ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΜΣ Κυβερνοασφάλεια και Επιστήμη Δεδομένων
MSc Cybersecurity and Data Science
DEPT OF INFORMATICS
UNIVERSITY OF PIRAEUS

Διαχείριση Μεγάλων Δεδομένων

Big Data Management

Lecture 4 - NoSQL DBs

George Papastefanatos (gpapas@athenarc.gr)

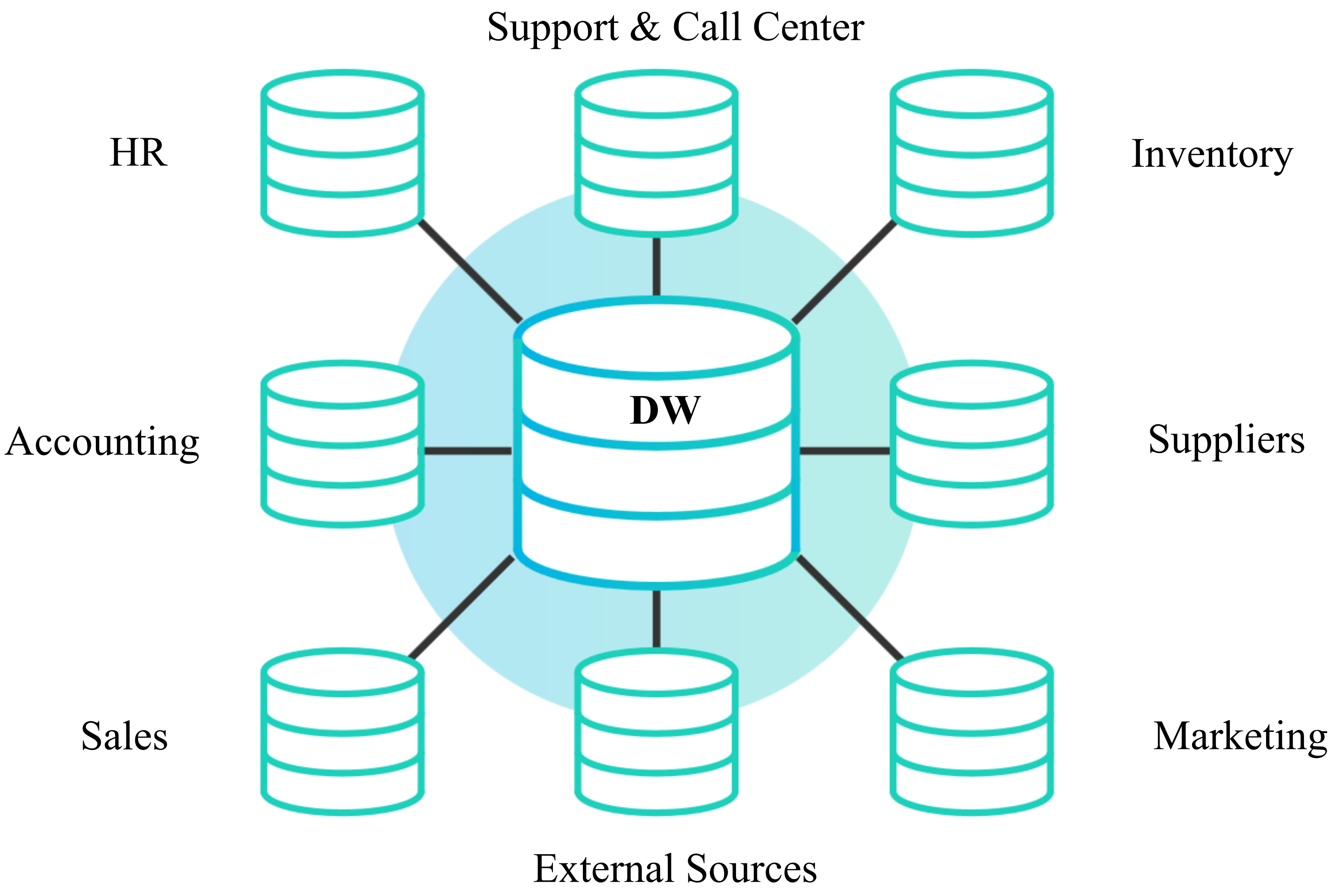
Principal Researcher @ ATHENA Research Center

