework for explicit graph traversal
- Defines how the graph is explored:
  - start node(s)
  - relationship type(s)
  - relationship direction
  - traversal order - BFS / DFS
  - stopping condition / depth
  - uniqueness - whether a node / relationship / path may be visited again
  - evaluator - whether to include the current node in the result, and whether to continue or stop from it
- Cypher describes what to find
- Traversal framework controls how to traverse the graph

Cypher = what to find

Traversal framework = how to explore

BFS Traversal

- BFS = breadth-first search
  - Explores the graph **level by level**
  - One possible traversal order
- First visits direct neighbors
- Then neighbors of neighbors
- …
- Useful for finding the closest reachable nodes
- BFS vs. DFS
  - BFS is useful for nearest nodes
  - DFS is useful for deep exploration of one path

```
Start at Alice
follow KNOWS
direction: outgoing
order: BFS
depth: up to 2
uniqueness: NODE_GLOBAL
evaluator: include nodes up to depth 2
```

Different traversal orders may visit the same graph in a different sequence

Neo4j Traversal Framework – Java API

- Explicit traversal in Java
- Define how the graph should be explored
- Configure:
  - start node
  - relationship type + direction
  - order (BFS / DFS)
  - evaluator
  - uniqueness
- Then explicitly call `traverse(startNode)`

```
Node alice = getNodeByName("Alice");

TraversalDescription td = tx.traversalDescription()
     .breadthFirst()
     .relationships(Rels.KNOWS, Direction.OUTGOING)
     .evaluator(Evaluators.toDepth(2))
     .uniqueness(Uniqueness.NODE_GLOBAL);

Traverser traverser = td.traverse(alice);

for (Path path : traverser) {
     System.out.println(
     path.endNode().getProperty("name") +
          " at depth " + path.length()
     );
}
```

Unlike Python + Cypher, here we do not just submit a query — we directly control the traversal strategy

Limits of Cypher

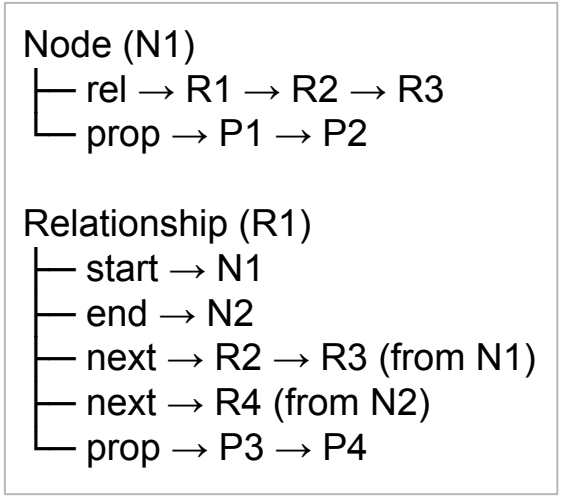
- Cypher cannot easily express:
  - Explicit BFS vs DFS
  - Custom pruning strategies
  - Fine-grained early stopping
  - Complex stateful traversal logic
- Example limitation
  - Prune at hub nodes: stop expanding from nodes with degree >100
  - Path-dependent traversal: continue only while accumulated path cost remains below a threshold
- This is where traversal API is needed

Cypher is weaker when traversal decisions depend on the current path or intermediate state

More on Internals of 

Data on Disk

- Nodes, relationships, properties stored as records
  - Have fixed size
  - Connected using references (pointers)
- Node contains:
  - pointer to first relationship
  - pointer to first property
- Relationship contains:
  - pointer to start node
  - pointer to end node
  - pointer to next relationship (for both nodes)
  - pointer to first property



Traversal = following pointers

Why is Graph Traversal Fast?

- Traversal = pointer chasing
  - No joins
  - No global scans
  - Only local pointer hops
- Cost depends on visited nodes, not on the size of the graph

Relational DB	→ join
Graph DB	→ pointer hop

Relationships are stored, not computed

Data on Disk

- Note: Neo4j is a schema-less database
  - Fixed record lengths + offsets in files → **direct access** (no scanning)
- Several types of files to store the data

Store File	Record size	Contents
neostore.nodestore.db	15 B	Nodes
neostore.relationshipstore.db	34 B	Relationships
neostore.propertystore.db	41 B	Properties for nodes and relationships
neostore.propertystore.db.strings	128 B	Values of string properties
neostore.propertystore.db.arrays	128 B	Values of array properties
Indexed Property	1/3 * AVG(X)	Each index entry is approximately 1/3 of the average property value size

https://neo4j.com/developer/kb/understanding-data-on-disk/

Transaction Management

- Transaction = a unit of work executed atomically
- In neo4j:
  - Every query runs inside a transaction
  - Support for **ACID** properties
- Transactions can be:
  - Implicit (auto-commit)
  - Explicit
- Why transactions matter?
  - Ensure consistency
  - Allow **commit** on success (make changes permanent) and **rollback** on error (discard all changes)
  - Isolate concurrent updates

Either all changes are applied, or none of them are

Implicit vs Explicit Transactions

- Implicit transaction
  - created automatically for a single query
  - usually committed automatically if the query succeeds
- Explicit transaction
  - started and controlled by the application
  - can contain multiple queries / updates
  - committed explicitly
  - rolled back if an error occurs
- Use implicit when one simple query is enough
- Use explicit when several steps must succeed together

```
create Alice
create Bob
create KNOWS
```

If step 3 fails:

- with transaction → nothing is stored
- without transaction semantics → partial update would remain

Implicit = convenience, Explicit = control

Java vs Python

Java

- Explicit transaction object
- Application controls begin / commit / rollback
- Suitable for fine-grained control

Python

- Transaction managed by the driver
- Callback receives the current transaction as `tx`
- Rollback is automatic if the callback fails

```
try (Transaction tx = db.beginTx()) {

    Node alice = tx.createNode(Label.label("Person"));
    alice.setProperty("name", "Alice");

    tx.commit();
}
```

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver(
    "neo4j://localhost:7687",
    auth=("neo4j", "password")
)

def create_person(tx, name):
    tx.run("CREATE (:Person {name: $name})",name=name)

with driver.session() as session:
    session.execute_write(create_person, "Alice")

driver.close()
```

Java: we create and control the transaction
Python: the driver manages the transaction

Java vs Python

```java
try (Transaction tx = db.beginTx()) {
    Node alice = tx.createNode(Label.label("Person"));
    alice.setProperty("name", "Alice");

    Node bob = tx.createNode(Label.label("Person"));
    bob.setProperty("name", "Bob");

    throw new RuntimeException("Something went wrong");
    // tx.commit();
}
```

```python
from neo4j import GraphDatabase

def create_two_people(tx):
   tx.run("CREATE (:Person {name: 'Alice'})")
   tx.run("CREATE (:Person {name: 'Bob'})")
   raise Exception("Something went wrong")

with driver.session() as session:
   try:
      session.execute_write(create_two_people)
   except Exception:
      print("Transaction rolled back")
```

Because the transaction failed, neither Alice nor Bob is stored

Why Do We Need Indexes?

- Without index: Database may need to scan many nodes
- With index: Database can find matching nodes directly
- Typical use:
  - find a person by name
  - find all movies from a given year
- Index
  - Used to find entry points in the graph
  - Based on property values
- Traversal
  - Used to explore connected data
  - Follows relationships

```
MATCH (p:Person)
WHERE p.name = 'Alice'
RETURN p
```

```
CREATE INDEX person_name_index
FOR (p:Person)
ON (p.name)
```

```
MATCH (p:Person {name: 'Alice'})
   -[:KNOWS]->(f:Person)
RETURN f.name
```

Traversal follows relationships, but an index helps us find the starting node

Neo4j Cluster: Replication, not Sharding

- Basic Neo4j cluster
  - one database is replicated across multiple nodes
    - one node acts as **leader** for writes
    - other nodes (**followers**) keep replicas and can serve read queries
  - main goal: high availability, failover, and read scaling
- What the basic cluster does not provide
  - The graph is not automatically partitioned into independent pieces
  - Clustering means replication, not horizontal partitioning/sharding

Cluster = same data on multiple servers for availability and read scaling

Neo4j Sharding with Composite Databases

- Sharding in Neo4j
  - handled through **composite databases**
  - one logical graph can be split into multiple databases (shards)
    - User-defined shards (e.g., one for each region, year, …)
  - queries can access these shards through a single entry point
- Relationships do not cross database boundaries
  - Cross-database links are modeled via proxy nodes and shared IDs
- Composite database
  - does not store data itself
  - contains aliases to local or remote constituent databases
  - enables a single Cypher query to read across multiple graphs/databases
- Only in Enterprise Edition

Fabric = different parts of the graph in different databases, queried as one logical whole

Why is Graph Sharding Difficult?

- A graph contains vertices, edges, and properties
  - We must decide what to shard:
- Two common approaches:
  - edge cut: vertices are partitioned, edges cross partitions (e.g., NebulaGraph)
  - vertex cut: edges are partitioned, high-degree vertices are shared (e.g., JanusGraph)
- Problem:
  - traversal crosses partitions
  - pointer hops become network hops
  - performance depends on communication

Sharding breaks locality → traversal becomes expensive

Beyond Property Graphs

Graph Query Paradigms

Declarative (Cypher, SPARQL)

- Describe what to find
- Database decides how to execute
- Pattern matching

Procedural (Gremlin)

- Describe how to traverse the graph
- Explicit sequence of steps
- Full control over traversal

Declarative = WHAT
Procedural = HOW

Gremlin



- Gremlin = traversal-based graph query language
  - Part of Apache TinkerPop
    - A vendor-neutral standard for graph computing
  - Works on property graphs
  - Queries are expressed as traversals
- Gremlin is procedural
  - We describe how to traverse the graph step by step
- Gremlin as pipeline (data flows through steps)

https://tinkerpop.apache.org/gremlin.html

Gremlin - Examples




```
g.V().has("name","Alice")

g.V().has("name","Alice").out("KNOWS")

g.V().has("name","Alice").out("KNOWS").values("name")

g.V().has("name","Alice").out("KNOWS").count()

g.V().has("name","Alice").out("KNOWS").out("KNOWS")

g.V().hasLabel("Person").has("age", gt(30)).values("name")
```

- selection
- traversal
- projection
- aggregation
- multi-hop
- filtering

Cypher: declarative (what)
Gremlin: procedural (how)

Property Graph vs RDF

```
SELECT ?f
WHERE {
   :Alice :knows ?f .
}
```

Property Graph (Neo4j)

- Nodes and relationships
- Both can have properties
- Relationships have types and direction
- Nodes have labels

RDF (Resource Description Framework)

- Data as triples
- Subject – predicate – object
- No explicit properties on nodes or edges
  - Uniform representation

```
(Alice)-[:KNOWS {since: 2022}]->(Bob)
```

```
Alice – knows – Bob
Alice – knowsSince – 2022
```

RDF flattens everything into triples; property graphs keep richer structure

When is RDF Useful?

- RDF / triple stores are useful when:
  - Data from many sources must be **integrated**
  - A **standard representation** is needed
  - We work with **knowledge graphs**
  - **Interoperability** is more important than compact structure
- Typical use cases
  - Semantic web
  - Linked open data
  - Enterprise knowledge graphs
- RDF graph: more uniform, more standardised
- Property graph: more natural for traversal and application queries

References

- Neo4j http://www.neo4j.org/
- Neo4j Docs https://neo4j.com/docs/
- RDF https://www.w3.org/RDF/
- SPARQL https://www.w3.org/2001/sw/wiki/SPARQL
- Sherif Sakr - Eric Pardede: Graph Data Management: Techniques and Applications

Lecture 07

Search and Information Retrieval Systems

NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Search Engines









- Designed for **retrieval-oriented workloads**
  - relevance-based search
  - full-text search
  - synonym and fuzzy search
- Compared to relational DBMSs
  - no rigid structural requirements
  - data may be structured, semi-structured, or unstructured
  - limited emphasis on relations, constraints, joins, and transactions
- Compared to NoSQL DBMSs
  - optimized primarily for searching, not editing
  - support ranking, stemming, grouping, filtering, geospatial search, analytics, …
- Typically built for **large-scale distributed processing**

Sometimes denoted as search engine data management systems

Why Search Is Different from Databases?

- Database queries usually ask for exactly matching data
  - equality, joins, aggregation, consistency
- Search queries usually ask for **relevant answers**
  - partial, approximate, ranked results
- The input is often **not cleanly structured**
  - documents, web pages, logs, emails
- Therefore, search systems optimize for
  - text processing, ranking, fast filtering over large collections, distributed execution

Goal: not only to find matching data, but to return the most useful results first

Typical Search Use Cases

- Full-text document retrieval
  - articles, reports, manuals, emails
- Product and website search
  - keyword search, faceted filtering, autocomplete
- Observability and log analytics
  - machine logs, traces, operational events
- Security and monitoring
  - anomaly investigation, incident search
- Analytical exploration over text-rich data
  - dashboards, aggregations, trends

The result is usually a ranked set of documents, not a single exact tuple

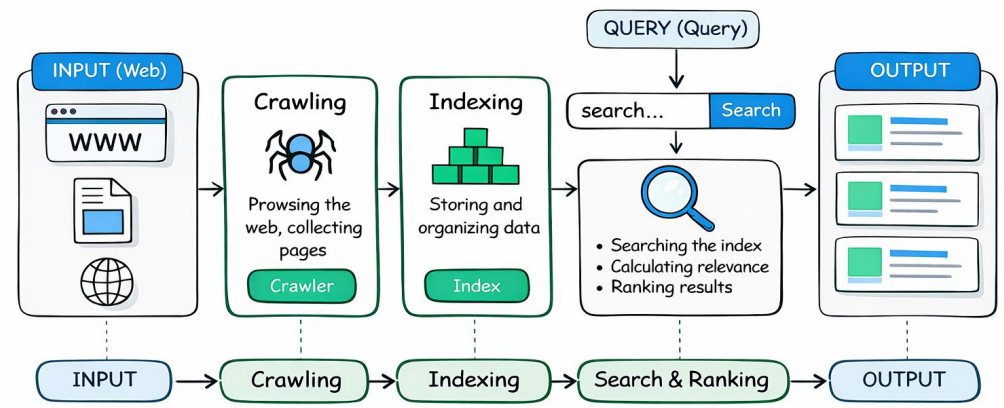
Information Retrieval Basics

- A discipline focused on searching in large collections of documents
- The basic objects are usually
  - documents
  - terms
  - queries
- The central problem is **relevance**
  - Which documents should be returned?
  - In what order?
- Unlike classical DB querying, IR must deal with
  - ambiguity
  - synonymy
  - morphology
  - noisy user input

A user often does not know the exact form of the stored data

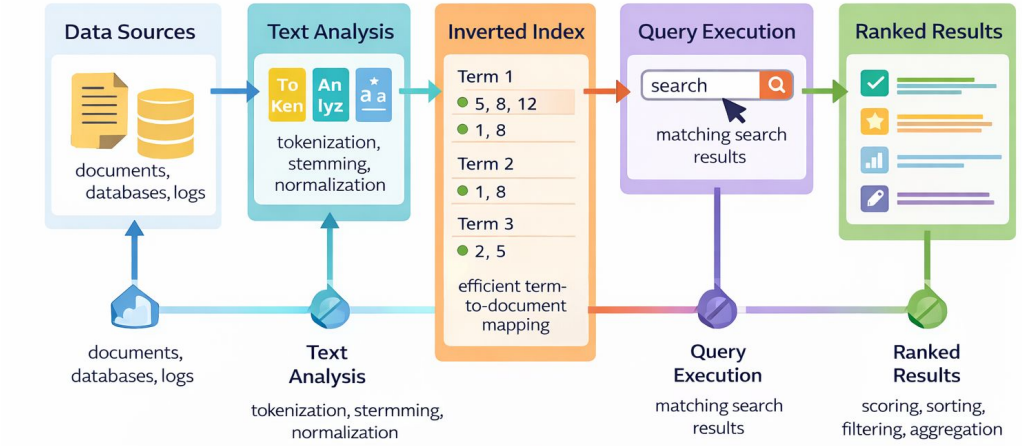
Core IR Tasks

- Document representation
  - how text is stored and indexed
- Query processing
  - how a user query is interpreted
- Matching
  - which documents satisfy the query
- Ranking
  - which matching documents are more relevant
- Result presentation
  - snippets, highlighting, facets, grouped results



Search Engine Pipeline

- Data ingestion
  - documents arrive from files, databases, logs, or streams
- Text analysis
  - tokenization, normalization, stemming, stop-word removal
- Index construction
  - creation of an inverted index
- Query execution
  - matching terms against the index
- Ranking and presentation
  - scoring, sorting, filtering, aggregation



Text Analysis

- Input text: `"Running systems are indexing the documents."`
  - Must be **transformed into a structured representation**

- After tokenization:
  - Split text into individual words/tokens
  - `["Running", "systems", "are", "indexing", "the", "documents"]`
- After normalization:
  - Convert tokens to a standard form
  - `["running", "systems", "are", "indexing", "the", "documents"]`
- After stemming:
  - Reduce words to a common root
  - `["run", "system", "are", "index", "the", "document"]`
- After stop-word removal:
  - Remove very common words with low information value
  - `["run", "system", "index", "document"]`

Inverted Index

- Why not scan all documents?
  - Brute-force search: check every document for every query
  - millions of documents → seconds/minutes per query
- Solution: we have to build an index
- Idea:
  - Instead of: document → words
  - **Inverted index**: term → documents containing the term
    - Core data structure of search engines

Documents:

D1: "Alice likes databases"
D2: "Alice likes graphs"
D3: "Graphs and databases"

Inverted index:

term	**documents**
alice	D1, D2
likes	D1, D2
databases	D1, D3
graphs	D2, D3

Query:

"databases" → D1, D3

Posting Lists & Positions

- For each term, the index stores a **posting list**
  - List of documents containing the term
- Optionally, it also stores positions
  - Where the term occurs in each document
- Example: "database" ⟶ (D1: pos 3), (D3: pos 2)
- Why positions matter:
  - Phrase queries, proximity queries, …
  - E.g. "data management" vs "management data"

D1: "modern data management systems"
D2: "management of data in modern systems"

data → D1, D2
management → D1, D2

Query:
"data management"

Without positions, we can only answer: “Does the document contain the terms?”
With positions, we can answer: “Does it contain the exact phrase?”

Key Properties of Inverted Index

- Very fast lookup of terms
- Supports:
  - keyword queries
  - phrase queries
  - boolean queries
- Trade-offs:
  - extra storage overhead
  - updates are expensive
    - document is re-indexed (no in-place update)
- Where does inverted index fit?
  - Text analysis → produces terms
  - Indexing → builds inverted index
  - Query execution → uses index for fast lookup

Matching vs Ranking

- Inverted index answers: Which documents **match** the query?
- **Ranking** answers: Which of them is the most relevant?
  - Relevance depends on:
    - how often terms appear (**term frequency**)
    - how informative the terms are (**rare in the collection**)
    - how focused the document is (**length**)
- TF-IDF - rewards terms that are:
  - frequent in a document (TF = term frequency)
  - rare in the whole collection (IDF = inverse document frequency)
- BM25 - improves TF-IDF by:
  - limiting the effect of repeated terms (TF, but limited - TF saturation)
  - normalizing for document length

BM25 prefers short, focused documents over long or repetitive ones

Search Engines as First-Class Systems

- Full-text search requires:
  - text analysis, inverted indexes, ranking models
- Search engines therefore provide:
  - specialized storage structures
  - retrieval-oriented query processing
  - scalable execution over large collections
- They are not only auxiliary tools on top of databases
- In modern architectures, they often serve as
  - the primary retrieval layer
  - the analytics layer for logs and events
  - the backend for user-facing search



This is why Elasticsearch is often deployed next to, not instead of, a DBMS

Elasticsearch



- Distributed search and analytics engine
- Built on top of Apache Lucene
  - A low-level search library that provides core indexing and ranking capabilities
- Typical input sources:
  - logs from applications and devices
  - events and monitoring data
  - plain text, JSON, YAML, CSV, streams
- Internally stores data as JSON documents
  - Data is parsed, transformed, eventually enriched, …
- Provides a REST/HTTP API
- Core part of the Elastic Stack (formerly ELK Stack)
  - Logstash / Beats – ingestion
  - Elasticsearch – indexing, search, analytics
  - Kibana – visualization

https://www.elastic.co/elasticsearch

Why Elasticsearch?



- It implements the key ideas of information retrieval:
  - text analysis
  - inverted indexing
  - relevance ranking
- It extends them into a practical system with:
  - distributed storage and query execution
  - shards and replicas
  - near real-time search
  - aggregations over large collections
- Elasticsearch is a natural representative of modern search systems

Lucene gives us the search core; Elasticsearch turns it into a distributed, production-ready platform

Documents and Indices



- **Document** - basic unit of storage and retrieval
- **Field** - named attribute in a document
- **Index** - collection of related documents
  - Elasticsearch term roughly corresponding to a collection in MongoDB
    - Not database index!!!
  - In a cluster, there can be many indices
  - Documents in one index typically share similar characteristics
    - e.g., customers, products, logs

Mapping and Field Types

elasticsearch

- Mapping = schema of an index
  - Defines **field names and types**
  - Affects indexing, querying, sorting, and aggregations
- Common field types:
  - integer / float – numeric values
  - date – timestamps and time-based queries
  - boolean – true/false values
  - text – analyzed, used for full-text search
  - keyword – exact value, used for filtering, sorting, aggregations