ver. 11/2023

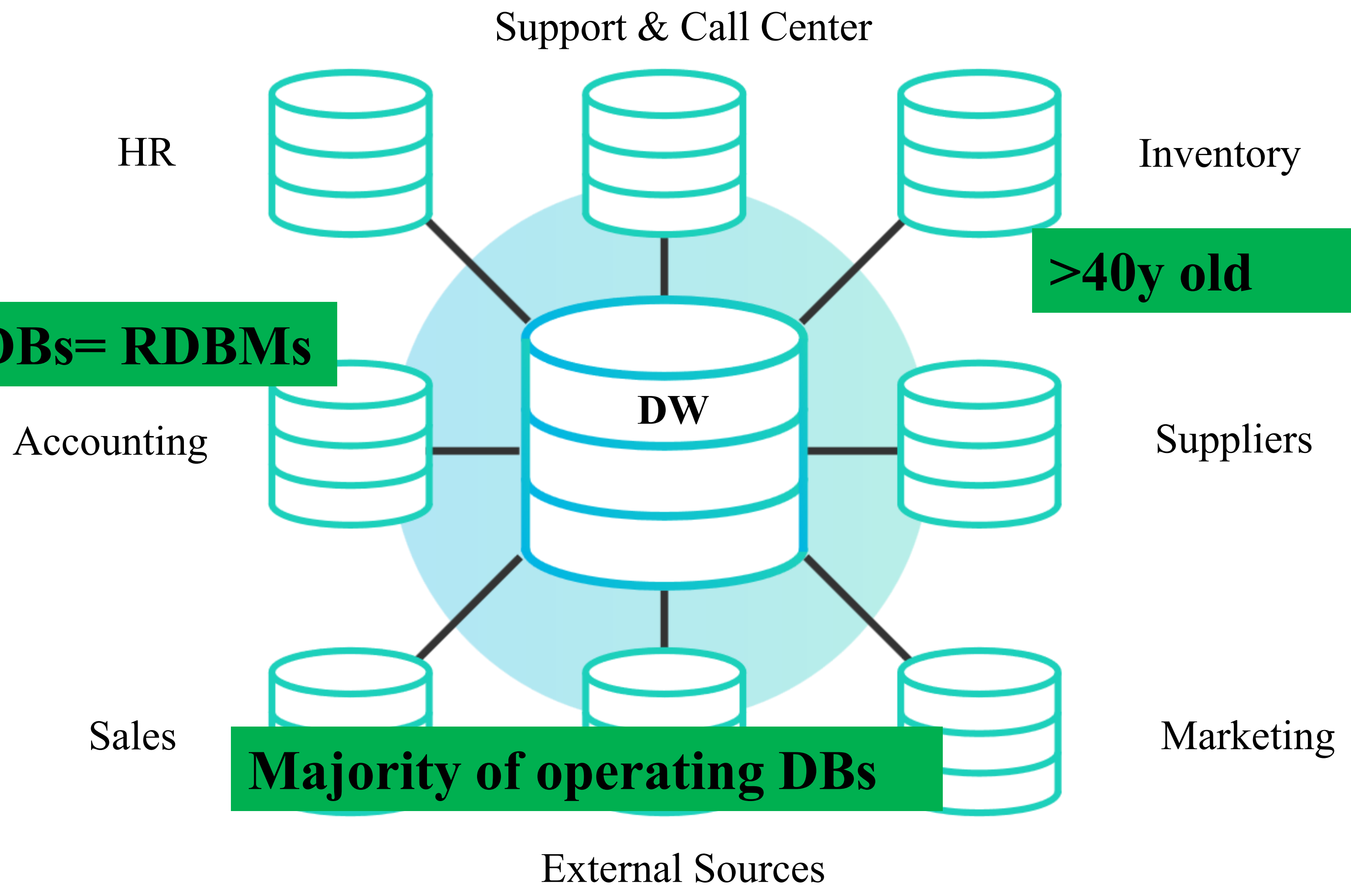
Lecture Outline

- Motivation
  - RDBMS characteristics
  - Current trends & RDBMS limitations
  - Cap Theorem
- NoSQL databases
  - Key-value stores
  - Document stores
  - Column stores
  - Graph stores
  - NewSQL DBs
- Overview of NoSQL Features

Relational databases



Relational databases



>40y old

SQL DBs= RDBMs

Majority of operating DBs

Relational Databases

- Data model
    - **Instance → database → table → row**
- Data access
    - **Selection** based on complex conditions, **projection**, **joins**, **aggregation**, derivation of new values, recursive queries,
    - **SQL** (*Structured Query Language*)

```
SELECT emp.name, dept.name
FROM emp INNER JOIN dept ON dept.id=emp.dept_id
WHERE dept.location = 'Athens'
```

- Formal: **Relational algebra**, relational calculi (domain, tuple)

$$\Pi_{emp.name,dept.name}(\sigma_{dept.location = \text{“Athens”}}(emp \bowtie dept))$$