```
{
  "city": "New York",
  "state": "CZ",
  "balance": 1200,
  "opened": "2024-01-10"
}
```

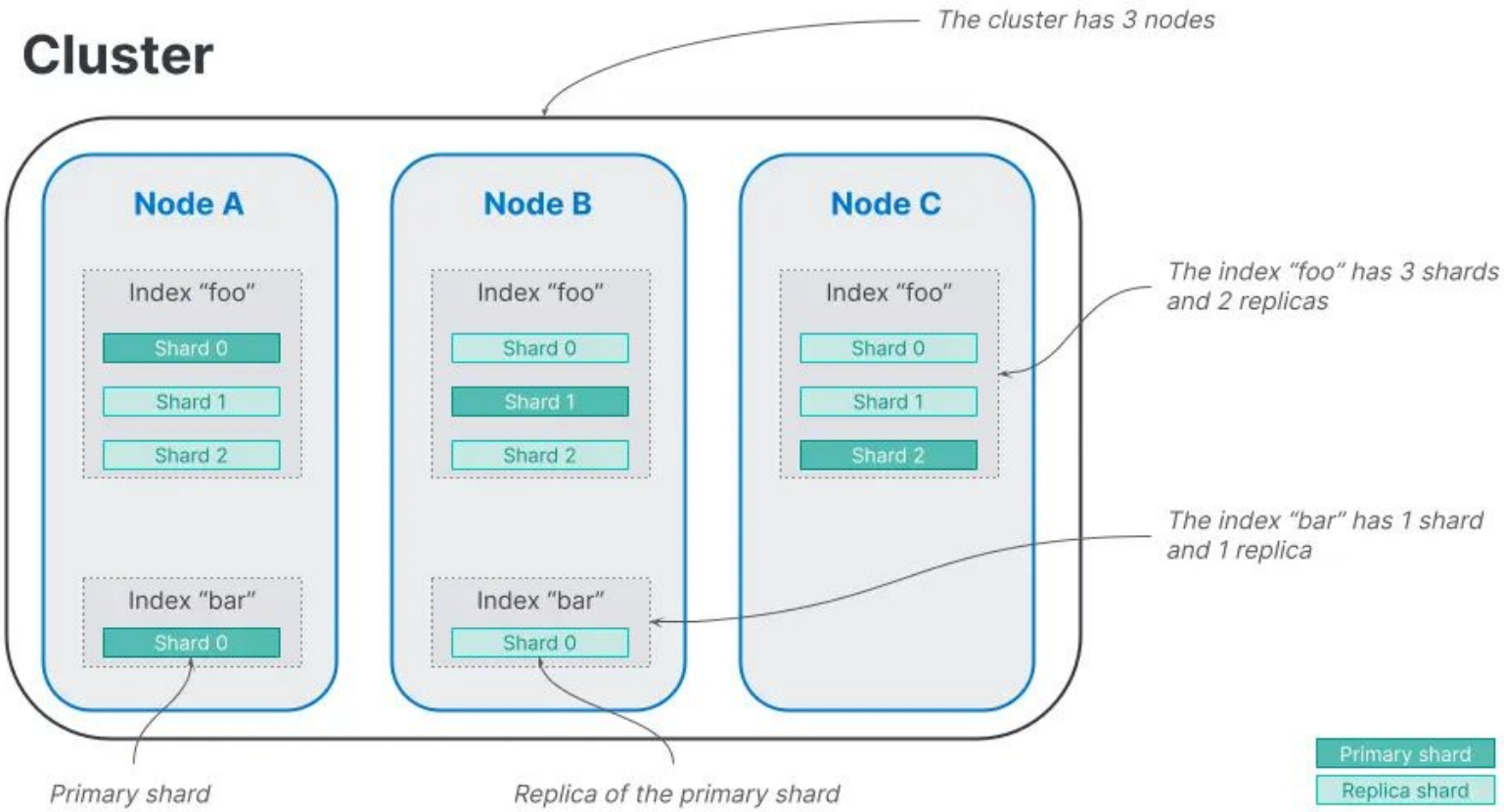
```
{
  "mappings": {
    "properties": {
      "city":    { "type": "text" },
      "state":   { "type": "keyword" },
      "balance": { "type": "integer" },
      "opened":  { "type": "date" }
    }
  }
```

The field type can be inferred automatically, but the correct choice between `text` and `keyword` should be made according to how the field will be queried

Cluster Architecture



- A cluster consists of one or more nodes
- Each index can be split into **shards**
  - an independent Lucene index
  - horizontal scaling, parallel query processing
- Each shard may have **replicas**
  - fault tolerance
  - parallel read operations
- Rebalancing and routing are handled automatically
- Each node can act as a **coordinator** to delegate operations to the respective shards
  - writes go to primary shards
  - reads can be served by primaries or replicas



https://www.elastic.co/search-labs/blog/thin-indexing-shards-elasticsearch-serverless

Near Real-Time Search

elasticsearch

- Indexed documents are not searchable instantly
- There is usually a **small delay**
  - approximately ~1 second
  - from the time you index (or update or delete) a document until the time it becomes searchable
- This is sufficient for:
  - search
  - analytics
  - monitoring
- Unlike transactional DBMSs, visibility for search is usually slightly delayed

Write first, search a moment later

Creating an Index and Basic CRUD

elasticsearch

- Basic operations
  - Create an index
  - Add a document
  - Replace and reindex the document
    - Note: If no ID is specified, Elasticsearch generates one automatically
  - Retrieve a document
  - Delete a document
  - Delete an index

```
PUT /customer

PUT /customer/_doc/1
{ "name": "John" }

PUT /customer/_doc/1
{ "name": "John Doe" }

GET /customer/_doc/1

DELETE /customer/_doc/1

DELETE /customer
```

Index Listing and Inspection

elasticsearch

- Elasticsearch provides administrative endpoints for quick inspection of indices
- Useful for:
  - listing existing indices
  - checking health and shard allocation
  - inspecting document counts and storage size
  - understanding index structure before querying or debugging

```
GET /_cat/indices?v

GET /_cat/indices/bank?v

GET /bank/_mapping

GET /bank/_count
```

```
health status index    pri rep docs.count store.size
green  open   bank       1   1       1000      450kb
green  open   customer   1   1          2        8kb
yellow open   logs       5   1    1245000      1.2gb
```

- health
  - green = all shards allocated, yellow = replicas missing, red = primary shards missing
- status
  - open = active and available for search/indexing
- index name
- number of primary shards
- number of replicas
- document count
- storage size

Document Updates

elasticsearch

- Elasticsearch does **not** update documents **in place**
  - old document is removed
  - new version is indexed
- This fits the index-oriented architecture
- Partial document updates are supported
  - modify fields
  - add fields
  - apply scripted changes

```
PUT /customer/_doc/1
{ "name": "John Doe" }

POST /customer/_doc/1/_update
{ "doc": { "name": "Jane Doe" } }

POST /customer/_doc/1/_update
{ "doc": { "name": "Jane Doe", "age": 20 } }

POST /customer/_doc/1/_update
{ "script": "ctx._source.age += 5" }
```

Internally an update is implemented as reindexing a new version of the document

Bulk Processing

elasticsearch

- Important for high-throughput ingestion
- Multiple actions can be sent in one request
- Typical use cases:
  - log ingestion
  - ETL pipelines
  - backfilling data

```
POST /_bulk
{"index":{"_id":"1"}}
{"name":"John Doe"}
{"index":{"_id":"2"}}
{"name":"Jane Doe"}

POST /_bulk
{"update":{"_id":"1"}}
{"doc":{"name":"John Doe becomes Jane Doe"}}
{"delete":{"_id":"2"}}
```

- 2x upsert document
- update 1 field + delete document

Search API - Sample Data

elasticsearch

- Sample bank document:


```
{
  "account_number": 0,
  "balance": 16623,
  "firstname": "Bradshaw",
  "lastname": "Mckenzie",
  "age": 29,
  "gender": "F",
  "address": "244 Columbus Place",
  "employer": "Euron",
  "email": "bradshawmckenzie@euron.com",
  "city": "Hobucken",
  "state": "CO"
}
```

Search API: URI vs Request Body

elasticsearch

- Search parameters can be sent:
  - in the request URI
  - in the request body
- Request body is usually:
  - more expressive
  - more readable
  - easier to extend
- Example:

```
GET /bank/_search?q=*&sort=account_number:asc&pretty
```

  - Search (`_search`) in the `bank` index,
  - Match all the documents (`q=*`),
  - Sort the results using the `account_number` field of each document in an ascending order (`sort=account_number:asc`)

```
{
  "took" : 9,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1000,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
```

```
      {
       "_index" : "holubova_bank",
       "_type" : "_doc",
       "_id" : "51",
       "_score" : 1.0,
       "_source" : {
         "account_number" : 51,
         "balance" : 14097,
         "firstname" : "Burton",
         "lastname" : "Meyers",
         "age" : 31,
         "gender" : "F",
         "address" : "334 River Street",
         "employer" : "Bezal",
         "email" : "burtonmeyers@bezal.com",
         "city" : "Jacksonburg",
         "state" : "MO"
       }
     }, …
```


The response contains execution metadata, shard-level information, and the actual matching documents with their relevance scores and stored JSON content

Search API: Result

elasticsearch

- `took` – time in milliseconds to execute the search
- `timed_out` – if the search timed out or not
  - The search exceeded the time limit (partial results may still be returned)
- `_shards` – how many shards were searched
  - Total, successful, failed, skipped
- `hits` – search results
  - `hits.total` – total number of documents matching our search criteria
  - `hits.hits` – actual array of search results
    - Default: first 10 documents
  - `hits.sort` – sort key for results
- …

- Regular search requests are typically stateless after the response is returned
  - Unlike cursor-based processing in many traditional relational systems

Search API: Request Body Form

elasticsearch

- The same exact search using the request body method

```
GET /bank/_search
{ "query": { "match_all": {} },
  "sort": [ { "account_number": "asc" } ] }
```

- Equivalent to URI-based search
- Better for complex queries

Query DSL

elasticsearch

- Domain-specific language for search queries
- JSON-style syntax
- Supports: matching, filtering, sorting, pagination, field selection, …

```
GET /bank/_search
{
  "query": { "match_all": {} },
  "from": 10,
  "size": 10,
  "_source": ["account_number", "balance"],
  "sort": { "balance": { "order": "desc" } }
}
```

- `query` – which documents match
- `from` – how many matching documents to skip
- `size` – how many documents to return
- `_source` – which fields are returned from each matching document
- `sort` – how matching documents are ordered

Main Query Categories

elasticsearch

- Full-text queries
  - match
  - match_phrase
- Term-level queries
  - term
  - range
  - exists
- Compound queries
  - bool
  - combine must / should / must_not / filter
- Aggregations
  - summaries and grouped statistics over result sets

Full-Text Query Examples

elasticsearch

```
"query": { "match": { "account_number": 20 } }
```

- Return the account numbered 20

```
"query": { "match": { "address": "mill" } }
```

- Return all accounts containing the term "mill" in the address

```
"query": { "match": { "address": "mill lane" } }
```

- Return all accounts containing the term "mill" or "lane" in the address

```
"query": { "match_phrase": { "address": "mill lane" } }
```

- Return all accounts containing the phrase "mill lane" in the address
  - Requires the terms in the given order and proximity

Term-Level Query Examples

elasticsearch

```
"query": { "term": { "state.keyword": "ID" } }
```

- Exact match on a keyword field

```
"query": { "range": { "balance": { "gte": 20000, "lte": 30000 } } }
```

- Numeric range query

```
"query": { "exists": { "field": "email" } }
```

- Return documents where a field is present

If dynamic mapping is used, string fields are often mapped as `text` fields with an additional `.keyword` subfield for exact matching, sorting, and aggregations

Scoring and `_score`

elasticsearch

- Search results include `_score`
  - Relative relevance measure = how well the document matches the search query
- Higher score = better match
  - Connects directly to TF-IDF / BM25 discussed earlier
- Intuition:
  - higher term frequency helps
  - more informative terms help

```
GET /bank/_search
{
  "query": {
    "match": {
      "address": "mill lane"
    }
  }
}
```

```
{
  "hits": {
    "hits": [
      {
        "_id": "1",
        "_score": 1.82,
        "_source": { "address": "Mill Lane" }
      },
      {
        "_id": "2",
        "_score": 1.14,
        "_source": { "address": "Old Mill Road" }
      },
      {
        "_id": "3",
        "_score": 0.76,
        "_source": { "address": "Lane Street" }
      }
    ]
  }
}
```

Exact scores depend on the ranking model and corpus statistics

Bool Query: `must` / `should`

elasticsearch

```
"query": {
  "bool": {
    "must": [
      { "match": { "address": "mill" } },
      { "match": { "address": "lane" } } ] } }
```

- Return all accounts that contain "mill" and "lane" in the address

```
"query": {
  "bool": {
    "should": [
      { "match": { "address": "mill" } },
      { "match": { "address": "lane" } } ] } }
```

- Return all accounts that contain "mill" or "lane" in the address

Bool Query: `must_not`

elasticsearch

```
"query": {
  "bool": {
    "must_not": [
      { "match": { "address": "mill" } },
      { "match": { "address": "lane" } } ] } }
```

- Excludes documents satisfying a condition

```
"query": {
  "bool": {
    "must": [
      { "match": { "age": "40" } } ],
    "must_not": [
      { "match": { "state": "ID" } } ] } }
```

- Combination

Filtering

elasticsearch

- Filters restrict the result set
  - Filters typically do not contribute to ranking
- Good for: numeric constraints, dates, ranges, exact conditions, …

```
"query": {
  "bool": {
    "must": { "match_all": {} },
    "filter": {
      "range": {
        "balance": {
          "gte": 20000,
          "lte": 30000
        } } } } }
```

Aggregations: Search + Analytics

elasticsearch

- Elasticsearch is not only for retrieving documents
- It can also compute summaries and statistics
  - Similar intuition to SQL `GROUP BY`
- We can execute searches returning both hits and aggregated results

```
GET /bank/_search
{
  "size": 0,    // do not show search hits
  "aggs": {
    "group_by_state": {
      "terms": {
        "field": "state.keyword"
      } } } }
```

Group all the accounts by `state`, and returns the top 10 (default) states sorted by count descending (default)

Aggregation with Sub-Aggregation

elasticsearch

```
GET /bank/_search
{
  "size": 0,
  "aggs": {
    "group_by_state": {
      "terms": { "field": "state.keyword" },
      "aggs": {
        "average_balance": {
          "avg": { "field": "balance" }
        } } } } }
```

First group by state, then compute average balance per state

The type of aggregation determines what is computed

- `terms` → count of documents per group (GROUP BY + COUNT)
- `avg` → average (AVG)
- `sum` → sum (SUM)
- `min` / `max` → minimum / maximum

Search vs Database Querying

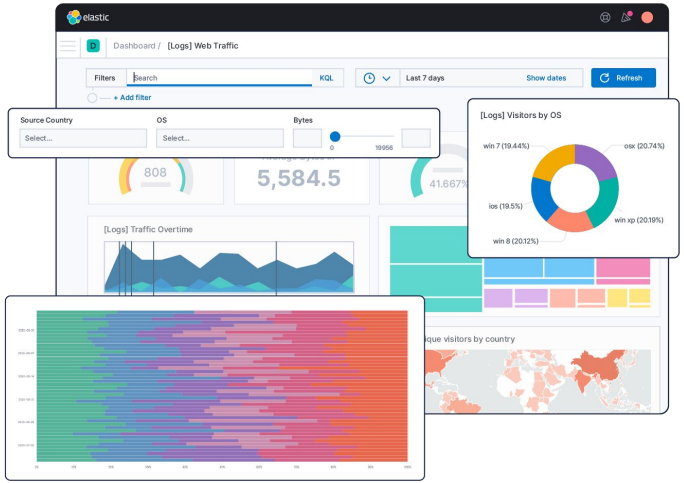
elasticsearch

- Elasticsearch is optimized for:
  - retrieval
  - ranking
  - filtering
  - aggregations over large collections
- Traditional DBMSs are optimized for:
  - transactions
  - joins
  - strong consistency
  - normalized updates

Elasticsearch in Big Data Pipelines

Logs / events / metrics
↓
Logstash / Beats
↓
Elasticsearch
↓
Kibana

- Elastic Stack (formerly ELK Stack)
  - **Logstash** / **Beats** – data collection and ingestion
  - **Elasticsearch** – indexing, search, analytics
  - **Kibana** – dashboards, visualization, exploration
- Common ingestion sources:
  - files
  - databases
  - logs
  - streams
- Good fit for:
  - operational analytics
  - event-centric workloads
  - monitoring pipelines
  - log exploration and observability



Elastic Stack Components

- Logstash
  - Parses, filters, enriches incoming data
  - Supports many input/output plugins
- Beats
  - Open source data shippers
  - Installed as agents on servers to send operational data
    - e.g., Filebeat (logs), Metricbeat (metrics)
  - Send data directly to Elasticsearch or Logstash
- Kibana
  - Visualization and exploration tool
  - Dashboards, charts, log exploration
  - Interactive querying and filtering

Elastic Stack turns raw data into searchable insights and visual dashboards

Apache Lucene



- Apache Lucene
  - Core search library used by Elasticsearch, but also Solr
- Provides:
  - inverted indexes
  - ranked searching
  - phrase / wildcard / proximity / range queries
  - sorting and grouping
- Elasticsearch builds a distributed system on top of it

Lucene = search core
Elasticsearch = distributed system built on top of it

https://lucene.apache.org/

References

- Elasticsearch https://www.elastic.co/elasticsearch
- Elasticsearch Query DSL https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html
- Elasticsearch Mapping https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html
- Kibana https://www.elastic.co/kibana
- Logstash https://www.elastic.co/logstash
- Apache Lucene https://lucene.apache.org/
- Apache Solr https://solr.apache.org/

Lecture 08

Multi-Model Data Management and System Integration

NDBI040 - Modern Database Systems

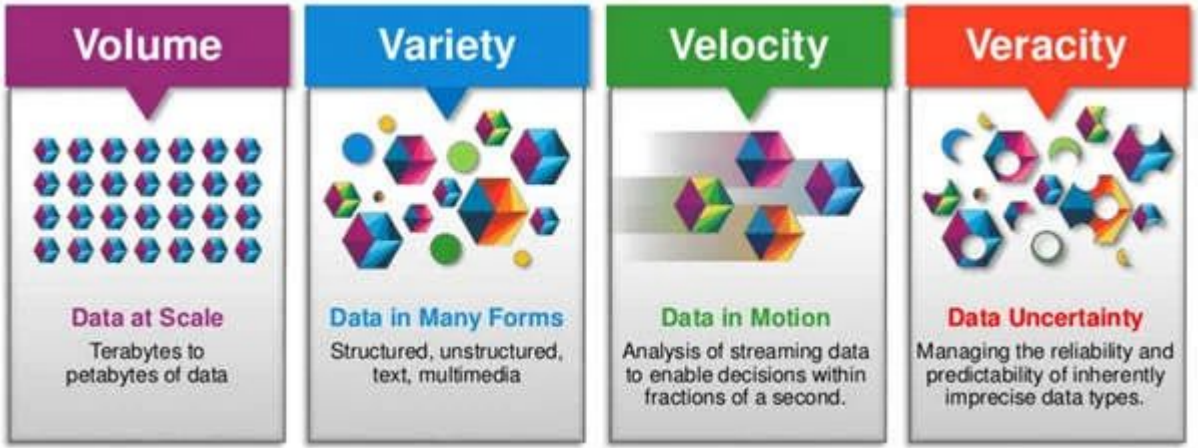
I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Big Data V-Characteristics

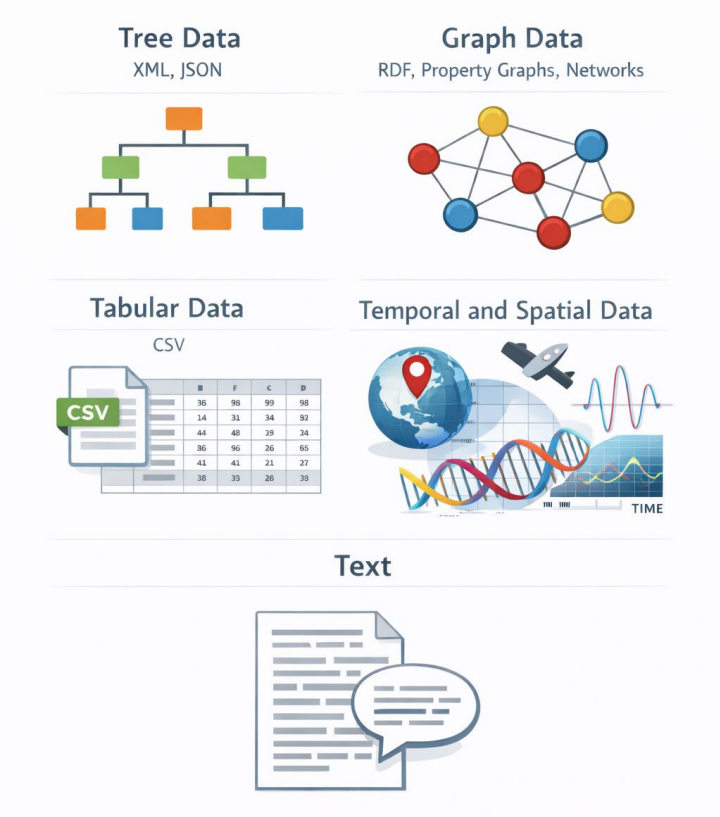
- Volume – scale
- Variety – complexity
- Velocity – speed
- Veracity – uncertainty
- Value
- Validity
- Volatility
- …



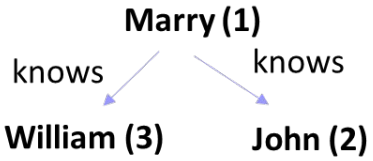
https://www.researchgate.net/figure/The-4-Vs-big-data-properties-volume-variety-velocity-veracity-9_fig1_338516812

Challenge: Data Variety

- Tree data
  - XML, JSON
- Graph data
  - RDF, property graphs, networks
- Tabular data
  - CSV
- Temporal and spatial data
- Text
- …



Example of Multi-Model Data



Social network graph

```
{ "Order_no":"0c6df508",
  "Orderlines": [
    { "Product_no":"2724f"
      "Product_Name":"Toy",
      "Price":66 },
    { "Product_no":"3424g",
      "Product_Name":"Book",
      "Price":40 } ]
}
```

Order JSON document

"1" --> "34e5e759"

"2"--> "0c6df508"

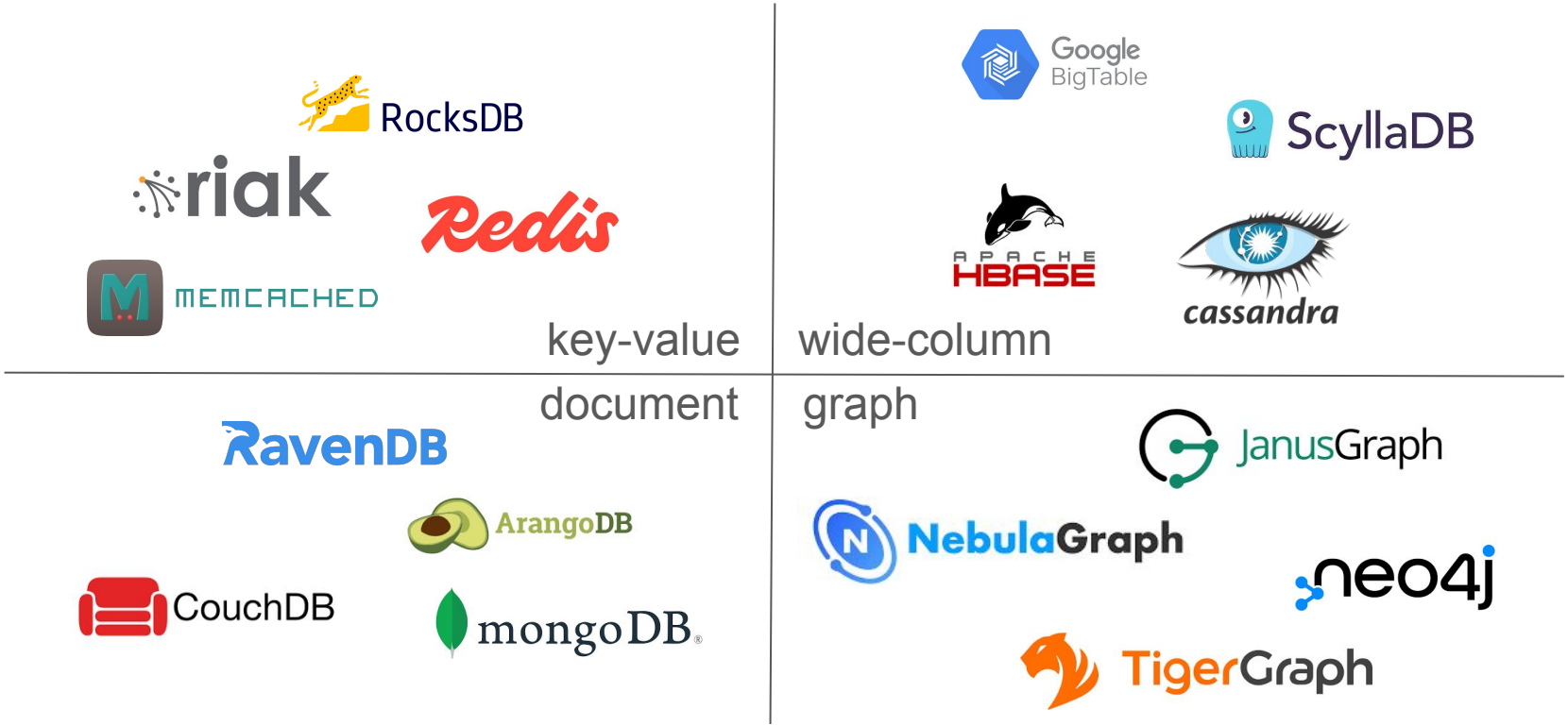
Key/value pairs
(Customer_ID , Order_no)

Customer relation

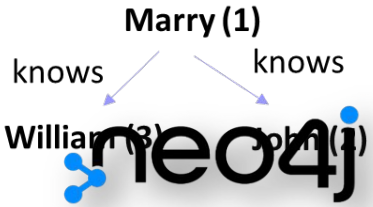
Customer_ID	Name	Credit_limits
1	Mary	5,000
2	John	3,000
3	William	2,000

Originally: different systems for different models

NoSQL DBMS



Example of Multi-Model Data



Social network graph

```
{ "Order_no":"0c6df508",
  "Orderlines": [
    { "Product_no":"2724f"
      "Product_Name":"Toy"
      "Price":66 },
    { "Product_no":"3424g",
      "Product_Name":"Book",
      "Price":40 } ]
}
```

MongoDB®

Order JSON document

"1" --> "34e5e759"

"2" --> "0c6df508"

riak

Key/value pairs
(Customer_ID , Order_no)

Customer relation

PostgreSQL

Customer_ID	Name	Credit_limits
1	Mary	5,000
2	John	3,000
3	William	2,000

Polyglot Persistence

- Idea: Use the right tool for the job
- Document store → for structured data with differences
- Graph store → for relations
- Key/value store → for simple managed structure

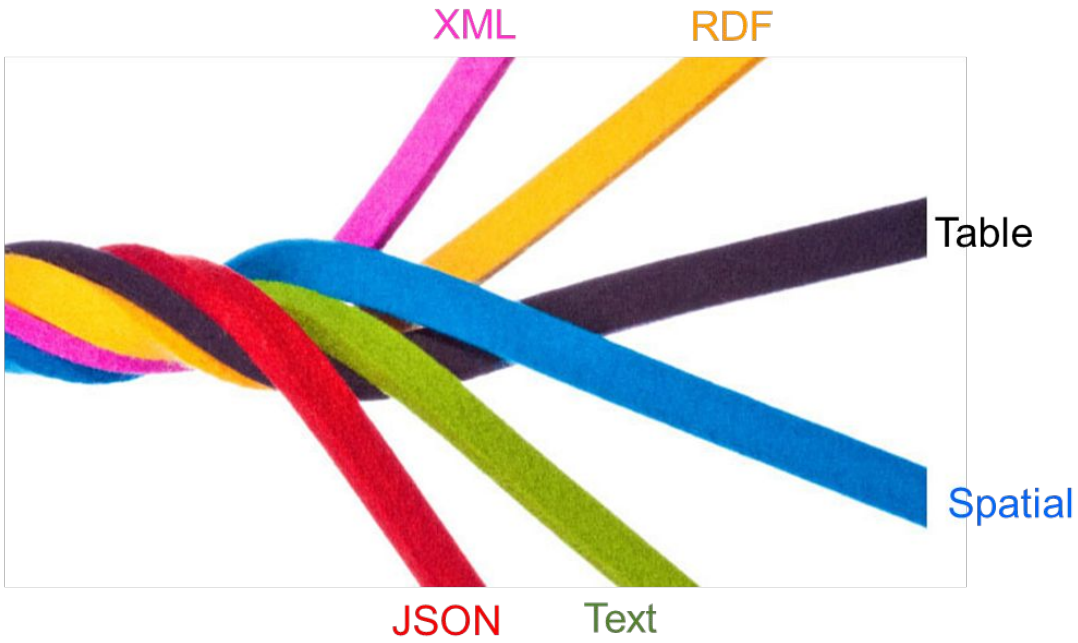
But integration becomes hard…

Pros and Cons of Polyglot Persistence

- Pros:
  - handles multi-model data
  - helps apps scale
  - rich experience, mature systems
    - standards
- Cons:
  - requires integration expertise
  - developers must learn multiple DBs
  - challenge of cross-model queries and transactions

Multi-Model Databases

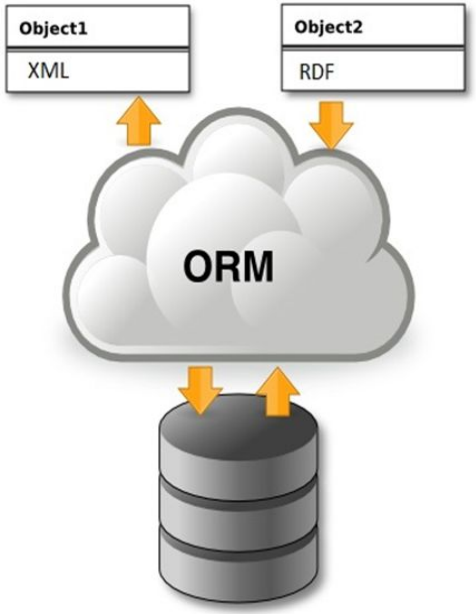




Designed to support multiple data models against a single, integrated backend

Multi-Model Databases Are Not New

- Its roots can be traced back to object-relational database systems (ORDBMS)
  - Extended the relational model with user-defined types, functions, indexes, …
- Moved beyond a single fixed data model



More DBMSs Become Multi-Model

Convergence trend: leading DBMSs gradually add support for multiple models

- relational systems ⟶ add JSON/XML
- document systems ⟶ add joins and graph-like operations
- graph systems ⟶ support nested attributes and documents

Rank			DBMS	Database Model	Score		
Apr 2026	Mar 2026	Apr 2025			Apr 2026	Mar 2026	Apr 2025
1.	1.	1.	Oracle	Relational, Multi-model	1157.93	-24.53	-73.12
2.	2.	2.	MySQL	Relational, Multi-model	857.69	-0.65	-129.41
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	702.08	-9.40	-82.93
4.	4.	4.	PostgreSQL	Relational, Multi-model			10
5.	5.	5.	MongoDB	Document, Multi-model			03
6.	6.	6.	Snowflake	Relational			28
7.	7.	↑ 12.	Databricks	Multi-model			19
8.	8.	↓ 7.	Redis	Key-value, Multi-model			58
9.	9.	9.	IBM Db2	Relational, Multi-model	113.25	+1.87	-12.79
10.	↑ 11.	↑ 11.	Apache Cassandra	Wide column, Multi-model	102.62	+0.74	-5.59
11.	↓ 10.	↓ 8.	Elasticsearch	Multi-model	99.51	-4.07	-28.57
12.	12.	↓ 10.	SQLite	Relational	95.90	-0.08	-18.19
13.	13.	↑ 14.	MariaDB	Relational, Multi-model	86.41	-0.59	-7.93

Relational DBMS
Document store
Graph DBMS
Spatial DBMS
Vector DBMS

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

Pros and Cons of Multi-model Databases

- Pros:
  - handle multi-model data
  - one fault-tolerant system
  - data consistency
  - unified query language
- Cons:
  - complex
  - immature
  - no standards

Example



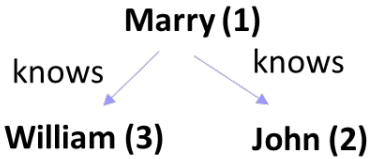
- open-source
- **document-first** multi-model
  - documents → graphs, key/value
- stores all data as documents
- vertices and edges are documents
- unified query language (AQL)



- open-source
- **graph-first** multi-model
  - graphs (objects) → documents
- records = vertices / edges / documents
- supports embedding and references
- SQL-like query language + graph traversals

Example of Multi-Model Data

Return all product numbers ordered by a friend of a customer whose credit limit is greater than 3000.



Social network graph

```
{ "Order_no":"0c6df508",
  "Orderlines": [
    { "Product_no":"2724f"
      "Product_Name":"Toy",
      "Price":66 },
    { "Product_no":"3424g",
      "Product_Name":"Book",
      "Price":40 } ]
}
```

Order JSON document

"1" --> "34e5e759"

"2"--> "0c6df508"

Key/value pairs
(Customer_ID , Order_no)

Customer relation

Customer_ID	Name	Credit_limits
1	Mary	5,000
2	John	3,000
3	William	2,000



Return all product numbers ordered by a friend of a customer whose credit limit is greater than 3000.

```
LET customerIDs = (
  FOR customer IN Customers
    FILTER customer.CreditLimit > 3000
    RETURN customer.id
)

LET friendIDs = (
  FOR customerID IN customerIDs
    FOR friend IN 1..1 OUTBOUND customerID Knows
      RETURN friend.id
)

FOR friendID IN friendIDs
  FOR order IN 1..1 OUTBOUND friendID Customer2Order
    RETURN order.Orderlines[*].Product_no
```



Return all product numbers ordered by a friend of a customer whose credit limit is greater than 3000.

```
SELECT expand( out("Knows").out("orders").orderlines.product_no )
FROM Customers
WHERE CreditLimit > 3000
```

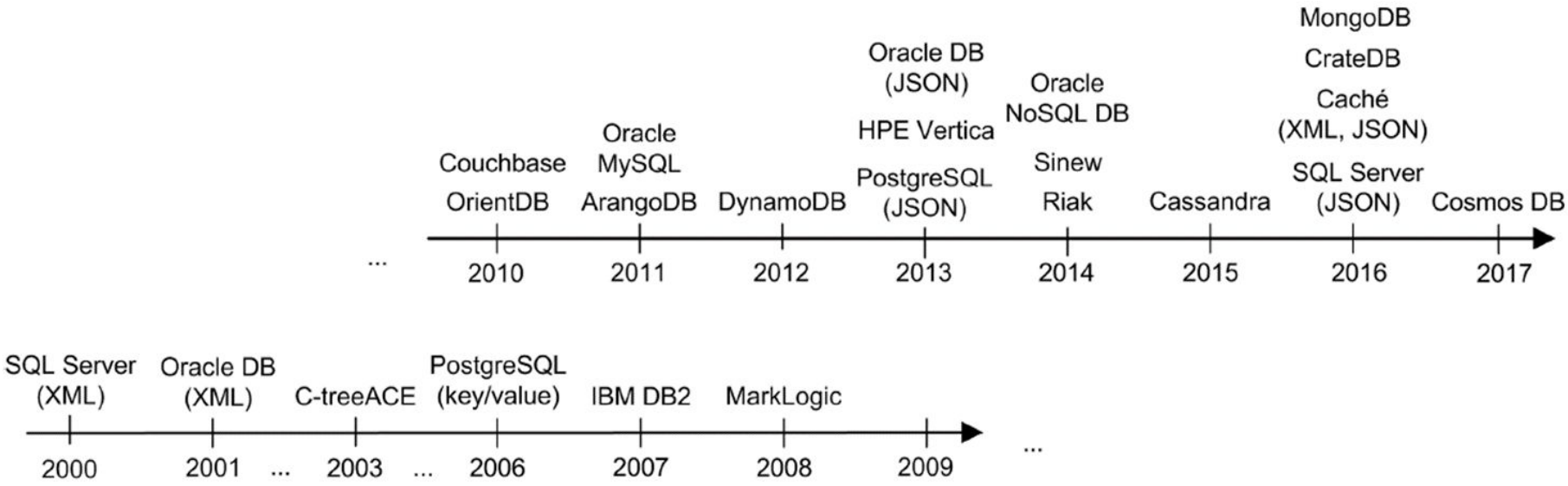
Classification of Multi-model Systems

According to the original model:

- Relational → PostgreSQL, SQL Server, IBM DB2, Oracle DB, MySQL
- Document → ArangoDB, Couchbase, MarkLogic, MongoDB, Cosmos DB
- Graph → OrientDB
- Key/value → Riak, Oracle NoSQL DB
- Column → Cassandra, CrateDB, DynamoDB, HPE Vertica

Different roots → different multi-model behavior

Early Evolution of Multi-Model Systems



How Systems Become Multi-Model

- New storage strategy for a new model
  - e.g., Relational systems adding native support for nested data
    - SQL/XML, SQL/JSOn standards
- Extend existing storage strategy
  - e.g., ArangoDB - nodes and edges stored as document collections
- New query interface
  - e.g., MarkLogic - stores JSON data in the same way as XML data
- No storage change
  - Formats simpler than the original one

Types of Cross-Model Transitions

1. Inter-model references

- links via IDs across models
- graph traversal over external data

✔ flexible, normalized
✖ requires joins / traversals

2. Model embedding

- relational data stored as nested JSON
- denormalization into document model

✔ fast reads, locality
✖ duplication, harder updates

3. Cross-model redundancy

- same data stored in multiple models

✔ performance, fewer joins
✖ consistency, update anomalies

Reference = integration via links
Embedding = integration via structure
Redundancy = integration via duplication

Practical Integration Challenges I

- Cross-model query processing
  - joins across models (graph ↔ document ↔ relational)
  - different execution strategies
  - no unified optimizer
- Data consistency
  - redundancy → update anomalies
  - embedding → duplicated state
  - references → distributed updates
- Schema & model evolution
  - JSON → flexible schema
  - relational → strict schema
  - graph → evolving structure

Practical Integration Challenges II

- Indexing & optimization
  - B-tree (relational)
  - inverted / JSON index (document)
  - traversal optimization (graph)
- Transactions across models
  - ACID vs eventual consistency
  - cross-model transactions expensive
- Developer complexity
  - multiple query languages
  - different APIs
  - mental model mismatch

Integration is not only about data — it is about combining incompatible assumptions



```
CREATE TABLE customer (
  id       INTEGER PRIMARY KEY,
  name     VARCHAR(50),
  address  VARCHAR(50),
  orders   JSONB
);
```

```
INSERT INTO customer
VALUES (1, 'Mary', 'Prague',
 '{"Order_no":"0c6df508",
   "Orderlines":[
     {"Product_no":"2724f", "Produ
     {"Product_no":"3424g", "Produ
  }');
INSERT INTO customer
VALUES (2, 'John', 'Helsinki',
 '{"Order_no":"0c6df511",
   "Orderlines":[
     { "Product_no":"2454f", "Product_Name":"Computer", "Price":34 }]
  }');
```

```
SELECT name,
  orders->>'Order_no' as Order_no,
  orders#>'{Orderlines,1}'->>'Product_Name as Prod_name
FROM customer
where orders->>'Order_no' <> '0c6df511';
```

id integer	name character varying (50)	address character varying (50)	orders jsonb
1	Mary	Prague	{"Orderlines":[{"Price":66,"Product_Name":"Toy","Product_no":"2724f"},{"Price":40,"Product_Name":...
2	John	Helsinki	{"Orderlines":[{"Price":34,"Product_Name":"Computer","Product_no":"2454f"}],"Order_no":"0c6df511"}




```
SELECT orders[0]
FROM myspace.customer
WHERE id = 1;
```

```
create keyspace myspace
WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
CREATE TYPE myspace.orderline (
   product_no text,
   product_name text,
   price float
   );
CREATE TYPE myspace.myorder (
   order_no text,
   orderlines list<frozen <orderline>>
   );
CREATE TABLE myspace.customer (
   id INT PRIMARY KEY,
   name text,
   address text,
   orders list<frozen <myorder>>
   );
```

```
INSERT INTO myspace.customer JSON
' {"id":1,
   "name":"Mary",
   "address":"Prague",
   "orders" : [
   { "order_no":"0c6df508",
     "orderlines":[
     { "product_no" : "2724f",
       "product_name" : "Toy",
       "price" : 66 },
       { "product_no" : "3424g",
         "product_name" :"Book",
         "price" : 40 } ] } ]
  }';
```



XQuery:

```
let $product := fn:doc("/myXML1.xml")/product
let $order := fn:doc("/myJSON1.json")[Orderlines/Product_no = $product/@no]
return $order/Order_no

Result: 0c6df508
```

JavaSript:

```
declareUpdate();
 xdmp.documentInsert("/myJSON1.json",
 {
   "Order_no":"0c6df508",
    "Orderlines":[
     { "Product_no":"2724f",
       "Product_Name":"Toy",
       "Price":66 },
     {"Product_no":"3424g",
      "Product_Name":"Book",
      "Price":40}]
 }
 );
```

XQuery:

```
xdmp:document-insert("/myXML1.xml",
 <product no="3424g">
   <name>The King's Speech</name>
   <author>Mark Logue</author>
   <author>Peter Conradi</author>
 </product>
 );
```



```
CREATE CLASS orderline EXTENDS V
 CREATE PROPERTY orderline.product_no STRING
 CREATE PROPERTY orderline.product_name STRING
 CREATE PROPERTY orderline.price FLOAT

CREATE CLASS order EXTENDS V
 CREATE PROPERTY order.order_no STRING
 CREATE PROPERTY order.orderlines EMBEDDEDLIST orderline

CREATE CLASS customer EXTENDS V
 CREATE PROPERTY customer.id INTEGER
 CREATE PROPERTY customer.name STRING
 CREATE PROPERTY customer.address STRING

CREATE CLASS orders EXTENDS E

CREATE CLASS knows EXTENDS E
```



```
CREATE VERTEX order CONTENT {
   "order_no":"0c6df508",
    "orderlines":[
     { "@type":"d",
       "@class":"orderline",
       "product_no":"2724f",
       "product_name":"Toy",
       "price":66 },
     { "@type":"d",
       "@class":"orderline",
       "product_no":"3424g",
       "product_name":"Book",
       "price":40}]
 }
```