Relational Databases - Representatives

ORACLE® DATABASE

Microsoft® SQL Server®

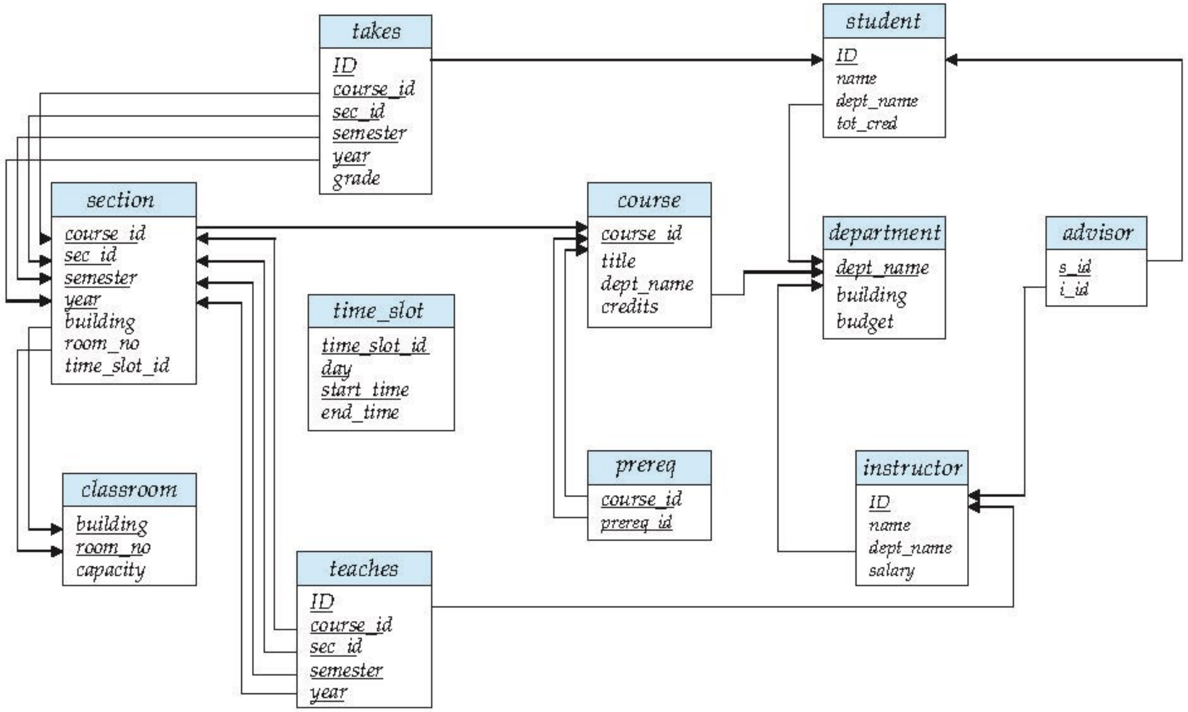
IBM DB2

MySQL®

PostgreSQL

… and many more

RDBMs Features – Normal Forms



Model Constraints

- **Functional** dependencies, 1NF, 2NF, 3NF, BCNF (Boyce-Codd normal form)

Objective

- Normalization of database schema to BCNF or 3NF, via decomposition or synthesis

Motivation

- Diminish **data redundancy**, **prevent update anomalies**
- However:
  - **Data is scattered into small pieces** (high granularity), and so
  - these pieces **have to be joined back together** when querying!

RDBMs Features – Transactions

Model

- **Transaction** = flat sequence of database operations (READ, WRITE, COMMIT, ABORT)

Objectives

- Enforcement of ACID properties
- **Efficient parallel / concurrent execution** (slow hard drives, …)

ACID properties

- **<u>A</u>tomicity** – partial execution is not allowed (all or nothing)
- **<u>C</u>onsistency** – transactions turn one valid database state into another
- **Isolation** – uncommitted effects are concealed among transactions
- **<u>D</u>urability** – effects of committed transactions are permanent

Where is Big Data?

- Social media and networks
  - …all of us are generating data
- Scientific instruments and e-Infrastructures
  - …producing all sorts of data, astronomical, biological, etc
- Mobile devices
  - …tracking social activity, mobility
- Internet of Things, sensors and networks
  - …machine-generated, measurements

Big Data Characteristics – The basic Vs

Volume (Scale)

- Data volume is increasing exponentially, not linearly
- Even large amounts of small data can result into Big Data

Variety (Complexity)

- Various formats, types, and structures
- (from semi-structured to unstructured multimedia)

Velocity (Speed)

- Data is being generated fast and needs to be processed fast

Veracity (Uncertainty)

- Uncertainty due to inconsistency, incompleteness, latency, ambiguities, or approximations

New Trends after 2000's

New trends

- **Heterogeneous** Data Models
- **Streaming Data,** fast OLTP
- **Distributed** Share-nothing systems
- **API data access, MapReduce, SPARK** and other programming models
- From Data **warehouses** to Data **Lakes**
- **Cloud** computing & **Edge** processing
- Large scale **machine learning**

Why not using RDBMs?

- Moto: One Size **Does not** Fit all

@ICDE2005

"One Size Fits All": An Idea Whose Time Has Come and Gone

Michael Stonebraker
Computer Science and Artificial Intelligence Laboratory, M.I.T., and StreamBase Systems, Inc.
stonebraker@csail.mit.edu

Uğur Çetintemel
Department of Computer Science Brown University, and StreamBase Systems, Inc.
ugur@cs.brown.edu

Abstract

The last 25 years of commercial DBMS development can be summed up in a single phrase: "One size fits all". This phrase refers to the fact that the traditional DBMS architecture (originally designed and optimized for business data processing) has been used to support many data-centric applications with widely varying characteristics and requirements.

In this paper, we argue that this concept is no longer applicable to the database market, and that the commercial world will fracture into a collection of independent database engines, some of which may be unified by a common front-end parser. We use examples from the stream-processing market and the data-warehouse market to bolster our claims. We also briefly discuss other markets for which the traditional architecture is a poor fit and argue for a critical rethinking of the current factoring of systems services into products.

1. Introduction

Relational DBMSs arrived on the scene as research prototypes in the 1970's, in the form of System R [10] and INGRES [27]. The main thrust of both prototypes was to surpass IMS in value to customers on the applications that IMS was used for, namely "business data processing". Hence, both systems were architected for on-line transaction processing (OLTP) applications, and their commercial counterparts (i.e., DB2 and INGRES, respectively) found acceptance in this arena in the 1980's. Other vendors (e.g., Sybase, Oracle, and Informix) followed the same basic DBMS model, which stores relational tables row-by-row, uses B-trees for indexing, uses a cost-based optimizer, and provides ACID transaction properties.

Since the early 1980's, the major DBMS vendors have steadfastly stuck to a "one size fits all" strategy, whereby they maintain a single code line with all DBMS services. The reasons for this choice are straightforward — the use of multiple code lines causes various practical problems, including:

- *a cost problem*, because maintenance costs increase at least linearly with the number of code lines;
- *a compatibility problem*, because all applications have to run against every code line;
- *a sales problem*, because salespeople get confused about which product to try to sell to a customer; and
- *a marketing problem*, because multiple code lines need to be positioned correctly in the marketplace.