```
CREATE VERTEX order CONTENT {
   "order_no":"0c6df511",
    "orderlines":[
     { "@type":"d",
     "@class":"orderline",
     "product_no":"2454f",
     "product_name":"Computer",
     "price":34 }]
 }
CREATE VERTEX customer CONTENT {
   "id" : 1,
   "name" : "Mary",
   "address" : "Prague"
   }
CREATE VERTEX customer CONTENT {
   "id" : 2,
   "name" : "John",
   "address" : "Helsinki"
   }
```

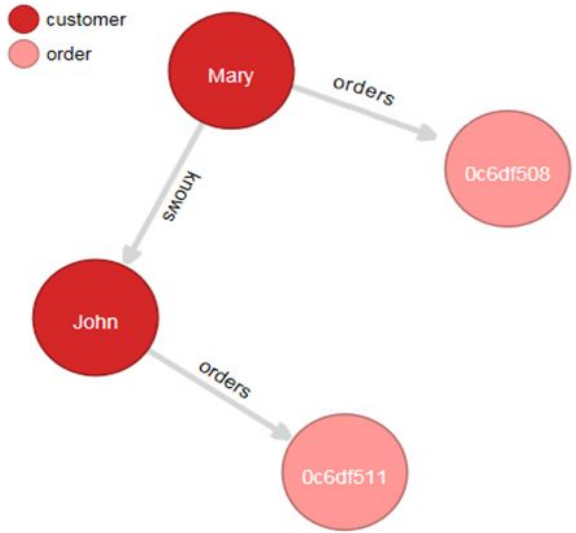
OrientDB

```
SELECT out("knows").out("orders").orderlines.product_no
FROM customer
WHERE name="Mary";
```

```
CREATE EDGE orders FROM
   (SELECT FROM customer WHERE name = "Mary")
   TO
   (SELECT FROM order WHERE order_no = "0c6df508")

CREATE EDGE orders FROM
   (SELECT FROM customer WHERE name = "John")
   TO
   (SELECT FROM order WHERE order_no = "0c6df511")

 CREATE EDGE knows FROM
   (SELECT FROM customer WHERE name = "Mary")
   TO
   (SELECT FROM customer WHERE name = "John")
```



Multi-Model Database Challenges

Modeling & Representation

- Open data model
- Schema & model evolution
- References vs embedding vs redundancy

Storage & Indexing

- Multi-model index structures
- Different access patterns
  - Graph / JSON / table
- No universal index



Transactions & Consistency

- Multi-model transactions
- ACID vs eventual consistency
- Consistency across representations

Query & Processing

- Unified query language
- Cross-model query processing
- No universal optimizer

ArangoDB



- Open-source multi-model DBMS
  - Document-first system
- Supports:
  - document model
  - graph model
  - key/value access
- Stores data in collections
  - Graph data represented by vertex and edge collections
- One query language for multiple models: AQL

How ArangoDB Stores Data



- ArangoDB is document-first
  - Everything is stored in collections
  - A document is a JSON-like object
- Each document has system attributes:
  - `_key` = local identifier (within a collection)
  - `_id` = collection/`_key`
  - `_rev` = revision (changes on update)
- Graphs are built on top of documents:
  - vertex collections store vertices
  - edge collections store edges
- Edge documents contain:
  - `_from`
  - `_to`

AQL: Arango Query Language



- Declarative query language
  - Designed for JSON-like data
  - Works over documents and graphs
- Supports:
  - filtering (`FILTER`)
  - projection / return (`RETURN`)
  - aggregation (`COLLECT`, `AGGREGATE`)
  - sorting (`SORT`), limiting (`LIMIT`)
  - traversals
  - shortest path queries
  - …

```
FOR x IN collection
FILTER ...
SORT ...
LIMIT ...
RETURN ...
```

AQL for Document Querying

collection: people_doc

```
{
  "_key": "u10",
  "id": 10,
  "given_name": "Alice",
  "family_name": "Brown",
  "email": "alice@example.com",
  "likes": [
    {"target": "databases", "weight": 5},
    {"target": "graphs", "weight": 3},
    {"target": "aql", "weight": 2}
  ]
}
```

```
{
  "_key": "u11",
  "id": 11,
  "given_name": "Bob",
  "family_name": "Miller",
  "email": "bob@example.com",
  "likes": [
    {"target": "graphs", "weight": 4},
    {"target": "ai", "weight": 2}
  ]
}
```

AQL for Document Querying




```
FOR p IN people_doc
  FILTER p.id == 10
  RETURN {
    person: {
      id: p.id,
      email: p.email,
      given_name: p.given_name,
      family_name: p.family_name
    },
    likes: p.likes
  }
```

```
LET me = FIRST(
  FOR p IN people_doc
    FILTER p.id == 10
    RETURN p
)
```

```
FOR l IN me.likes
  COLLECT target = l.target
  AGGREGATE score = SUM(l.weight)
  SORT score DESC
  LIMIT 10
  RETURN { target, score }
```

AQL for Graph Traversal

vertex collection: `people`

```
{
  "_key": "u10",
  "name": "Alice"
}
```

```
{
  "_key": "u11",
  "name": "Bob"
}
```

```
{
  "_key": "u12",
  "name": "Carol"
}
```

edge collection: `follows`

```
{
  "_key": "f1",
  "_from": "people/u10",
  "_to": "people/u11",
  "since": 2023,
  "strength": 0.9
}
```

```
{
  "_key": "f2",
  "_from": "people/u11",
  "_to": "people/u12",
  "since": 2024,
  "strength": 0.7
}
```

AQL for Graph Traversal




```
FOR v, e, p IN 1..2 OUTBOUND startVertex edgeCollection
RETURN ...
```

Traverse from `startVertex` along `edgeCollection`, following outgoing edges up to the given depth

- **`v` = current vertex**
- **`e` = the last edge**
- **`p` = the whole path**

```
FOR v, e, p IN 1..2 OUTBOUND DOCUMENT("people/u10") follows
  RETURN {
    vertex: v.name,
    via: e._key,
    path: p.vertices[*].name
  }
```

```
[
  { "vertex": "Bob",   "via": "f1", "path": ["Alice", "Bob"] },
  { "vertex": "Carol", "via": "f2", "path": ["Alice", "Bob", "Carol"] }
]
```

```
FOR v, e IN OUTBOUND SHORTEST_PATH DOCUMENT("people/u10") TO DOCUMENT("people/u12")
  follows
  RETURN v.name
```

```
["Alice", "Bob", "Carol"]
```

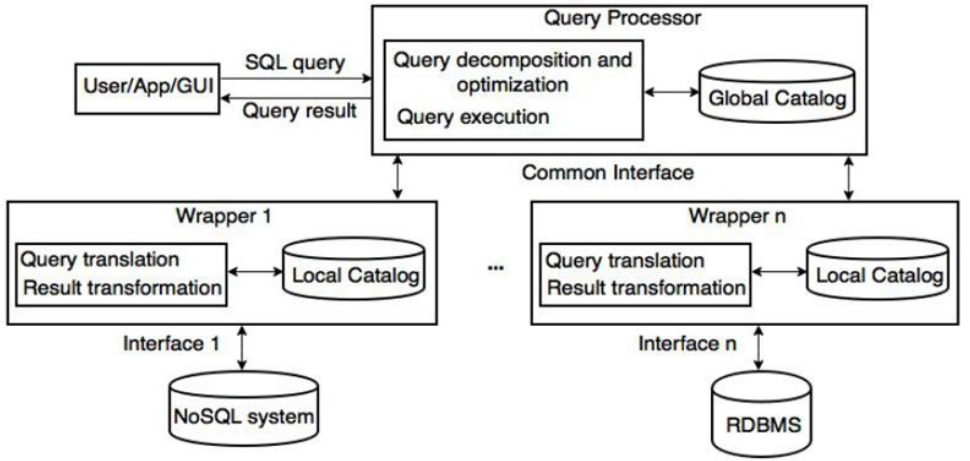
Two Integration Strategies for Multi-Model Data

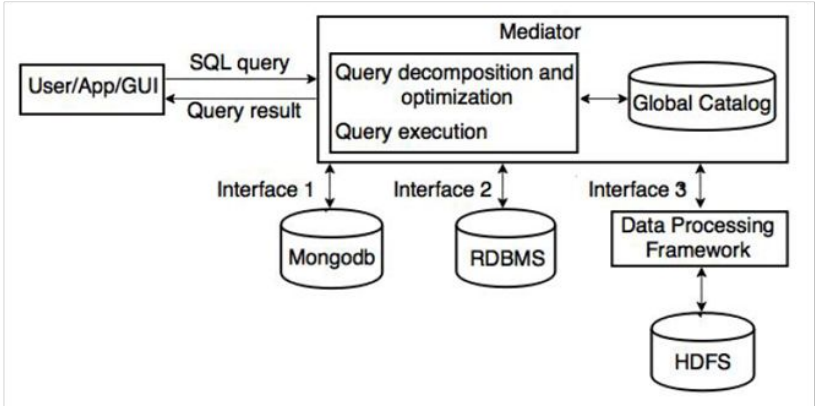
- **Multi-model databases**
  - One single, integrated backend
- **Polystores**
  - Multiple data storage technologies used jointly
  - Idea: Use the right tool for (each part of) the job
    - hierarchical data ⟶ document store
    - relationships ⟶ graph database
    - simple scalable acces ⟶ key-value store
    - full-text search ⟶ search engine
    - …

  and glue everything together

Types of Polystores

- **Loosely-coupled**
  - mediator-wrapper style
  - common interface
  - autonomy of local stores
- **Tightly-coupled**
  - direct use of local interfaces
  - better performance
  - lower autonomy
- **Hybrid**
  - combines both





Dimensions of Polystores

- Heterogeneity
  - different data models, different query languages, different engines
- Autonomy
  - local stores keep own interfaces and evolution
  - federation must respect local behavior
- Transparency
  - hide data location and movement
  - but still explain decisions when needed
- Flexibility
  - modular architecture
  - extensible interfaces and mappings
- Optimality
  - federated plans
  - data placement and migration matter

Mostly academic / research systems

System	Origin / Creators	Combines
BigDAWG	MIT + Intel + Brown University	PostgreSQL + SciDB + Accumulo
CloudMdsQL	INRIA / LIRMM (France)	relational + NoSQL stores
Myria	University of Washington	distributed analytics backends
Musketeer	University of Cambridge	multiple processing engines
QoX / Forward / SQL++	academic prototypes	heterogeneous stores

Related enterprise platforms

System	Origin	Focus
Denodo	Denodo Technologies	data virtualization
Teiid	Red Hat / JBoss	federation
Trino / Presto	Facebook / open source	SQL over many sources

Why Polystores Are Hard?

- Each engine has
  - own model
  - own query language
  - own optimizer
  - own consistency model
- Integration must bridge all layers
- Query processing:
  - parse query
  - split into subqueries
  - run in local engines
  - move partial data
  - join / merge results

Legacy of Polystores

- Polystores as a standalone category promised to:
  - connect multiple heterogeneous engines
  - send subqueries to the most suitable system
  - move data between systems when needed
  - hide heterogeneity behind a single integration layer
- This did not become mainstream, but ideas reappeared in cloud data platforms
  - unified access over multiple engines and sources
  - support for heterogeneous workloads under one platform
  - data movement via caching, replication, ETL / ELT
  - …

Polystore ideas survived more as platform architecture than as standalone DBMSs

References

- Neither Fish Nor Fowl: the Rise of Multi-model Databases. The 451 Group, 2013.
- J. Lu, Z. H. Liu, P. Xu, and C. Zhang. UDBMS: road to unification for multi-model data management. CoRR, abs/1612.08050, 2016
- J. Lu: Towards Benchmarking Multi-model Databases. CIDR 2017
- Kolev, B. et al.: Benchmarking Polystores: the CloudMdsQL Experience
- Kharlamov, E. et al.: A Semantic Approach to Polystores
- ArangoDB https://arango.ai/

Lecture 09

Specialized and Emerging Database Systems

NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Why Specialized Database Systems Matter?

- Traditional relational DBMS excel at:
  - transactions
  - structured tabular data
  - strong consistency
  - business workloads
- But modern workloads often need:
  - similarity search over embeddings
  - multidimensional scientific arrays
  - geospatial indexing
  - globally distributed SQL
  - uncertain / incomplete data reasoning
  - …

→ rise of specialized database systems

Not all data problems are best solved by classical relational DBMS

One Size Does Not Fit All

Need	Example Systems
Numerical tensors / matrices	SciDB, [tile]DB
Global ACID transactions	CockroachDB, TiDB
Similarity search	Pinecone, Qdrant
Maps / geometry / routing	PostGIS, Oracle Spatial

Different workloads optimize different primitives:

- joins
- nearest neighbors
- raster operations
- replication latency

How to Choose a DBMS?

Ask these questions:

- What is the dominant data model?
  - tables? vectors? graphs? arrays? polygons?
- What matters most?
  - latency? throughput? consistency? scalability? analytical power?
- What operations dominate?
  - joins? aggregations? nearest neighbor search? spatial joins? matrix algebra?
- Can one system do all?
  - **Usually no**

Reality: Polyglot persistence is common

NoSQL as a First Step Beyond Relational Databases

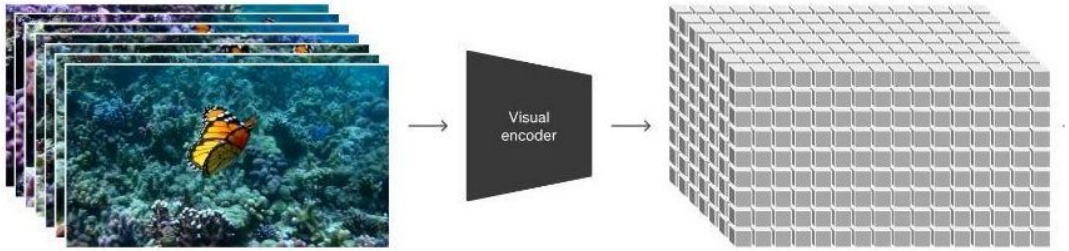
- NoSQL brought:
  - alternative data models
  - horizontal scalability
  - schema flexibility
  - workload-specific optimization
- Mainly addressed: web applications, large-scale operational data, semi-structured or highly connected data, …
- It did not fully address
  - multidimensional scientific data → array DBMS
  - global ACID SQL at scale → NewSQL
  - high-dimensional similarity search → vector DBMS
  - geometry, topology, map operations → spatial DBMS
  - uncertain / probabilistic tuples and queries → probabilistic DBMS

NoSQL broadened the database world, but specialized systems pushed specialization even further

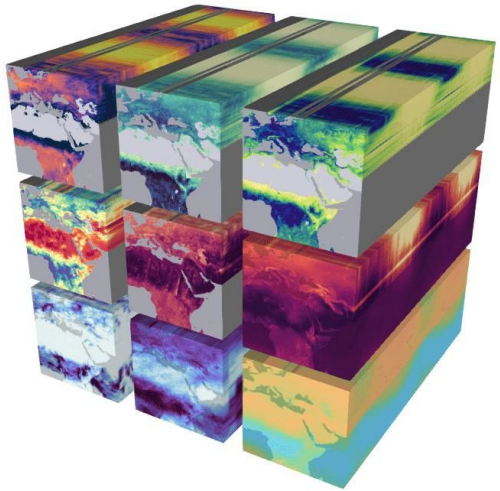
Array Databases

Motivation

Video Analytics



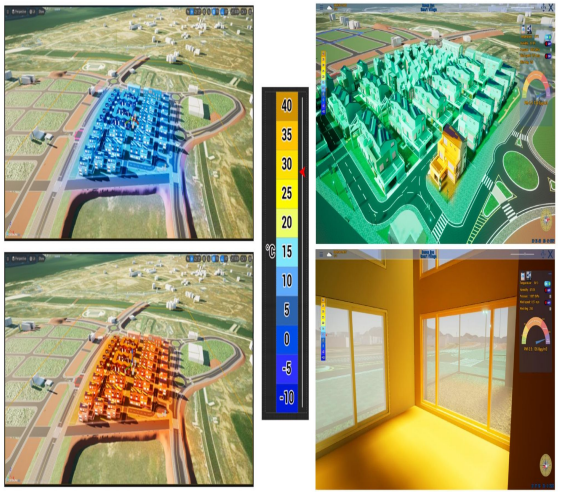
Many modern datasets are naturally tensors, grids, cubes, or volumes rather than tables



Climate / Earth Data Cube



CT Volume Reconstruction

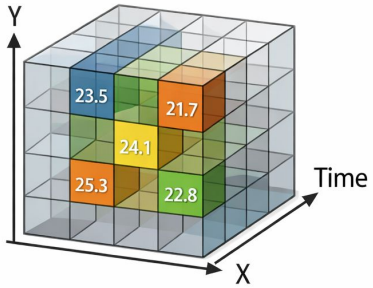


City Map + Time Layers

Why Array Databases?

X	Y	Time	Value
1	2	12:00	23.5
1	3	12:00	21.7
2	1	12:00	24.1
2	2	13:00	25.3
3	1	13:00	22.8

vs.



- Some data are naturally multidimensional, not relational
  - Biology, chemistry, physics, geology, …
  - e.g. astronomical measurements, changes of climate, satellite pictures of the Earth, oceanographic data, human genome, …
- Natural dimensions
  - space: x, y, z
  - time
  - wavelength / frequency
  - experiment / sample / scenario
- Problem with plain relational tables
  - possible, but unnatural
  - too many tuples for dense grids
  - neighborhood and window operations become expensive
  - poor fit for matrix-style analytics

rasdaman
raster data manager

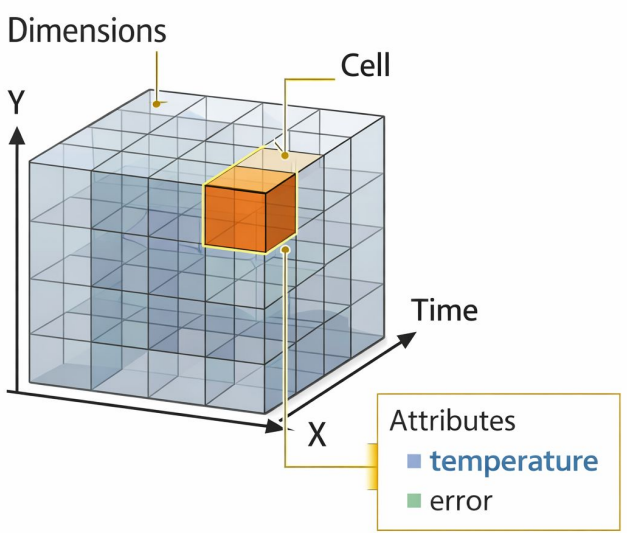
[tile]DB

SciDB

Arrays are a native model for data with coordinates and locality

What Is an Array Database?

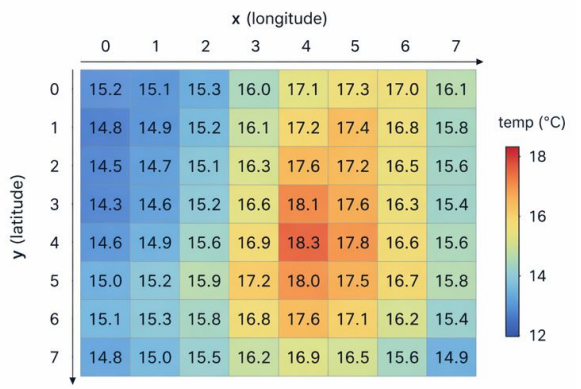
- An array database stores data as multidimensional arrays rather than only as flat tables
- Core idea: An array consists of:
  - dimensions = coordinate axes
  - cells = positions in the array
  - attributes = values stored in each cell
- Why this matters
  - locality-aware storage
  - chunk-based partitioning
  - efficient slicing and window queries
  - natural support for matrix and raster operations

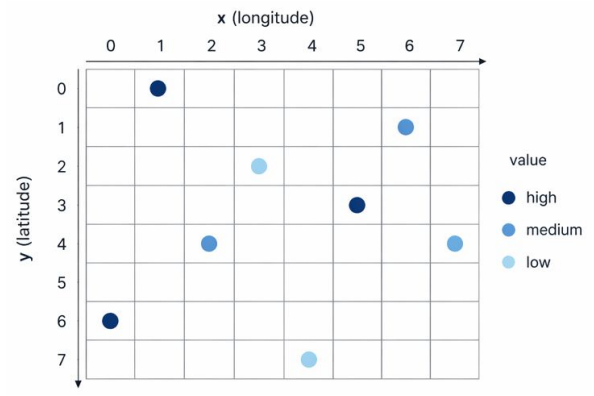


Dense vs Sparse Arrays

Type	Example	Characteristic
Dense	image pixels, matrices	most cells contain values
Sparse	graphs, sensor gaps	many cells empty

- Why important
  - storage layout differs
  - compression matters
  - operators may exploit sparsity

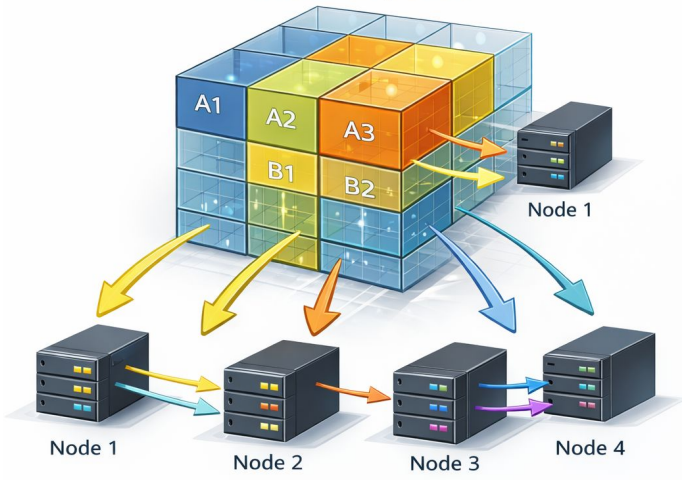




Modern array DBMS often support both dense and sparse arrays

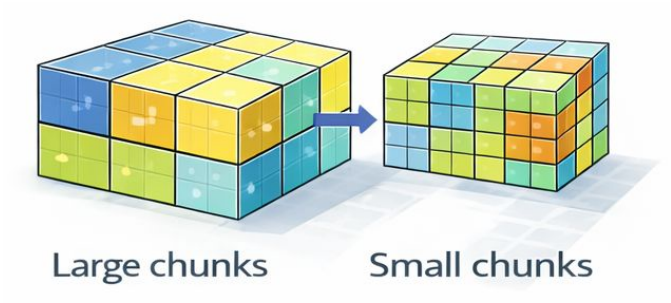
Chunk-Based Storage and Distribution

- Large arrays are partitioned into chunks = basic unit of storage, transfer, and parallel processing
- Why chunk arrays?
  - scalable storage of massive multidimensional data
  - efficient sequential I/O on contiguous blocks
  - parallel execution across nodes
  - locality-preserving access to nearby cells
  - reduced metadata overhead compared to cell-level storage



Chunks are the key physical abstraction behind scalable array processing

1. Chunk Size



- Trade-off: I/O efficiency vs parallelism
- Larger chunks
  - fewer metadata entries
  - better sequential reads
  - lower scheduling overhead
  - efficient full scans
- Smaller chunks
  - more parallel tasks
  - better selective access
  - finer-grained load balancing
  - lower wasted I/O for small queries
- Typical tension:
  - Too large → underutilized cluster
  - Too small → overhead dominates

2. Chunk Shape

- Trade-off: workload locality
- Examples:
  - 1024 x 16 → row-oriented access
  - 16 x 1024 → column/stripe access
  - 128 x 128 → balanced 2D workloads
  - 64 x 64 x 16 → 3D cubes
- Choose shape according to query patterns:
  - row scans
  - slicing one dimension
  - neighborhood/window queries

3. Placement Policy

- Where chunks are stored
- Hash-based
  - good balance
  - random placement
  - weak locality
- Range-based
  - preserves order / locality
  - useful for range scans
  - risk of skew / hotspots
- Cost-aware/adaptive
  - uses workload statistics
  - colocates frequently joined/accessed chunks
  - harder to maintain

4. Replication

- Reliability and performance
- Benefits
  - fault tolerance
  - higher read throughput
  - faster local reads
  - maintenance flexibility
- Costs
  - extra storage
  - update coordination
  - consistency overhead

Chunk Overlap for Neighborhood Queries

- Many array operations need neighboring cells → systems may store overlapping border regions
- Benefits
  - lower network traffic
  - faster window queries
  - better parallel scalability
  - simpler operator execution
- Costs
  - duplicated storage
  - update complexity
  - refreshing halo regions

Overlap trades extra storage for faster neighborhood analytics

Defining an Array Schema

SciDB

```
CREATE ARRAY temperature
<temp: double, err: double>
[x=0:999,100,1, y=0:999,100,1, t=0:364,30,0];

STORE(
  BUILD(
    temperature,
    iif(x = 10 AND y = 20 AND t = 1, 10.5, null),
    iif(x = 10 AND y = 20 AND t = 1, 0.1, null)
  ),
  temperature
);
```

- temp, err = attributes
- x, y, t = dimensions
  - x = dimension name
  - 0:999 = coordinate range
  - 100 = chunk length
  - 1 = overlap

Basic Array Queries

SciDB

```
-- Slice / projection
SELECT temp
FROM temperature;

-- Coordinate slice
SELECT temp
FROM temperature
WHERE t = 100;

-- Subregion filter
SELECT temp
FROM temperature
WHERE t = 100 AND
      x > 100 AND x < 200 AND
      y > 200 AND y < 400;
```

```
-- Derived value
SELECT sqrt(temp) AS temp_sqrt
FROM temperature
WHERE t = 100;

-- Aggregate over one dimension
SELECT avg(temp)
FROM temperature
WHERE t = 100;
```

More Complex Operations

SciDB

- Joins and array combination
  - merge or align multiple arrays
  - combine values from compatible coordinate spaces
- Nested queries
- Aggregation and sorting
- Block-wise aggregation
  - partition arrays into blocks and compute per-block statistics
  - e.g., sum, average, centroid
- Linear algebra and transformations
  - matrix–matrix and matrix–vector multiplication
  - factorizations and image-processing transformations
- Array-native operations
  - subarray / between, filter, regrid, join aligned arrays, window / neighborhood computation

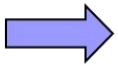
Array: **input < a:int32 > [I=0:6,J=0:8]**

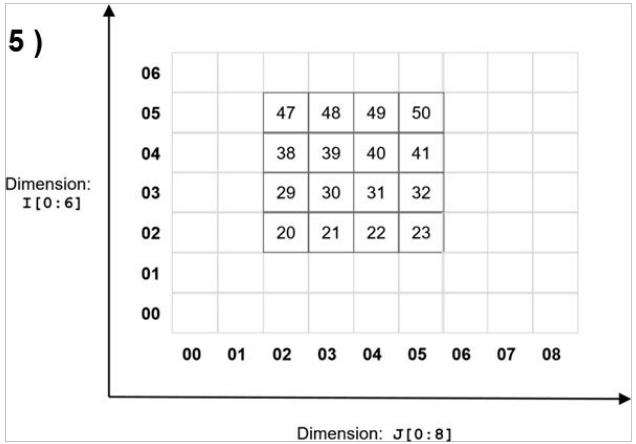
	00	01	02	03	04	05	06	07	08
06	54	55	56	57	58	59	60	61	62
05	45	46	47	48	49	50	51	52	53
04	36	37	38	39	40	41	42	43	44
03	27	28	29	30	31	32	33	34	35
02	18	19	20	21	22	23	24	25	26
01	9	10	11	12	13	14	15	16	17
00	0	1	2	3	4	5	6	7	8

Dimension: **I[0:6]**

Dimension: **J[0:8]**

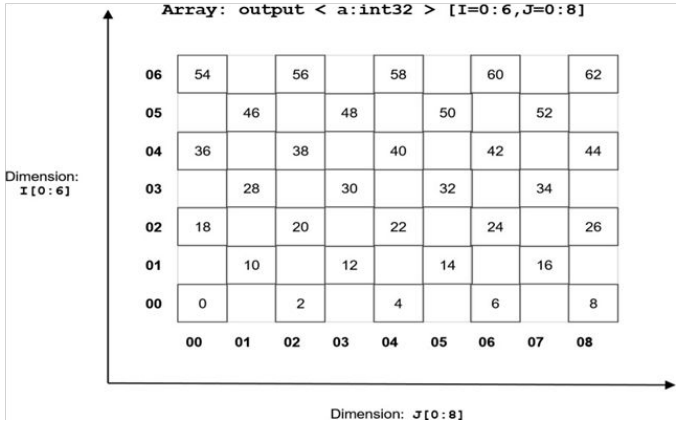
between (input, 2, 2, 5, 5)





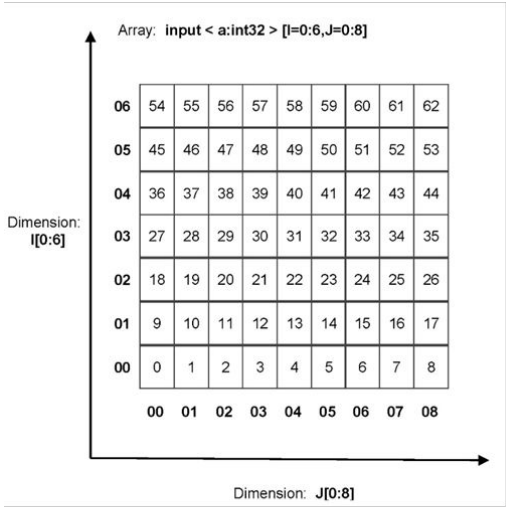
filter (input, a%2 = 0)

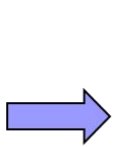


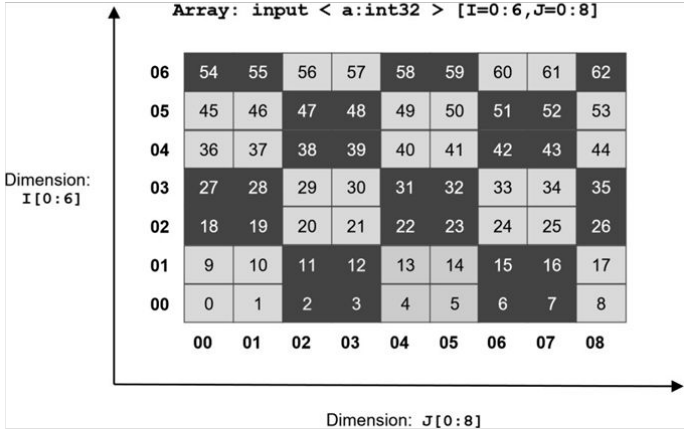


SciDB

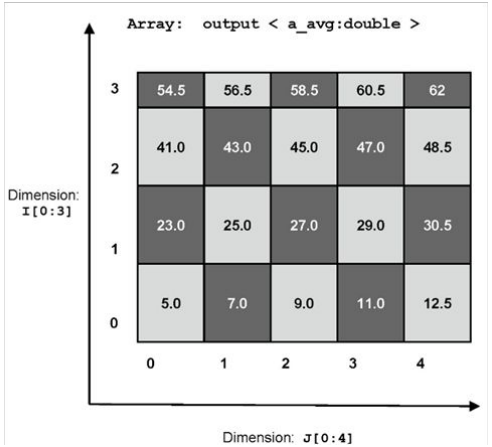
regrid (input, 2, 2, avg (a) as a_avg)





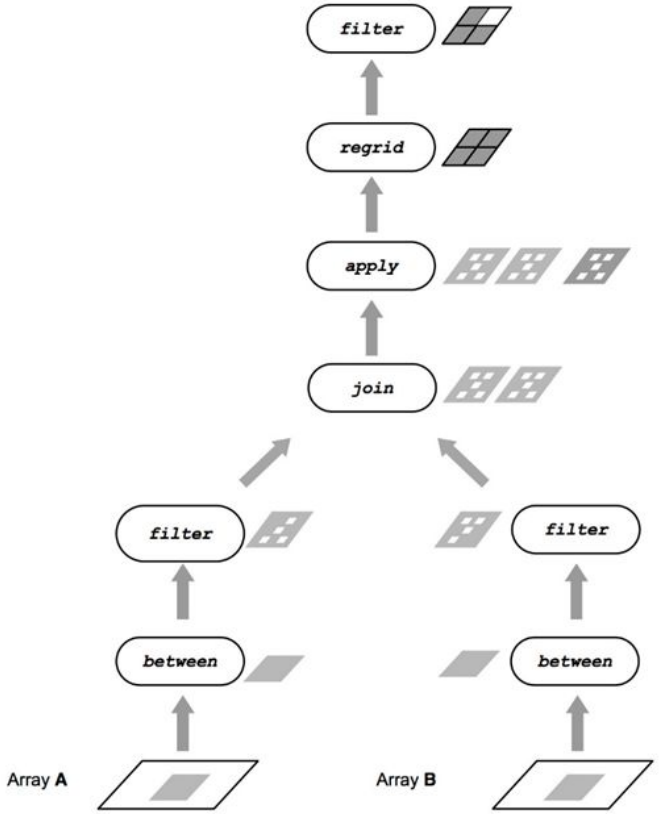






Query Evaluation

- A query is executed as a pipeline of array operators
- The system derives an efficient parallel execution plan that can:
  - reorder operators
  - introduce auxiliary steps
  - replace operator sequences with logically equivalent but more efficient alternatives.
- The execution engine works as a data pump
  - intermediate results are not materialized unless necessary
  - data are streamed through the operator pipeline from storage to the final result
- This execution style reduces unnecessary I/O and supports efficient multidimensional processing



NewSQL Databases

Why NewSQL?



- Classical relational DBMS excel at:
  - ACID transactions
  - SQL ecosystem
  - mature tooling
  - business workloads
- But scaling traditional SQL systems is hard:
  - vertical scaling has limits
  - application-level sharding is complex
  - failover and geo-replication are difficult
- NoSQL solved scale, but often sacrificed:
  - joins
  - strong consistency
  - standard SQL
  - transactional semantics

→ NewSQL = SQL + ACID + horizontal scalability

Keep SQL, rebuild the engine

What Is NewSQL?

- Distributed relational database systems designed for scale-out OLTP
- Typical properties:
  - SQL interface
  - relational model
  - ACID transactions
  - horizontal scalability
  - automatic partitioning / sharding
  - replication and fault tolerance
- Main goal:
  - preserve DBMS functionality known from traditional relational systems
  - while supporting high throughput and distributed operation









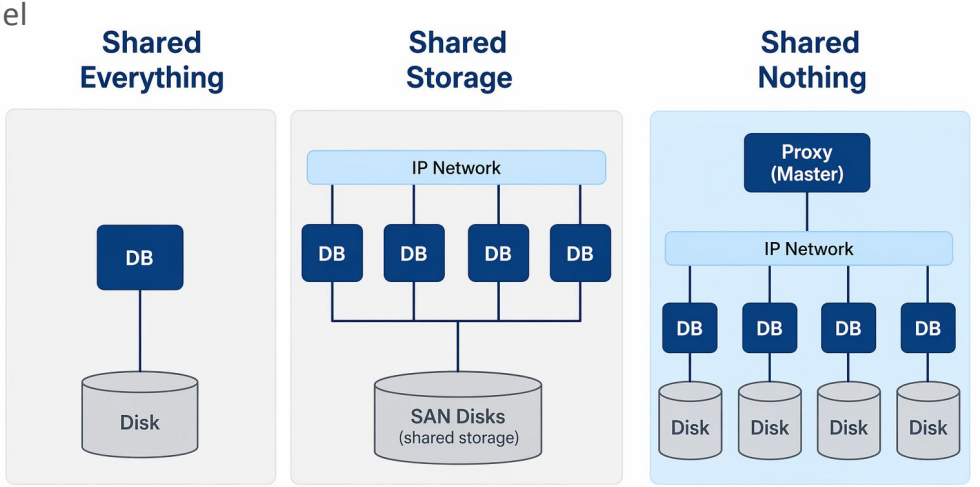
Main Approaches

- Two main approaches:
  - Distributed systems which add advantages of relational model + ACID
    - built from scratch
    - combine relational model, SQL, ACID, and horizontal scalability
    - examples: VoltDB (Volt Active Data), CockroachDB, Spanner, YugabyteDB
  - Traditional relational DBMS extended toward horizontal scalability
    - preserve relational foundations
    - add distributed storage and scale-out architecture
    - examples: TiDB, SingleStore, TokuDB
- Common goal:
  - relational model
  - SQL support
  - ACID transactions
  - horizontal scalability

Two engineering paths, one goal

Shared-Nothing Architecture

- Each node has its own CPU, memory, and local storage
- Nodes communicate over the network
- Data are split into partitions and distributed across nodes
  - Each node processes its local data in parallel
- Benefits:
  - horizontal scalability
  - parallel query execution
  - fault isolation
  - elastic cluster growth
- Main challenge:
  - coordination across nodes
  - distributed transactions
  - rebalancing after failures or scaling



Scale is achieved by dividing data and work across independent machines

Distributed Transactions

- If rows are on different nodes, the system must coordinate commit
- Typical techniques:
  - Two-Phase Commit (2PC)
    - A coordinator asks all participating nodes to prepare, then orders commit or abort for the transaction
  - Consensus protocols - e.g. Raft, Paxos
    - Make sure replicas agree on the same committed data
- Guarantees:
  - atomic commit across nodes
  - isolation between concurrent transactions
  - durability of committed changes
  - consistency constraints preserved
- Main cost: network round trips

```
BEGIN;

UPDATE accounts
SET balance = balance - 100
WHERE id = 1;

UPDATE accounts
SET balance = balance + 100
WHERE id = 2;

COMMIT;
```

ACID consistency in NewSQL is the same as in classical DBMS; what changes is that the system must preserve it across multiple nodes

NewSQL from the User Perspective

- For users, NewSQL often looks like a classical SQL database
- Familiar features:
  - `CREATE / ALTER / DROP TABLE`
  - `INSERT / UPDATE / DELETE`
  - `SELECT, JOIN, GROUP BY`
  - indexes
  - views
  - transactions
  - constraints
- What changes internally:
  - distributed storage
  - automatic partitioning
  - replication
  - coordination across nodes

```
CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT,
  amount DECIMAL(10,2)
);

SELECT customer_id, SUM(amount)
FROM orders
GROUP BY customer_id;
```

Why Still Use Classical SQL Systems?

- NewSQL solves scale-out transactional workloads
- But many applications do not need:
  - distributed clusters
  - elastic scale-out
  - complex cross-node coordination
- Classical SQL DBMS are often better for:
  - smaller and medium deployments
  - simpler administration
  - lower hardware and operational cost
  - mature ecosystem and tooling
  - predictable single-node performance
  - low-latency local transactions
  - workloads that fit on one server

If one server is enough, a classical DBMS may be the better choice

Vector Databases

Vector Databases







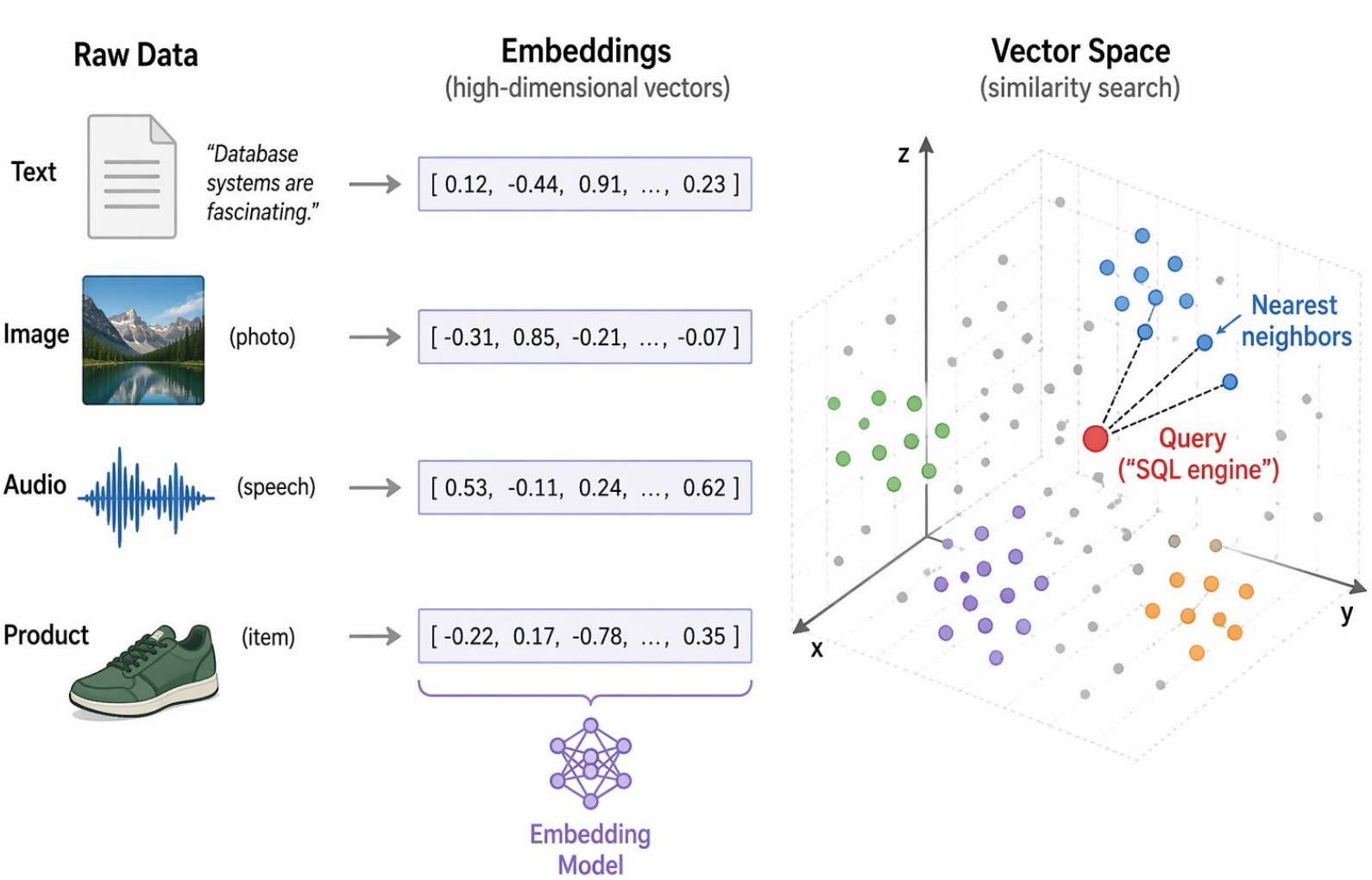
- Modern AI systems often work with embeddings instead of exact keys
  - text embeddings
  - image embeddings
  - audio embeddings
  - user / product representations
- Core task: similarity search in high-dimensional space
- Typical queries:
  - semantic search - find documents whose embeddings are nearest to the query embedding
  - recommendation - find products / users / items nearest to a user or item embedding
  - duplicate detection - find objects with almost identical embeddings
  - clustering support - group vectors that are close to each other

Not all queries ask for equality - many ask for similarity

What Is an Embedding?

- An embedding is a dense numerical representation of an object in a vector space
  - E.g. word, sentence, document, image, audio segment, user or product profile, …
- The representation is learned so that semantically or structurally similar objects are mapped to nearby vectors
- Typical properties:
  - fixed dimensionality
  - real-valued coordinates
  - model-dependent meaning of dimensions (no human interpretation)
  - similarity captured by distance or angle in vector space
- Common similarity measures:
  - cosine similarity
  - Euclidean distance
  - scalar product
- Typical dimensions: hundreds to thousands of components

Embeddings transform complex objects into vectors that support similarity-based retrieval


Raw Data
Embeddings
(high-dimensional vectors)
Vector Space
(similarity search)
Text
"Database systems are fascinating."
[0.12, -0.44, 0.91, …, 0.23]
Image
(photo)
[-0.31, 0.85, -0.21, …, -0.07]
Audio
(speech)
[0.53, -0.11, 0.24, …, 0.62]
Product
(item)
[-0.22, 0.17, -0.78, …, 0.35]
Embedding Model
z
x
y
Nearest neighbors
Query ("SQL engine")

Similarity in Vector Space

- Vector search does not ask whether two objects are identical
- It asks how close their vector representations are
- Intuition:
  - smaller distance → more similar objects
  - smaller angle → more similar direction
  - higher inner product → vectors point in more similar directions
- The choice of metric matters because it affects:
  - ranking of results
  - index design
  - query semantics
- Different embedding models are often designed with specific similarity measures in mind

Why Classical Indexes Are Not Enough?

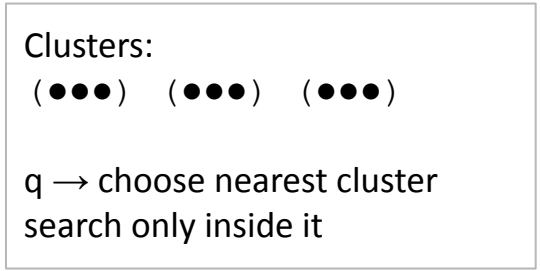
- Classical indexes (B+ trees, hash indexes) are designed for:
  - exact lookup
  - range queries
  - ordered access
- Vector search has a different goal:
  - find nearest neighbors (approximation)
  - rank objects by similarity
  - work in high-dimensional space
- Why this is difficult:
  - there is no natural total ordering of vectors
  - exact pruning becomes much less effective in many dimensions
  - brute-force comparison against all vectors is too expensive at scale
- Consequence: vector databases use specialized indexing methods

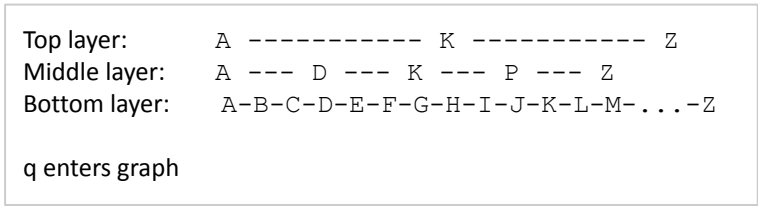
Similarity search requires different access methods than classical SQL lookup

How Vector Indexes Work

- Goal: avoid comparing the query vector q with all stored vectors
- Flat index
  - stores all vectors directly
  - query is compared with every vector → exact result
  - simple, but expensive for large collections
- IVF (Inverted File Index)
  - vectors are first grouped into coarse clusters
  - at query time, the system finds the nearest cluster centers
  - then it searches only vectors in those selected clusters
  - faster than flat search, but may miss some true nearest neighbors
- HNSW (Hierarchical Navigable Small World)
  - vectors are organized as a multi-layer graph
  - each vector is connected to several nearby vectors
  - search starts from upper layers and moves to lower, denser layers
  - usually very fast and accurate, but memory-intensive







Qdrant



- Open-source vector database for similarity search
- Core data model:
  - collection = named set of searchable points
  - point = id + vector + optional payload
  - payload = metadata used for filtering
- Each collection defines:
  - vector dimensionality
  - distance metric (cosine, Euclidean, dot product,…)
- Typical use cases: semantic search, recommendation, image similarity, duplicate detection, ..
- REST API
- Indexing:
  - HNSW for approximate nearest-neighbor search
  - payload indexes for efficient metadata filtering
- Interesting feature: combines vector similarity with structured filters

Storing and Querying Vectors

Qdrant