To avoid these problems, all the major DBMS vendors have followed the adage "put all wood behind one arrowhead". In this paper we argue that this strategy has failed already, and will fail more dramatically off into the future.

The rest of the paper is structured as follows. In Section 2, we briefly indicate why the single code-line strategy has failed already by citing some of the key characteristics of the data warehouse market. In Section 3, we discuss stream processing applications and indicate a particular example where a specialized stream processing engine outperforms an RDBMS by two orders of magnitude. Section 4 then turns to the reasons for the performance difference, and indicates that DBMS technology is not likely to be able to adapt to be competitive in this market. Hence, we expect stream processing engines to thrive in the marketplace. In Section 5, we discuss a collection of other markets where one size is not likely to fit all, and other specialized database systems may be feasible. Hence, the fragmentation of the DBMS market may be fairly extensive. In Section 6, we offer some comments about the factoring of system software into products. Finally, we close the paper with some concluding remarks in Section 7.

2. Data warehousing

In the early 1990's, a new trend appeared: Enterprises wanted to gather together data from multiple operational databases into a data warehouse for business intelligence

RDBMs Limits

- Need for well-defined schemas
- Need for skilled DBA
- SQL and complex tuning
- Hard to make transactions scalable

Big Data on Clouds

- Everything is on the cloud
  - SaaS: Software as a Service
  - PaaS: Platform as a Service
  - IaaS: Infrastructure as a Service
- Processing paradigms
  - OLTP: Online Transaction Processing
  - OLAP: Online Analytical Processing
  - **…but also…**
  - **RTAP: Real-Time Analytic Processing – time to analysis is minimal**

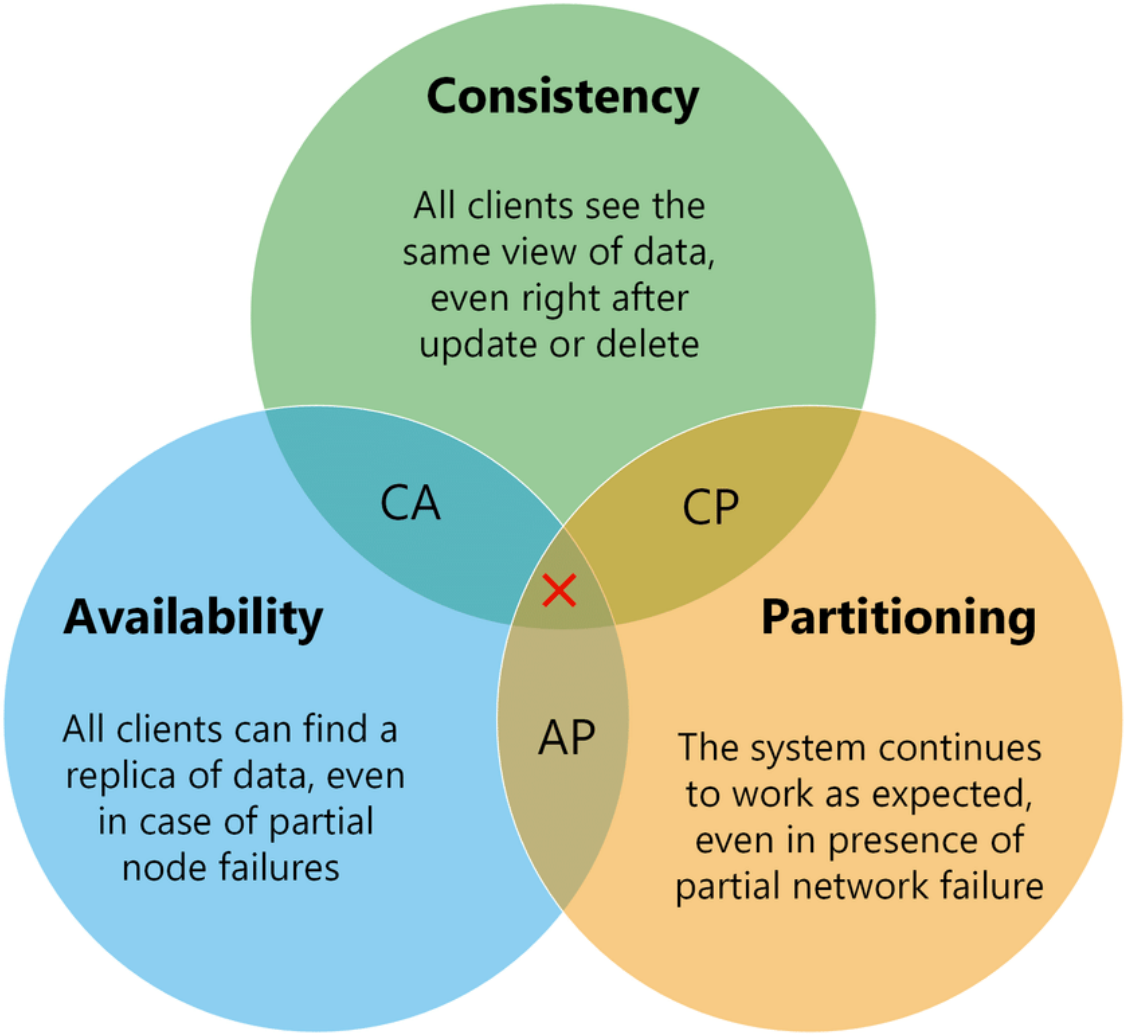
New Data assumptions

- Data format is **becoming unknown or inconsistent** (csv, json, text, compressed, …)
- Data **updates** are no longer frequent, mostly **additions in streams**
- Data is expected to **be replaced**
- Linear growth → **unpredictable exponential** growth
- Strong **consistency is no longer** mission-critical
- **Read requests** prevail write requests