```
{
  "collection_name": "papers",
  "vectors": {
    "size": 768,
    "distance": "Cosine"
  }
}
```

PUT /collections/papers

```
{
  "id": 101,
  "vector": [0.12, -0.44, 0.91, "..."],
  "payload": {
    "title": "Graph databases for fraud detection",
    "year": 2024,
    "type": "paper"
  }
}
```

PUT /collections/papers/points

```
{
  "vector": [0.10, -0.41, 0.88, "..."],
  "limit": 5,
  "filter": {
    "must": [
      { "key": "type", "match": { "value": "paper" } },
      { "key": "year", "range": { "gte": 2022 } }
    ]
  }
}
```

POST /collections/papers/points/query

- Find 5 nearest vectors
- Search only among papers
- Restrict to papers published since 2022

Deployment and Updates

Qdrant

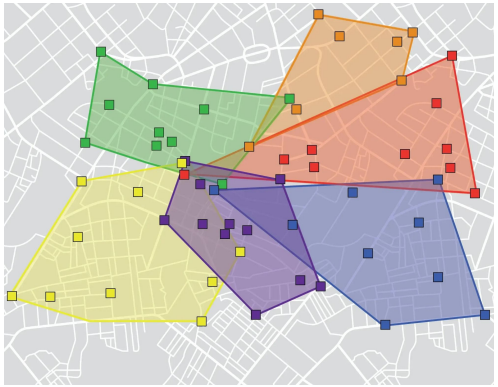
- Qdrant deployment:
  - single-node or distributed cluster
  - collections split into shards
  - shards can be replicated
- Updates:
  - update vector
  - update payload fields
  - delete point
  - upsert point by id
- Not a NewSQL-style transactional DBMS:
  - optimized for vector search
  - not for complex ACID transactions across records

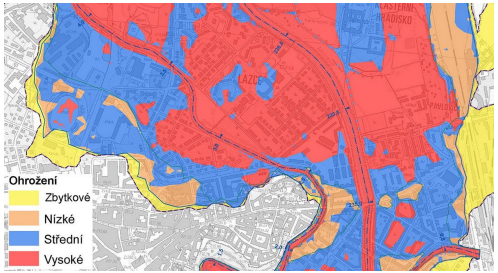
(Geo)Spatial Databases

Why (Geo)Spatial Databases?

- Many applications manage data with location, shape, or geometry
- Examples:
  - maps
  - GPS tracks
  - buildings
  - roads
  - parcels
  - delivery zones
  - sensor coverage areas
- Typical questions:
  - What is nearby?
  - What intersects?
  - What is inside a region?
  - What is the best route?
  - What changed over time?







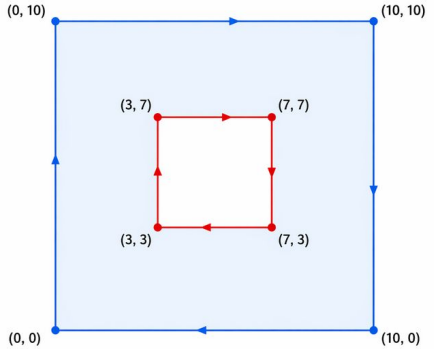
Not all data live in rows — some live in space

What Makes Spatial Data Different?

- Spatial objects are not only values — they have geometry
- Important properties:
  - position - where the object is
  - shape - point, line, polygon, surface
  - size - area, length, perimeter
  - relationships - intersects, contains, touches, overlaps
  - distance - how far objects are from each other
- Examples:
  - Store A is inside district Prague 6
  - Road R intersects river Vltava
  - Hospital H is 1.2 km from user
- Geometry must be stored, indexed and queried respectively

Spatial data add coordinates and topology to ordinary records

Spatial Data Types



- Spatial DBMS introduce geometry as native data types
- Common types:
  - Point
    - e.g., store location, GPS position, sensor node, …
  - LineString
    - e.g., road, route, river, …
  - Polygon
    - e.g., district, parcel, building footprint, lake, …
  - MultiPoint / MultiLineString / MultiPolygon
    - Collections of geometries
    - e.g. city district consisting of several separated parts

```
POINT(14.4208 50.0880)

LINESTRING(
  14.40 50.08,
  14.43 50.09,
  14.46 50.10)

POLYGON(
  (0 0, 10 0, 10 10, 0 10, 0 0),
  (3 3, 7 3, 7 7, 3 7, 3 3)
)
```

Spatial Database Systems

- Typical capabilities:
  - geometry types
  - spatial predicates
  - R-tree indexes
  - nearest-neighbor search
  - routing / map analytics
- Deployment styles:
  - embedded
  - enterprise SQL server
  - cloud-native analytics
  - big geospatial platforms
- Distributed spatial systems also exist







Many modern DBMS support geometry — some specialize in it

PostGIS



- Open-source geospatial extension for PostgreSQL
- Adds native spatial support to a relational DBMS
- Core features:
  - geometry and geography data types
  - spatial predicates - intersects, contains, within, touches
  - spatial operations - distance, area, length, buffer, union, intersection
  - spatial indexes - GiST / SP-GiST, efficient nearest-neighbor search
- Why important:
  - combines SQL + transactions + GIS functionality
  - integrates with PostgreSQL ecosystem
  - widely used in industry and research
- Can also be used in scaled deployments via PostgreSQL clustering / sharding tools

PostGIS turns PostgreSQL into a full spatial database system

Storing and Querying Spatial Data

```
CREATE TABLE pharmacies (
  id INT PRIMARY KEY,
  name TEXT,
  location GEOMETRY(Point, 4326)
);

INSERT INTO pharmacies VALUES
(1, 'Pharmacy A', ST_SetSRID(ST_Point(14.4208, 50.0880), 4326)),
(2, 'Pharmacy B', ST_SetSRID(ST_Point(14.4550, 50.0750), 4326));
```

```
SELECT name
FROM pharmacies
ORDER BY location <-> ST_SetSRID(ST_Point(14.4300, 50.0830), 4326)
LIMIT 1;
```

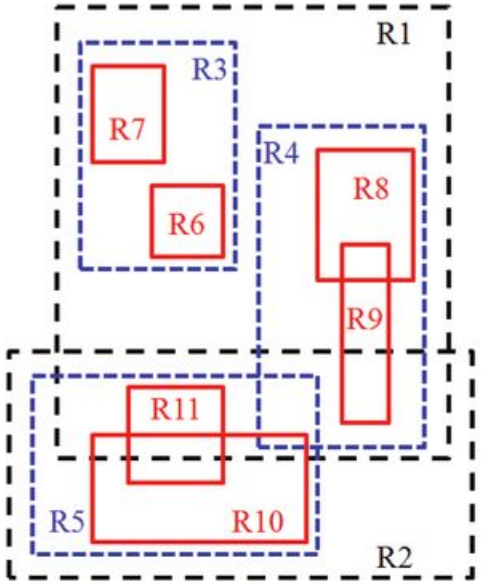
`<->` = distance ordering operator
`4326` is SRID — coordinate system identifier

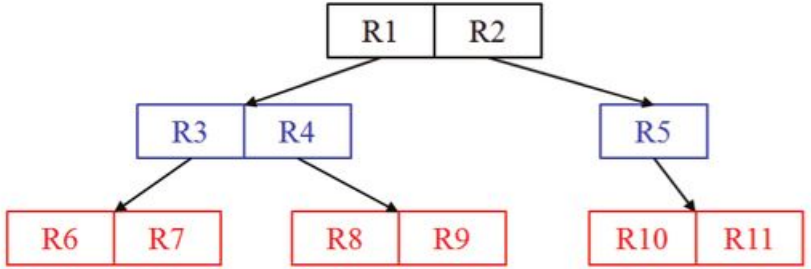
Spatial Indexes



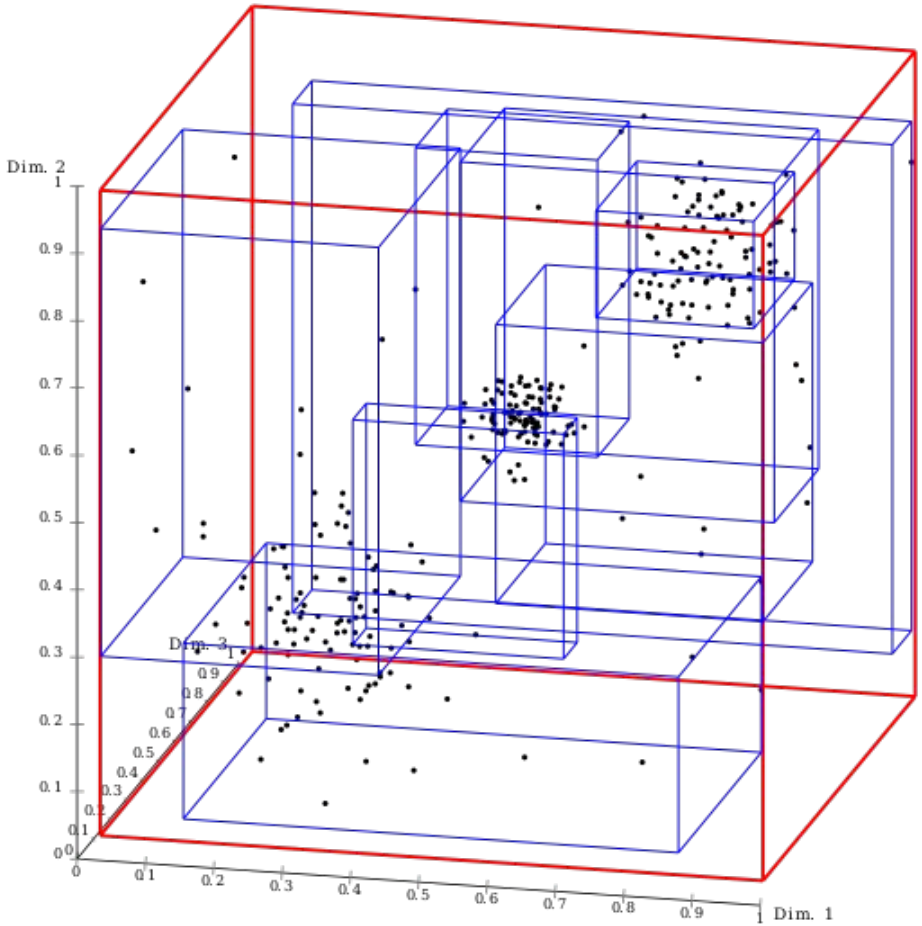
- Classical B-tree indexes sort one-dimensional values
- Spatial objects need multidimensional indexing
- Common structures:
  - R-tree
    - groups nearby rectangles
    - popular in many systems
  - GiST
    - PostgreSQL indexing framework
    - used by PostGIS
  - Quadtree - recursively divides space into quadrants
  - Grid index - partitions map into cells
- Accelerates:
  - nearby search
  - containment
  - overlap queries
  - spatial joins

R-tree





R-tree



https://en.wikipedia.org/wiki/R-tree

R-tree definition

- Every leaf contains between ***m*** and ***M*** index records
  - ***M*** = the maximum number of entries in a node
  - ***m <= M/2***
    - Not fixed to 50%! (We can choose.)
- Every inner node other than root contains between ***m*** and ***M*** entries
- The root has at least 2 children unless it is a leaf
- Each node contains a set of entries ***E*** consisting of
  - ***E.p*** – pointer ***p*** to the child node (inner node) or spatial object identifier (leaves)
  - ***E.I*** – ***n***-dimensional bounding box ***I***
  - For each record ***B***, ***E.I*** is the minimum bounding rectangle
- All leaves appear at the same level
- The tree is balanced

Conclusion

One Size Does Not Fit All

- Different workloads require different abstractions
- Key lesson:
  - data model matters
  - query model matters
  - consistency needs differ
  - physical design follows workload
- Reality:
  - modern architectures often combine multiple systems
  - polyglot persistence is common

References

- SciDB Documentation https://paradigm4.atlassian.net/wiki/spaces/scidb/overview?homepageId=339587 8913
- Volt Active Data Documentation https://docs.voltdb.com/
- Qdrant Documentation https://qdrant.tech/documentation/
- PostGIS Documentation https://postgis.net/documentation/

Lecture 10

Evolution Management in Big Data Systems

NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Change is the Norm

Systems Are Designed Once, Used for Years

- A database system is usually designed for:
  - current schema
  - current workload
  - current scale
  - current hardware
  - current regulations
- But during lifetime, all of them change
- Mismatch: design-time assumptions ≠ runtime reality
  - This mismatch is source of evolution pressure

At deployment	After 3 years
schema fits app	new entities needed
workload known	analytics added
hardware fixed	cloud migration
users limited	global traffic
one team	many teams

Good systems create new requirements

Forces That Break Original Design

- Application-driven
  - new entities
  - new attributes
  - new relationships
  - new APIs
- Workload-driven
  - new queries
  - higher concurrency
  - stricter response-time requirements
  - analytical workloads added
- Infrastructure-driven
  - cloud move
  - distributed deployment
  - move old data to cheaper storage
- Governance-driven
  - keeping old data for years
  - history of data changes required
  - GDPR

Business success often destroys original schema assumptions

Example: Product Growth

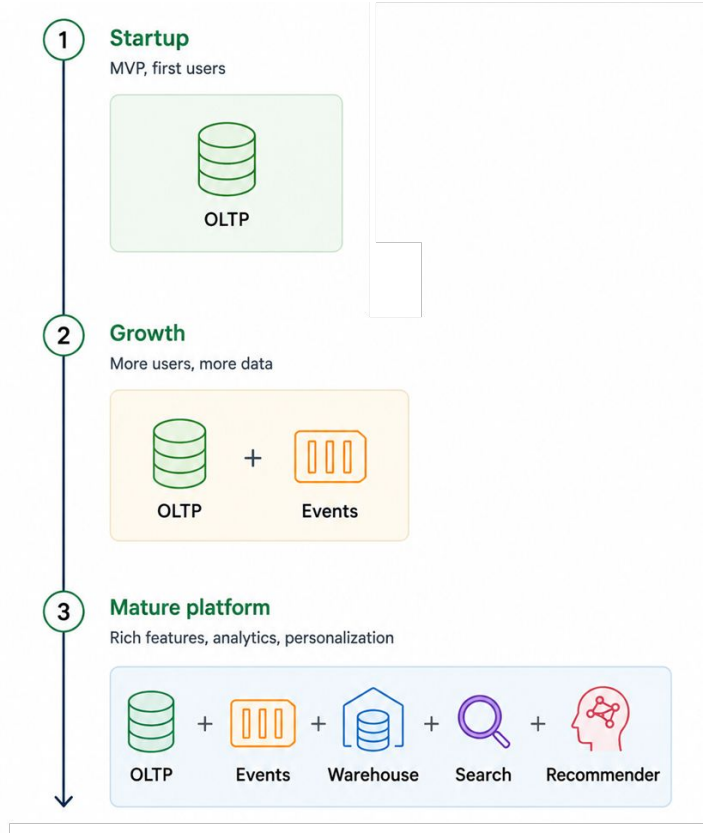
v1 Startup:

```
Customer(id, name)
Order(id, customer_id, total)
```

v2 Growth:

New needs:

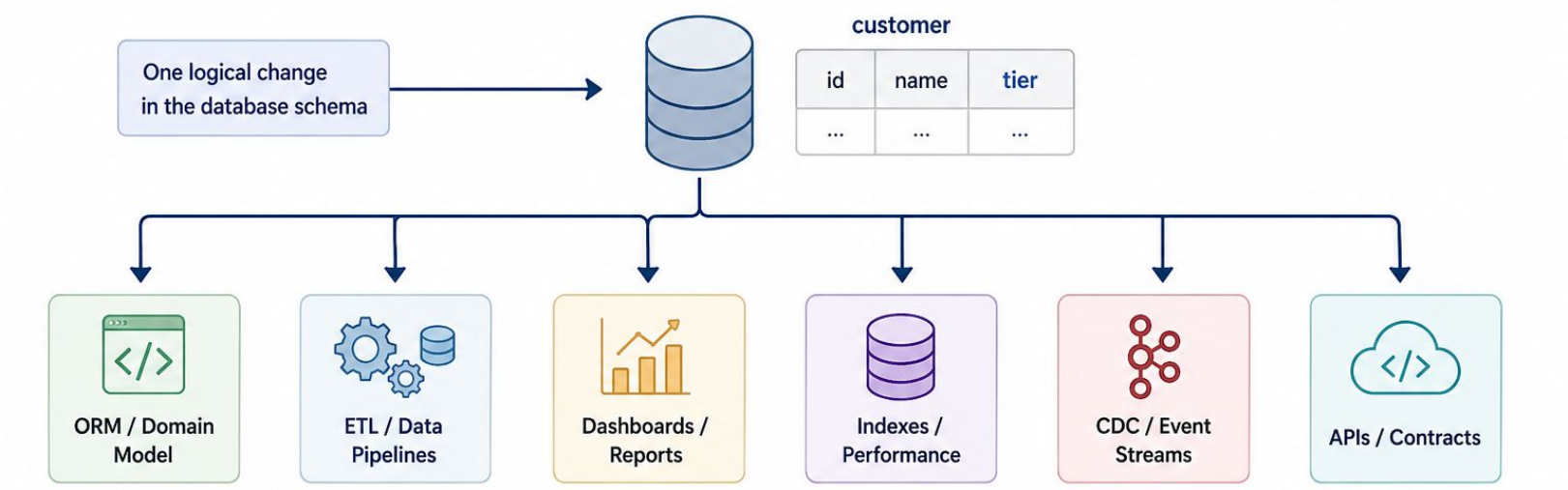
- multiple addresses
- loyalty program
- product recommendations
- event history
- …



v3 Mature Platform: OLTP DB + event log + warehouse + search engine + recommender

Why Small Change Has Large Impact?