CAP Theorem

CAP Theorem

- **Any distributed data store can only provide TWO of the THREE properties**
- History
  - At the PODC 2000 conference, Brewer (UC Berkeley) conjectures that one can have only two properties at the same time
  - In 2002, Gilbert and Lynch (MIT) proved the conjecture, which becomes a theorem

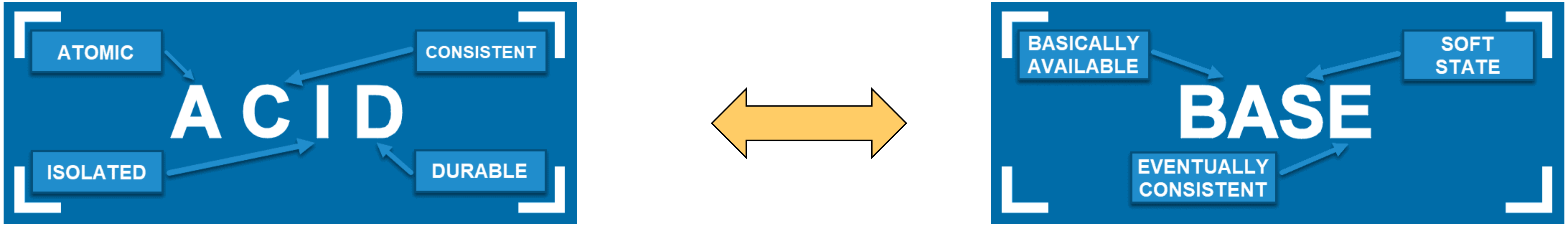


CAP Theorem: https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

Two types of transactions

- Polemical topic
  - The CAP theorem states that it is impossible to **achieve both consistency and availability** in a **partition tolerant distributed system** (i.e., a system which continues to work in cases of temporary network loss).
  - Argument used by NoSQL to justify their lack of ACID properties
  - But has nothing to do with scalability
- Two different points of view
  - Relational databases – ACID Transactions
    - **Consistency** is essential
  - Distributed systems – BASE Transactions
    - High **availability** is essential

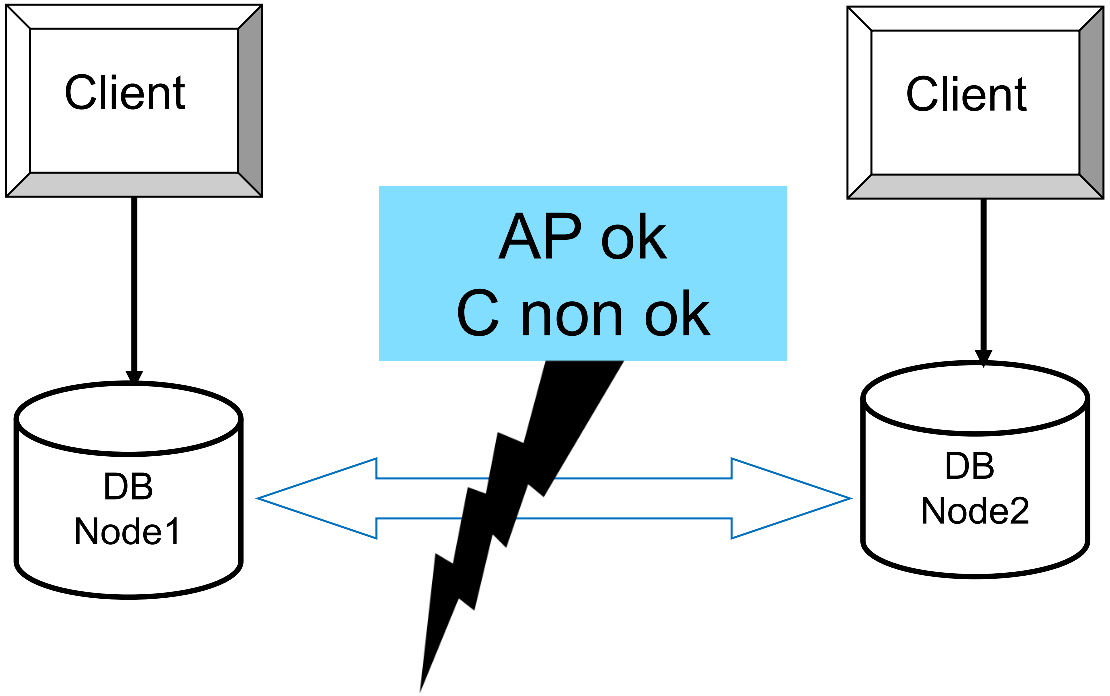
Strong vs Eventual Consistency



- Strong consistency (ACID)
  - All nodes see the same data values at the same time
- Eventual consistency (BASE)
  - **Basic Availability**: The database appears to work most of the time.
  - **Soft-state**: Stores don’t have to be write-consistent, nor do different replicas have to be mutually consistent all the time. Some nodes may see different data values at the same time
  - **Eventual consistency**: Stores exhibit consistency at some later point (e.g., lazily at read time). If we stop injecting updates, the system reaches strong consistency.

Symmetric, Asynchronous Replication

- How do achieve eventual consistency
  - After reconnection (and resolution of update conflicts), consistency can be obtained



BASE properties

- BASE properties are much looser than ACID guarantees
- A BASE data store
  - values **availability**
  - doesn’t offer **guaranteed consistency of replicated data at write time**.
  - Overall, the BASE consistency model provides a less strict assurance than ACID: data will be **consistent in the future**, at read time

Lecture Outline

- Motivation
  - Big Data Characteristics
  - Current trends & RDBMS limitations
- **NoSQL databases**
  - Key-value stores
  - Document stores
  - Column stores
  - Graph stores
- NewSQL DBs

NoSQL Databases

- What does NoSQL actually mean?
- A bit of history …
  - 1998
    - First used for a relational database that omitted usage of SQL for data access.
  - 2009
    - First used during a conference to advocate non-relational databases
- So?
  - Not: no to SQL
  - Not: not only SQL
- NoSQL is an accidental term with no precise definition

What does **NoSQL** actually mean?

- **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 DEFINITION:** *Next Generation Database Management Systems mostly addressing <u>some of the points:</u> being **non-relational, distributed, open-source** and **horizontally scalable.***
- The original intention has been **modern web-scale database management systems**. The movement began early 2009 and is growing rapidly. Often more characteristics apply such as: **schema-free, easy replication support, simple API, eventually consistent / BASE** (not ACID), a **huge amount of data** and more.
- The misleading term "nosql" (the community now translates it mostly with "**not only sql**") should be seen as an alias to something like the definition above.

Source: https://hostingdata.co.uk/nosql-database/

NoSQL (Not Only SQL)

- Specialized data model
  - Key-value, column-based, document, graph
- Query-based and API-based data access & manipulation
- Trade relational DBMS properties
  - Full SQL, ACID transactions, data independence
- For
  - **Simplicity** (schema-free, few or no constraints, basic API)
  - **Scalability** and performance – deployed over **distributed** environment
  - **Flexibility** for the programmer (integration with programming language)

NB: SQL is just a language and has nothing to do with the story

RDBMS vs NoSQL Overview

	SQL	NoSQL
Data storage	**Stored in a relational model**, with rows and columns. Rows contain all of the information about one specific entry/entity, and columns are all the separate data points.	The term "NoSQL" encompasses a host of databases, each with **different data storage models**. The main ones are: **document, graph, key-value and columnar**.
Schemas and Flexibility	**Each record conforms to a fixed schema and integrity constraints**, meaning the columns must be decided and locked before data entry and each row must contain data for each column. This can be amended, but it involves altering the whole database and going offline.	**Schemas can be dynamic**. Information can be added on the fly, and each 'row' (or equivalent) doesn't have to contain data for each 'column'.
Scalability	**Scaling is vertical**. In essence, more data means a bigger server, which can get very expensive. It is possible to scale an RDBMS across multiple servers, but this is a difficult and time-consuming process.	**Scaling is horizontal, meaning across servers**. They can be cheap commodity hardware or cloud instances, making it a lot more cost-effective than vertical scaling. Many NoSQL technologies also **distribute data across servers automatically**.
ACID Compliancy	The vast majority of relational databases are ACID compliant.	Varies between technologies, but many NoSQL solutions **sacrifice ACID compliancy for performance and scalability**

Source: https://dataconomy.com/2014/07/sql-vs-nosql-need-know/

NoSQL Approaches

- Core types
  - **Key-value** stores
  - **Document** stores
  - **Wide column** (column family, column oriented, …) stores
  - **Graph** databases
  - **Multimodel**
- Non-core types
  - **Object** databases
  - Native **XML** databases
  - **RDF** stores
  - …

Were there much before NoSQL
Sometimes presented as NoSQL
But not really scalable

KEY-VALUE STORES

Key-Value Stores

- Data model
  - The most simple NoSQL database type
    - Works as a simple hash table (mapping)
  - **Key-value pairs**
    - Key (id, identifier, primary key)
    - Value: binary object, black box for the database system
- Query patterns
  - Create, update or remove value for a given key
- **Get value** for a given key Characteristics
  - Simple model ⇒ **great performance, easily scaled, …**
  - Simple model ⇒ **not for complex queries nor complex data**

Key-Value Stores

- **Suitable use cases**
  - Session data, user profiles, user preferences, shopping carts, …
    - i.e. **when values are only accessed via keys**
- When not to use
  - **Relationships** among entities
  - Queries requiring **access to the content of the value part**
  - **Set operations** involving multiple key-value pairs