```
ALTER TABLE customer ADD COLUMN tier TEXT;
```



One logical change → many physical changes

Structural Evolution Complexity

Schema change	Only metadata update	Physical rewrite	Operational impact
add nullable column	yes often	no	low
delete nullable column	maybe	no	medium/high
delete non-nullable column	yes often	yes often	high
rename column	yes	no	medium
widen type	maybe	maybe	medium
split table	no	yes	high
merge tables	no	yes	high

All `ALTER TABLE` statements are not equal

Why Schema Exists?

- Schema provides:
  - Semantics
    - What data means?
  - Integrity
    - What values are legal?
  - Optimization
    - How queries can be executed efficiently?
- Without schema, hidden schema still exists in code
  - Schemaless is not schema-free!

```
{ "id" : 1,   "name" : "Ann"}
{ "id" : 2,   "name" : "Bob", "tier" : "gold"}
{ "id" : "3", "fullName" : "Eva"}
```

Problems:

- missing fields
- type inconsistency
- naming drift
- semantic drift

Schemaless moves schema control from DBMS to application layer

Schema-on-Write vs Schema-on-Read

- Schema-on-Write
  - Validate before storing
  - Examples: PostgreSQL, MySQL, …
- Schema-on-Read
  - Validate during query/use
  - Examples: JSON data lake, logs on S3, …

Model	Validation Time	Advantage	Risk
schema-on-write	ingest	consistency	less flexibility for new data
schema-on-read	query time	flexibility	late failures

Error delayed is not error removed

Why Change Is Often Ignored

Motivation Problem

- Teams rewarded for:
  - launch now
  - deliver feature
  - hit benchmark
- Not rewarded for:
  - future migration ease
  - safe rollback
  - clear data ownership

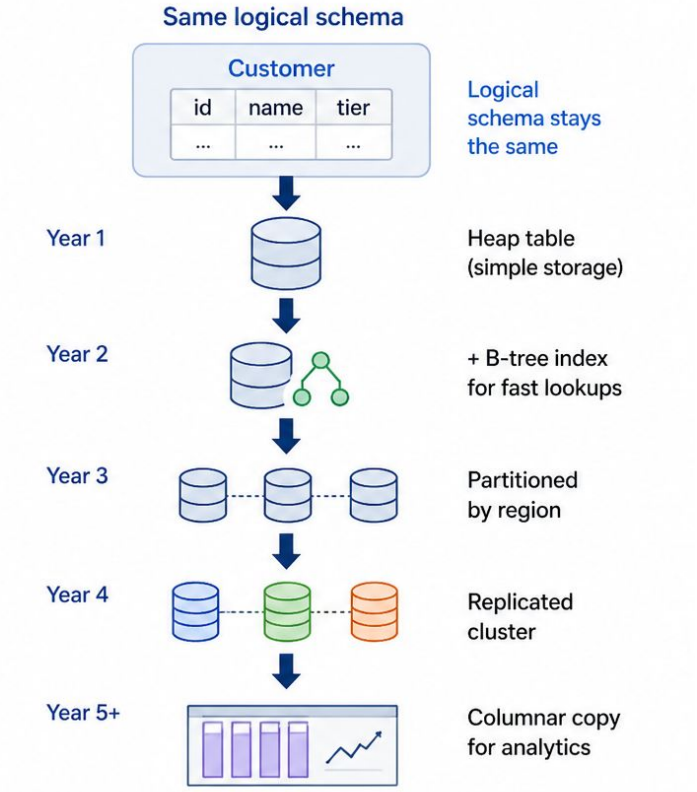
- Result: technical debt in data layer
  - duplicated customer data
  - overloaded generic columns
  - JSON blob inside SQL table
  - no primary keys
  - undocumented transformations
  - …
- Debt reduces future evolution speed

Example:
- Initial workload: OLTP reads + inserts
- Later: joins + aggregates + ML features + search + graph traversal

Systems become harder to change every year

Physical Design Evolution

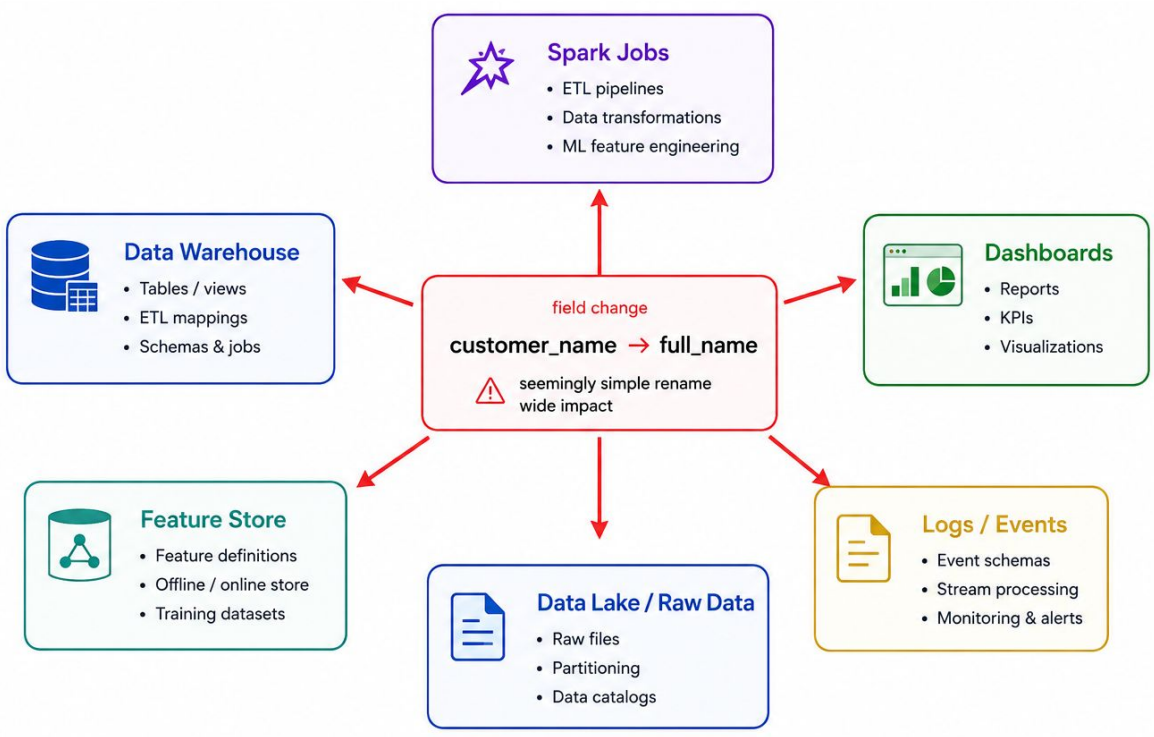
New requirement	Physical design change
faster point reads	add / change indexes
larger table size	partition or shard data
faster joins / reports	denormalize or materialize views
faster full scans	columnar layout and compression
higher write throughput	batching / write-optimized layout
global low latency	replication and geo-distribution
lower storage cost	cold storage and tiering



Logical schema may stay stable, but physical design must evolve

Why Evolution Is Harder in Big Data Systems?

- Scale effects
  - TB/PB rewrite cost
  - many pipelines
  - distributed consistency
  - no downtime allowed
- Heterogeneity effects
  - multiple engines
  - multiple schemas
  - multiple owners



Evolution Is the Real Challenge

- Not storing data, but keeping data systems useful while assumptions change
  - Evolution management = controlled adaptation under constraints
- If system lives long enough:
  - schema changes
  - workload changes
  - scale changes
  - ownership changes

A design without evolution strategy is temporary design

Migration vs Transformation

Two Different Dimensions of Evolution

Migration	Transformation
changes running infrastructure	changes data structure
operational focus	semantic focus
usually persistent	often repeatable
affects deployment/runtime	affects structure/content
system-oriented	data-oriented

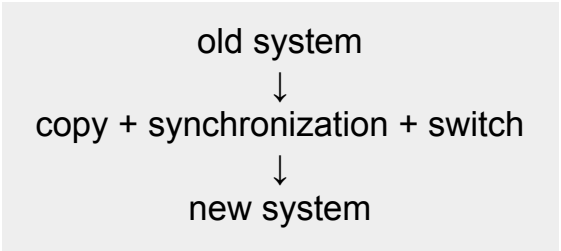
Typical examples

- Migration
  - MongoDB ⟶ Cassandra
  - single-node ⟶ distributed cluster
  - on-prem ⟶ cloud
- Transformation
  - JSON ⟶ relational rows
  - CSV ⟶ Parquet
  - field rename / nesting change

Real evolution projects usually require both

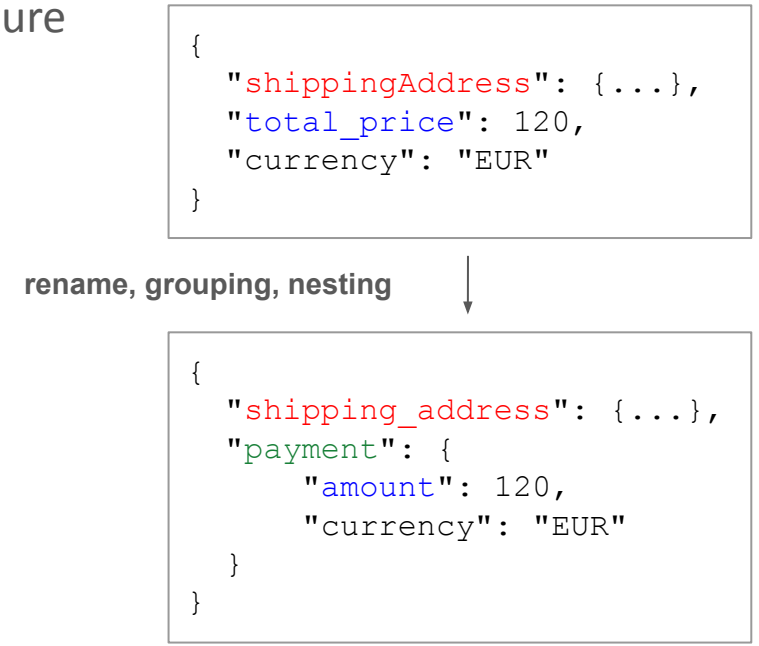
Migration Changes Operational State

- Goal: Move production system to new stable state
- Typical concerns
  - downtime
  - rollback
  - consistency
  - replication lag
  - application compatibility
  - deployment ordering
- Examples
  - PostgreSQL ⟶ cloud-managed PostgreSQL
  - MongoDB ⟶ Cassandra read model
  - on-prem Hadoop ⟶ cloud lakehouse



Transformation Changes Representation

- Goal: Preserve meaning while changing structure
- Typical operations
  - rename field
  - split field
  - merge fields
  - move
  - nesting
  - flatten structure
  - aggregate values



Migration Is Rarely 1:1 Copy

- Wrong assumption: new database = old schema in different engine
- Reality
  - Different systems optimize different workloads
- Therefore migration often includes:
  - schema redesign
  - denormalization
  - re-partitioning
  - query rewriting
  - new consistency assumptions

MongoDB document
↓
Cassandra query-oriented table

Concrete Evolution Scenario

MongoDB v1
↓ **transformation**
MongoDB v2
↓ **migration + redesign**
Cassandra

New requirements

- structured payment info
- clearer naming
- scalable city-based reads

```
{
  "order_id": 101,
  "customer": "Ann",
  "shippingAddress": {
      "city": "Prague"
  },
  "total_price": 120,
  "currency": "EUR"
}
```

MongoDB v1

```
{
  "order_id": 101,
  "customer": "Ann",
  "shipping_address": {
      "city": "Prague"
  },
  "payment": {
      "amount": 120,
      "currency": "EUR"
  }
}
```

MongoDB v2

```
CREATE TABLE orders_by_city (
    city text,
    order_id int,
    customer text,
    amount decimal,
    currency text,
    PRIMARY KEY (city, order_id)
);
```

Cassandra query-oriented read model

Evolution changes both data structure and access patterns

Query Rewriting

```
db.orders.find({
  "shippingAddress.city":"Prague"
})
```

MongoDB v1

→

```
db.orders.find({
  "shipping_address.city":"Prague"
})
```

MongoDB v2

```
SELECT *
FROM orders_by_city
WHERE city='Prague';
```

Cassandra query-oriented read model

- Cassandra partition key = `city` → distributed reads optimized for city queries
  - Workload-driven evolution

Schema evolution propagates into query evolution

Why Big Data Evolution Is Difficult

Traditional migration

- 100 GB
- 1 database
- night maintenance window

- Possible approach:
  - stop ⟶ migrate ⟶ restart

Big Data migration

- 10 TB – PB scale
- multiple distributed replicas
- streaming ingestion
- 24/7 availability
- many connected systems

- Large distributed systems usually cannot stop during evolution

Big Data evolution becomes distributed systems engineering

Operational Techniques for Big Data Evolution

Rolling deployment

- System is updated gradually instead of full shutdown
  - node1 → updated
  - node2 → updated
  - node3 → updated
  - …
- Goal
  - reduce downtime
  - reduce risk
  - verify incrementally

Continuous validation

- Migration correctness is checked during execution
- Typical checks:
  - row/document counts
  - missing values
  - schema consistency
  - source vs target comparison
- Important: TB/PB migrations may run for hours or days

Rollback planning

- Migration may fail
- Need strategy for safe recovery
- Typical approaches:
  - snapshots
  - dual-write phase
  - replay from logs
  - fallback to previous cluster

Evolution Patterns

Manage transition from old system/state to new one

Pattern	Basic idea	When useful	Main risk
Big bang	stop old system → migrate → start new system	small systems, maintenance window possible	downtime, hard rollback
Blue/green	old and new systems run in parallel; traffic is switched later	safer production switch	data divergence
Dual write	application writes temporarily to both old and new targets	zero-downtime transition	inconsistent writes
CDC (Change Data Capture) sync	changes from old DB are captured and replayed to new system	large data, continuous writes	lag, schema compatibility

Transformation Patterns

Manage change of data shape, format, or meaning

Pattern	Basic idea	Example	Main risk
ETL (Extract, Transform, Load)	transform before loading to target	Mongo JSON → Cassandra rows	transformation bug before load
ELT (Extract, Load, Transform)	load raw data first, transform inside target	logs → lake → warehouse tables	raw zone grows, late errors
Batch rewrite	rewrite existing historical data	CSV → Parquet	expensive full scan/copy
Stream enrich	transform events as they arrive	add customer segment to order events	late/missing reference data
Materialized view refresh	recompute derived read model	`orders_by_city` table	stale or inconsistent view

Tooling for Evolution Management

Why Evolution Tooling Exists?

- Manual evolution does not scale
- Single script may work for:
  - 1 DB, 1 owner, 1 deployment
- Modern systems require coordination of:
  - schema versions, pipelines, replicas, streaming systems, connected analytics and services, dashboards, …

Evolution becomes managed workflow, not isolated `ALTER TABLE`

Tooling must provide:

Capability	Why needed
version tracking	reproducibility
orchestration	ordered execution
synchronization	consistency
rollback	failure recovery
observability	debugging
validation	correctness

Categories of Evolution Tooling

ALTER TABLE customer ADD COLUMN tier
↓
Debezium propagates updates
↓
NiFi transforms downstream records
↓
Airflow coordinates dependent workflows
↓
Iceberg stores historical snapshots

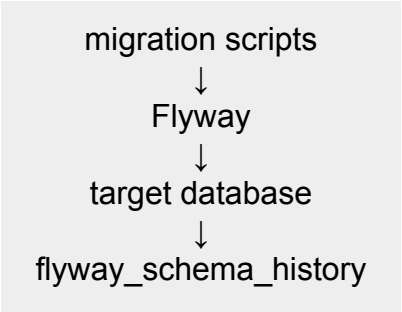
Category	Main role	Typical systems
schema migration	controlled schema updates	Flyway Liquibase
CDC / replication	incremental propagation	debezium
dataflow / ETL	transformation pipelines	APACHE nifi
orchestration	workflow coordination	Apache Airflow
schema governance	schema compatibility & contracts	CONFLUENT
versioned storage	snapshot-based versioned storage	ICEBERG DELTA LAKE

Different tools manage different dimensions of evolution

Flyway



- Open-source schema migration system
- Developed originally by Redgate
- Main goal:
  - Keep database schema synchronized with application evolution
  - Philosophy: Database schema should evolve similarly to source code
    - versioned
    - reproducible
    - automated
    - tracked in Git
- Supported databases: PostgreSQL, MySQL, MariaDB, Oracle, SQL Server, SQLite, Cassandra, Snowflake, …



https://www.red-gate.com/products/flyway/

What Flyway Actually Does?

- Migration discovery
  - scans migration directory
- Version tracking
  - stores executed versions in `flyway_schema_history`
    - version, checksum, execution timestamp, success/failure
- Ordering
  - enforces deterministic execution order
- Execution control
  - runs only missing migrations
- Checksum validation
  - detects modified historical migrations
- Failure tracking
  - records failed migrations
- Transactional execution
  - avoids partial schema updates
- Repeatable migrations
  - reruns selected objects when changed
- CI/CD (Continuous Integration and Continuous Delivery) integration
  - automates schema deployment

What Flyway Does NOT Do



- Flyway does NOT:
  - redesign schema automatically
  - rewrite queries
  - migrate downstream systems
  - synchronize Kafka topics
  - validate semantic correctness
  - manage ETL pipelines

Flyway tracks, orders, validates, and executes schema migrations

Example: Multi-Step Evolution

Flyway

Business change

- Application introduces:
  - customer loyalty tiers
  - tier-based analytics
  - faster queries for premium customers

Example migration directory:

```
V1__create_customer.sql
V2__add_customer_tier.sql
V3__backfill_customer_tier.sql
V4__create_customer_tier_index.sql
```

Required evolution

V2 — schema extension

```
ALTER TABLE customer
ADD COLUMN tier TEXT DEFAULT 'basic';
```

V3 — backfill historical data

```
UPDATE customer
SET tier='gold'
WHERE total_spent > 10000;
```

V4 — add optimized index

```
CREATE INDEX idx_customer_tier
ON customer(tier);
```

How Flyway Uses Migration Directory?

Project repository

```
project/
├── src/
├── tests/
└── db/migrations/
    └── V1__create_customer.sql
```

1. Developer A adds loyalty tiers = creates `V2__add_customer_tier.sql`
2. Developer B backfills historical tiers = creates `V3__backfill_customer_tier.sql`
3. Developer C optimizes analytics queries = creates `V4__create_customer_tier_index.sql`
4. New staging database currently contains V1 applied
5. CI/CD runs: `flyway migrate`
6. Flyway process
   - Step 1: scan migration directory → finds: V1, V2, V3, V4
   - Step 2: check `flyway_schema_history` → applied V1
   - Step 3: compute missing migrations → missing = V2, V3, V4
   - Step 4: execute in deterministic order V2 → V3 → V4
   - Step 5: update `flyway_schema_history` → applied V1, V2, V3, V4

Schema evolution becomes part of software engineering workflow

Limits of Schema Migration Systems

- Flyway works well for:
  - controlled relational schema evolution
  - ordered DDL execution
  - reproducible deployments
  - CI/CD integration
- Schema migration alone does NOT solve:
  - downstream propagation
  - streaming synchronization
  - warehouse updates
  - search index refresh
  - cache invalidation
  - schema drift in events
  - multi-system coordination

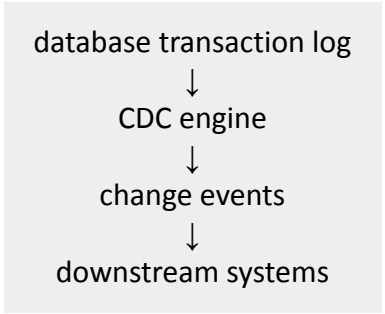
```
ALTER TABLE customer ADD
COLUMN tier
```

may additionally require:

- Kafka topic update
- ETL modification
- dashboard update
- feature store update
- materialized view refresh

One logical change = many downstream updates
How do we propagate changes continuously across distributed systems?

CDC (Change Data Capture)



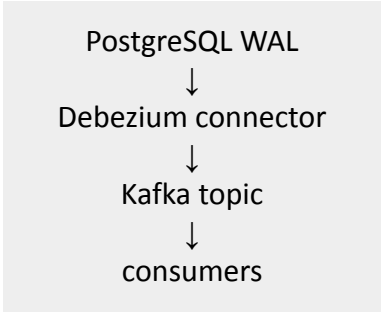
- Main idea: capture only incremental changes
  - Instead of repeatedly copying entire database
- Typical captured operations
  - `INSERT`
  - `UPDATE`
  - `DELETE`
- Transaction log sources: PostgreSQL WAL (Write-Ahead Logging), MySQL binlog, MongoDB oplog, …
- CDC enables:
  - near real-time synchronization
  - incremental migration
  - zero-downtime evolution

CDC propagates changes, not full database snapshots

Debezium

debezium

- Open-source CDC platform
- Built on Kafka Connect ecosystem
- Supported databases: PostgreSQL, MySQL, MongoDB, SQL Server, Oracle, Cassandra
- Transforms database changes into event stream
- Events contain:
  - operation type
  - before/after state
  - timestamp
  - transaction metadata
  - schema metadata



Database changes become event stream

https://debezium.io/

Debezium Event Example

debezium

Original database update