Key-Value Stores

redis

MEMCACHED

Amazon DynamoDB
Fast, flexible NoSQL database service

hazelcast

AEROSPIKE

EHCACHE

riakKV

SimpleDB

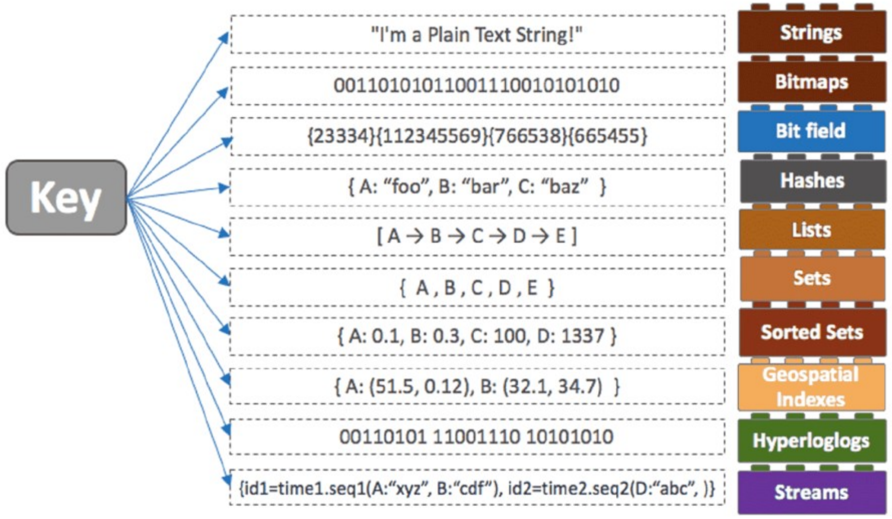
ORACLE
BERKELEY DB

ArangoDB

Redis

Redis is not a *plain* key-value store, supports different kinds of values.

- Key : Redis keys are binary safe, this means that you can use any binary sequence as a key, from a string like "foo" to the content of a JPEG file
  - **Very long keys are not a good idea.** A key of 1024 bytes is a bad idea not only memory-wise, but also because the lookup of the key in the dataset may require several costly key-comparisons. Even when the task at hand is to match the existence of a large value, hashing it (for example with SHA1) is a better idea, especially from the perspective of memory and bandwidth.
  - **Very short keys are often not a good idea**. There is little point in writing "u1000flw" as a key if you can instead write "user:1000:followers".
  - **Try to stick with a schema**. For instance "object-type:id" is a good idea, as in "user:1000". Dots or dashes are often used for multi-word fields, as in "comment:1234:reply.to" or "comment:1234:reply-to".



Redis Basic Commands

SET key to hold the string value. If key already holds a value, it is overwritten, regardless of its type

```
redis>  SET mykey "Hello" → "OK"
```

GET retrieves the values of the key. If key is nonexisting nil is returned.

```
redis>  GET nonexisting →(nil)
redis>  SET mykey "Hello" → "OK"
redis>  GET mykey → "Hello"
```

GETDEL gets the value of key and deletes the key.

```
redis>  SET mykey "Hello" → "OK"
redis>  GETDEL mykey → "Hello“
redis>  GET mykey → (nil)
```

MSET sets multiple key values.

```
redis>  MSET key1 "Hello" key2 "World“→"OK“
redis>  GET key1→"Hello"
redis>  GET key2→"World"
```

Redis Partitioning

Range partitioning vs Hash partitioning

- Redis instances R0, R1, R2, R3, and keys representing users like user:1, user:2, ...
  - RP: Map ranges of objects into specific Redis instances. Users from ID 0 to ID 10000 will go into instance R0, while users form ID 10001 to ID 20000 will go into instance R1 and so forth.
    - It has the disadvantage of requiring a table that maps ranges to instances. This table needs to be managed and a table is needed for every kind of object, so therefore range partitioning in Redis is often undesirable because it is much more inefficient than other alternative partitioning approaches.
  - HP: An alternative to range partitioning is hash partitioning. This scheme works with any key, without requiring a key in the form object_name:<id>
    - Take the key name and use a hash function (e.g., the crc32 hash function) to turn it into a number. For example, if the key is foobar, crc32(foobar) will output something like 93024922.
    - Use a modulo operation with this number in order to turn it into a number between 0 and 3, so that this number can be mapped to one of my four Redis instances. 93024922 modulo 4 equals 2, so I know my key foobar should be stored into the R2 instance.

Amazon DynamoDB

Amazon DynamoDB
Fast, flexible NoSQL database service

- Major service of AWS for data storage
  - E.g. product lists, shopping carts, user preferences
- Data model (key, structured value)
  - Partitioning on the key and secondary indices on attributes
  - Simple queries on key and attributes
  - Flexible: no schema to be defined (but automatically inferred)
- Consistency
  - Eventual consistent reads (default)
  - Atomic updates with atomic counters
- High availability and fault-tolerance
  - Synchronous replication between data centers
- Integration with other AWS services
  - Identity control and access
  - MapReduce
  - Redshift data warehouse

DynamoDB – data model

Amazon DynamoDB
Fast, flexible NoSQL database service

- Table (items)
- Item (key, attributes)
  - 2 types of primary (unique) keys
    - Hash (1 attribute)
    - Hash & range (2 attributes)
  - Attributes of the form "name":"value"
    - Type of value: scalar, set, or JSON
- API with methods
  - *Add, update, delete* item
  - *GetItem*: returns an item by primary key in a table
  - *BatchGetItem*: returns the items of same primary key in multiple tables
  - *Scan* : returns all items
  - *Query*
    - Range on hash & range key
    - Access on indexed attribute

Table: Forum_Thread

Forum	Subject	Date of last post	Tags
“S3”	“abc”	“2017 . . . ”	“a” “b”
“S3”	“acd”	“2017 . . . ”	“c”
“S3”	“cbd”	“2017 . . . ”	“d” “e”
“RDS”	“xyz”	“2017 . . . ”	“f”
“EC2”	“abc”	“2017 . . . ”	“a” “e”
“EC2”	“xyz”	“2017 . . . ”	“f”