```
UPDATE customer
SET tier='gold'
WHERE id=101;
```

→

Debezium event

```
{
  "op":"u",
  "source":"customer",
  "before":{
    "id":101,
    "tier":"silver"
  },
  "after":{
    "id":101,
    "tier":"gold"
  }
}
```

Downstream consumers may:

- refresh materialized views
- update Elasticsearch index
- invalidate cache
- update warehouse rows
- recompute features

One DB update may trigger <u>many</u> downstream updates

CDC During Migration

Traditional migration

stop system

↓

copy database

↓

start new system

- Works only for smaller systems

Big Data migration

- System cannot stop
- Need:
  - continuous writes
  - continuous reads
  - continuous synchronization

CDC-based migration

old database

↓

CDC stream

↓

new database **synchronized incrementally**

CDC enables near-zero downtime migration

CDC Limitations

- CDC solves propagation problem, but NOT:
  - semantic redesign
  - query rewriting
  - denormalization strategy
  - consistency conflicts
  - invalid source data
- Therefore CDC usually works together with:
  - schema governance
  - ETL/dataflow systems
  - validation pipelines
  - versioned schemas

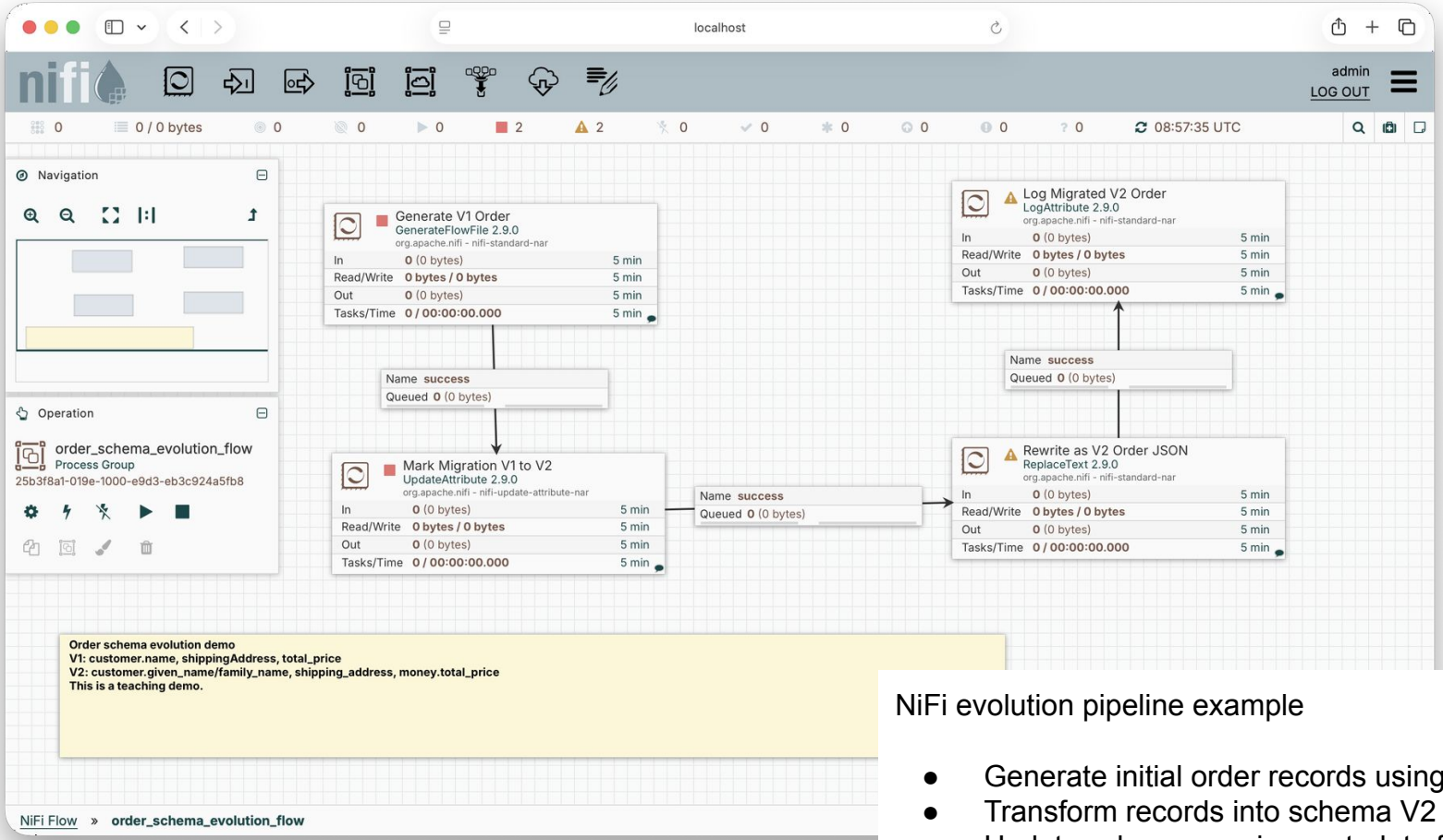
CDC propagates changes mechanically, not semantically

Apache NiFi

- Open-source visual dataflow system
  - Pipeline orchestration + transformation platform
- Main goal: build and manage dataflow pipelines visually
- Core philosophy: Data movement should be
  - observable
  - configurable
  - traceable
  - replayable
  - backpressure-aware
- Typical use cases: ETL pipelines, CDC processing, format conversion, system integration, migration pipelines, …

NiFi is flow-based engineering, not traditional programming

https://nifi.apache.org/



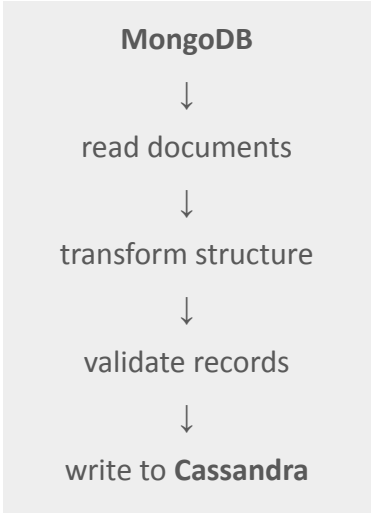
NiFi evolution pipeline example

- Generate initial order records using schema V1
- Transform records into schema V2 representation
- Update schema-version metadata from V1 to V2
- Log successfully migrated records

NiFi Core Model

- NiFi models dataflow as pipeline:
  - source → processors → queues → target systems
- **Processor**
  - Active operation:
    - E.g., read, transform, validate, route, write, …
  - Examples: QueryMongoDB, ConvertRecord, PutCassandraQL
- **Queue** - buffer between processors
  - Stores records temporarily
  - Supports: asynchronous execution, backpressure, retry, batching
- **FlowFile** - single moving data object
  - Contains: payload data + metadata attributes

Example migration pipeline



NiFi manages movement and transformation of records between systems

Why Visual Dataflow Matters

- Traditional script-based pipeline `python migrate.py`
- Problem:
  - hidden logic
  - difficult debugging
  - unclear dependencies
  - difficult replay
  - difficult observability
- NiFi approach: pipeline becomes explicit graph (that can be visualized)
- Why important in Big Data?
  - TB/PB migrations may run for hours or days
  - Need continuous monitoring

Evolution pipeline becomes observable runtime system

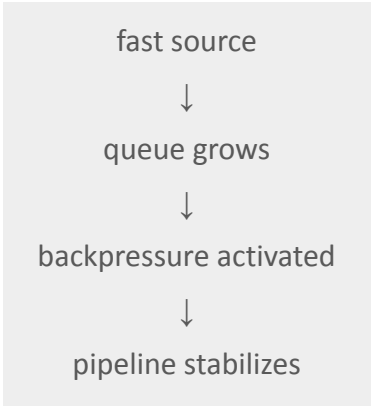
NiFi Provenance and Replay

- NiFi tracks:
  - where record came from
  - which processors modified it
  - where it was routed
  - whether delivery succeeded
- Without restarting entire migration failed records may be:
  - replayed
  - rerouted
  - revalidated
  - reprocessed

Replayability becomes critical at large scale

Backpressure and Flow Control

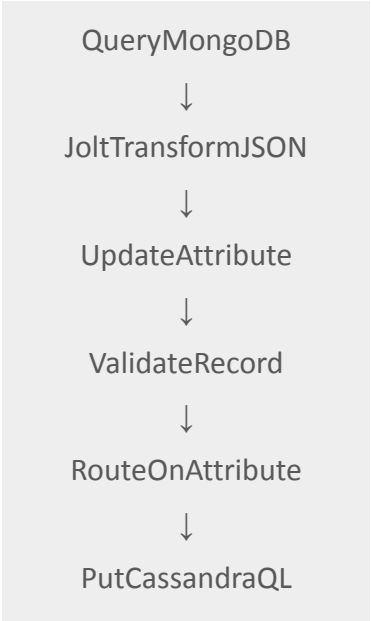
- Problem: target systems may become slower than source
  - Example: MongoDB export > Cassandra insert throughput
- Without flow control queues grow indefinitely:
  - memory pressure
  - node overload
  - crashes
  - unstable pipeline
- NiFi solution: backpressure
  - When queue exceeds threshold upstream processors slowed or paused



Distributed evolution pipelines must tolerate uneven throughput

Complete NiFi Migration Example

- Source: MongoDB v2 documents
- Goal: create Cassandra query-oriented read model
- Responsibilities
  - QueryMongoDB - reads source documents
  - JoltTransformJSON - reshapes nested JSON
    - Rename, nesting, flattening, …
  - UpdateAttribute - adds metadata/schema version
  - ValidateRecord - checks required fields/types
  - RouteOnAttribute - routes invalid records separately
  - PutCassandraQL - writes transformed rows



Migration pipeline combines:
transformation + validation + routing + loading

Flyway + Debezium + NiFi

- Modern evolution is multi-layer process
- Responsibility separation
  - Flyway - controls schema state
  - Debezium - captures incremental changes
  - NiFi - transforms and routes data

Flyway (schema evolution)

↓

Application writes new data

↓

Debezium captures changes

↓

NiFi transforms records

↓

warehouse / Cassandra / search updated

Schema Evolution

- Main problem: distributed systems evolve asynchronously
  - Producer, consumer, storage, and pipelines are rarely upgraded simultaneously
  - Old and new versions co-exist
- What is schema evolution?
  - Schema changes over time
  - Historical data must remain readable
  - Queries and pipelines must survive change
- **Safe changes** - usually backward-compatible
  - E.g., add nullable field, add field with default value, widen type, append optional nested object
- **Breaking changes** - may break existing consumers
  - E.g., remove field, rename field, incompatible type change, move field to different nesting level

Schema evolution is compatibility management under asynchronous deployment

Compatibility Types

```
old schema:
{
  "customer":"Ann"
}
```

→

```
new schema:
{
  "customer":"Ann",
  "tier":"gold"
}
```

- **Backward compatibility**
  - NEW consumer can process OLD data (consumer upgraded first)
  - e.g., new consumer handles missing tier using default/null.
- **Forward compatibility**
  - OLD consumer can process NEW data (producer upgraded first)
  - e.g., old consumer ignores unknown field
- Full compatibility
  - Both directions work

```
old schema:
{
  "customer":"Ann"
}
```

→

```
new schema:
{
  "customer": {
     "first":"Ann"
  }
}
```

- Breaking change
  - e.g., string ⟶ object change breaks consumers

Compatibility allows producers and consumers to be upgraded independently

Real Evolution Timeline

```
Day 1
producer v1
consumer v1
warehouse v1

Day 10
producer upgraded to v2

Day 20
some consumers still v1

Day 30
warehouse still expects v1 schema

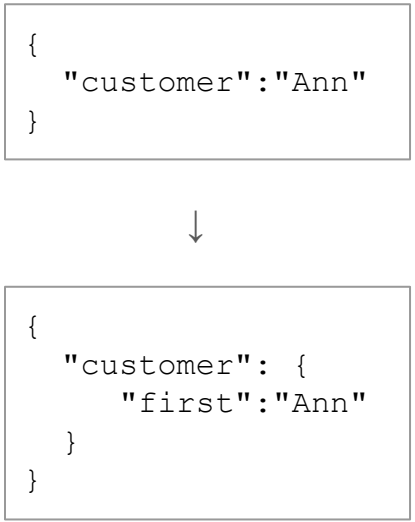
Day 45
historical files still contain v1
```

- Ecosystem contains mixed versions simultaneously
  - old events
  - new events
  - old consumers
  - new consumers
  - historical snapshots
  - partially upgraded pipelines
- Without compatibility:
  - deserialization failures
  - query failures
  - missing dashboards
  - broken ETL jobs
  - inconsistent analytics

Old schema versions may survive for years

Managing Schema Compatibility

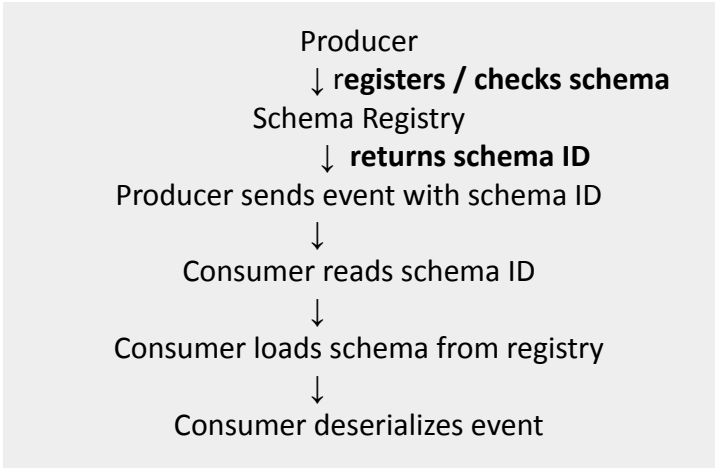
- Initial situation:
  - Application produces events
  - Another application reads them
  - Everything works
- New situation:
  - Developer changes event structure
  - And deploys new producer version
- Problem: older consumers still expect the initial schema
  - Possible consequences
    - pipeline crash
    - missing dashboards
    - broken analytics



Confluent Schema Registry

CONFLUENT

- Central service for managing schemas of exchanged events
- It does not store events
  - It stores schemas, their versions, and compatibility rules
- Main role
  - keeps event schemas centrally registered
  - assigns schema IDs
  - validates schema compatibility
  - prevents incompatible producer deployments
  - allows consumers to interpret events correctly
  - supports schema evolution over time
- Typical formats: Avro, Protobuf, JSON Schema



Schema Registry acts as compatibility gatekeeper

https://docs.confluent.io/platform/current/schema-registry/

Working with Schema Registry

CONFLUENT

- Step 1: Producer defines event schema structure
- Step 2: Producer registers schema in Schema Registry
- Step 3: Registry compares new schema and previous schema versions
  - Checks selected compatibility mode:
    - backward, forward, full
- Step 4: Registry assigns unique schema ID
- Step 5: Producer sends events
  - Events contain: schema ID + serialized data
- Step 6: Consumer resolves schema
  - Reads schema ID
  - Fetches schema from registry
  - Deserializes event correctly

```
{
  "type":"record",
  "name":"CustomerCreated",
  "fields":[
    {"name":"customer","type":"string"},
    {"name":"tier","type":["null","string"],"default":null}
  ]
}
```

```
{
  "id":17
}
```

```
{
  "schema_id":17,
  "data":{
    "customer":"Ann",
    "tier":"gold"
  }
}
```

Schemaless Does Not Mean Schema-Free

- Even without explicit schema applications still assume structure
- Consequences:
  - difficult analytics
  - invalid aggregations
  - query failures
  - inconsistent dashboards
  - broken pipelines
  - difficult indexing

```
{ "id" : 1,   "name" : "Ann"}
{ "id" : 2,   "name" : "Bob", "tier" : "gold"}
{ "id" : "3", "fullName" : "Eva"}
```

- Important observation: Schemaless systems trade upfront rigidity for long-term governance complexity

Schema still exists,
only moved into application code and pipelines

Principles of Safe Schema Evolution

- Prefer additive evolution than destructive rewrite
- Avoid immediate field removal
- Version explicitly
- Preserve backward compatibility

Typical safe evolution sequence

add new field

↓

dual-read phase

↓

migrate consumers

↓

stop writing old field

↓

remove old field later

Why Versioned Systems Exist?

- Traditional database thinking
  - Update data in place: old value → new value
  - Previous state disappears
- Problem in Big Data systems - what if we need:
  - rollback after failed migration
  - historical analytics
  - auditability
  - replay old pipelines
- Rewriting PB-scale data is expensive
- Therefore modern systems keep multiple versions of data simultaneously

Versioned systems treat data evolution as sequence of snapshots

Snapshot-Based Storage

- Main idea: table is not one mutable object
- Instead: table state = **snapshot**
  - Each snapshot represents consistent table version
- Traditional systems:
  - Rewrite existing files/pages
- Versioned systems:
  - Create new data files + update metadata pointer
  - Old files remain temporarily available
- Benefits:
  - rollback, reproducibility, time travel queries, concurrent readers, safer evolution

```
Snapshot 1
orders:
101 Prague
102 Brno
```

```
Snapshot 2
orders:
101 Prague gold
102 Brno   basic
```

Apache Iceberg



- Open table format for large analytical datasets
- Designed for: data lakes, lakehouses, PB-scale analytics
- Traditional approach: schema change → rewrite all data
  - e.g., add new column with default value
- Iceberg does NOT immediately rewrite all data:
  - Old files remain unchanged
  - New files use new schema
  - Metadata records current table version

schema change

↓

new snapshot metadata

↓

old files remain valid

↓

cheap evolution

https://iceberg.apache.org/

Apache Iceberg



- Why important for evolution?
- Large tables can evolve without:
  - rewriting entire dataset
  - blocking readers
  - stopping ingestion
- Example schema evolution
  - Old schema: (order_id, customer, total)
  - New schema: (order_id, customer, total, tier)
- Iceberg updates metadata instead of rewriting all historical files immediately
  - Example: It knows that old data do not have tier and returns null/default

In Big Data systems, metadata evolution is often cheaper than physical rewrite

Multi-Model Evolution Management

Evolution in Multi-Model Environments

- Typical ecosystem:
  - OLTP relational database
  - document database
  - search engine
  - graph system
  - data warehouse
- Same business entity (e.g., `customer`) appears multiple times:
  - PostgreSQL rows
  - MongoDB documents
  - Cassandra read models
  - Elasticsearch index
  - graph relationships

- One logical change must propagate into:
  - different schemas
  - different storage models
  - different query languages
  - different consistency assumptions

`shippingAddress → shipping_address`
must propagate into:

- SQL queries
- Mongo queries
- Cassandra tables
- Spark jobs
- dashboards

Different Systems Require Different Evolution Strategies

Relational systems

- Typical evolution:
  - ALTER TABLE
  - indexes
  - normalization changes
- Strong schema enforcement

Document databases

- Typical evolution:
  - partial schema evolution
  - mixed document versions
  - lazy migration
- Applications often handle compatibility

Column-family systems

- Typical evolution:
  - query-oriented redesign
  - denormalization
  - repartitioning
- Schema driven by workload

Search systems

- Typical evolution:
  - reindexing
  - analyzer changes
  - mapping evolution
- Often requires rebuilding indexes

Query Evolution Across Models

```
SELECT customer
FROM orders
WHERE city='Prague';
```

Relational query

```
db.orders.find({
 "shipping_address.city":"Prague"
})
```

MongoDB query

```
SELECT *
FROM orders_by_city
WHERE
city='Prague';
```

Cassandra query

- Same logical question, but:
  - different physical layouts
  - different access paths
  - different optimization assumptions

Evolution changes not only schemas, but also access strategies

Main Engineering Challenges

- Technical challenges
  - schema compatibility
  - distributed coordination
  - incremental synchronization
  - query rewriting
  - replay correctness
  - rollback safety
- Organizational challenges
  - multiple teams
  - unclear ownership
  - asynchronous deployments
  - inconsistent governance

⇒ Therefore evolution management simultaneously becomes

- distributed systems engineering
- data engineering
- governance problem
- operational problem

Systems evolve longer than originally expected

Conclusion

- Evolution is multidimensional: logical evolution + physical evolution + operational evolution
- Modern evolution requires ecosystem thinking
  - Changes propagate through databases, pipelines, replicas, analytics, search systems, …
- Therefore evolution management requires
  - compatibility management
  - incremental synchronization
  - observability
  - replayability
  - rollback capability
  - versioned storage

The hardest problem is not storing data,
but coordinating change across evolving distributed systems

References

- Carlo A. Curino, Hyun J. Moon, MyungWon Ham, Carlo Zaniolo: The PRISM Workbench: Database Schema Evolution Without Tears. ICDE 2009. https://www.cs.ucla.edu/~zaniolo/papers/ccurino-prismworkbench-demo.pdf
- Stefanie Scherzinger, Meike Klettke, Uta Störl: Managing Schema Evolution in NoSQL Data Stores. DBPL 2013. https://arxiv.org/abs/1308.0514
- Pramod J. Sadalage, Martin Fowler: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, 2012. https://martinfowler.com/books/nosql.html
- Flyway https://www.red-gate.com/products/flyway/
- Debezium https://debezium.io/
- Apache NiFi https://nifi.apache.org/
- Confluent Schema Registry https://docs.confluent.io/platform/current/schema-registry/
- Apache Iceberg https://iceberg.apache.org/