Hash key — Range key

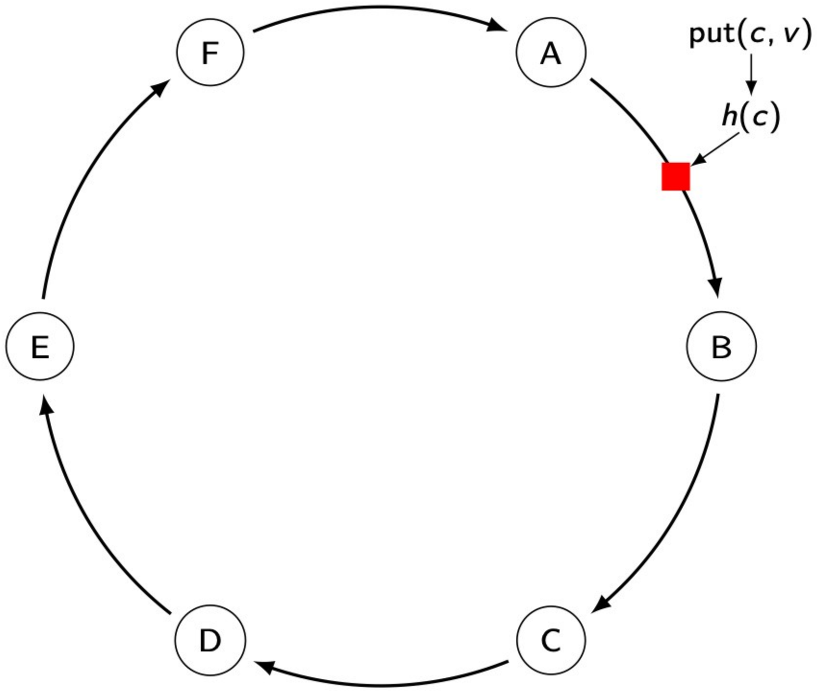
GetItem (Forum="EC2", Subject="xyz")

Query (Forum="S3", Subject > "ac")

DynamoDB - data partitioning

Amazon DynamoDB
Fast, flexible NoSQL database service

- **Consistent hashing**: the interval of hash values is treated as a ring
- **Advantage**: if a node fails, its successor takes over its data
  - No impact on other nodes
- Data is **replicated on next nodes**



Node B is responsible for the hash value interval (A,B]. Thus, item (c,v) is assigned to node B

DOCUMENT STORES

Data Model

- Documents
  - **Hierarchical structure,** with **nesting** of elements
  - Weak **structuring**, with "similar" elements
  - **Scalar** types (text, integer, real, date) but also **maps**, **lists**, **sets**, **nested** documents, …
  - Identified by a **unique identifier** (key, …)
  - Documents are organized into **collections**
- Two main data models
  - XML (eXtensible Markup Language): W3C standard (1998) for exchanging data on the Web
    - Complex and heavy
  - JSON (JavaScript Object Notation) by Douglas Crockford (2005) for exchanging data JavaScript
    - Simple and light

Queries in Document Stores

- Query patterns
  - Create, update or remove a document
  - Retrieve documents according to complex query conditions
- Consider as…
  - Extended key-value store where the value part is a document that you can query.

Document Stores

- **Suitable use cases**
  - Event logging, content management systems, blogs, web analytics, e-commerce applications, Analysis of messages (tweets, etc.) in real time
    - I.e. **for structured documents with similar schema**
- When not to use
  - **Set operations** involving multiple documents
  - Design of document structure is constantly changing
    - I.e. when the required level of granularity would outbalance the advantages of aggregates

Document Stores

mongoDB®

Couchbase

RethinkDB

RAVENDB
Safe by Default, Optimized for Efficiency

CouchDB
relax

terrastore

MarkLogic™

OrientDB®

MongoDB

- Objective: performance and scalability
  - A document is a collection of (key, typed value) with a unique key (generated by MongoDB)
- Data model and query language based on JSON
  - Binary JSON (BSON): more compact
- No schema, no join, no complex transaction
- Shared-nothing cluster architecture
- Secondary indices
- Integration with MapReduce & Spark

A MongoDB Collection (posts)

_id: ObjectId("abc")	author: "alex", title: "No Free Lunch", text: "This is ...", tags: ["business", "ramblings"], comments: [{who: "jane", what: "I agree."},{who: "joe", what: "No ..."}]
_id: ObjectId("abd")	A post by X
_id: ObjectId("acd")	A post by Y

Unique key generated by MongoDB

Value = JSON object with nested arrays

MongoDB – query language

- Expression of the form
  - db.documentType.function (JSON expression)
- Update examples
  - db.posts.insert({author:’alex’, title:’No Free Lunch’})
  - db.posts.update({author:’alex’, {$set:{age:30}})
  - db.posts.update({author:’alex’, {$push:{tags:’music’}})
- Select examples
  - db.posts.find({author:"alex"})
    - All posts from Alex
  - db.posts.find({comments.who:"jane"})
    - All posts commented by Jane

COLUMN STORES

Wide Column Stores

- Data model

  Column family (table)

    - Table is a collection of **similar rows** (not necessarily identical)

  Row

    - Row is a collection of **columns**
      - – Should encompass a group of data that is accessed together
    - Associated with a unique **row key**

  Column

    - Column consists of a **column name** and **column value**
      - (and possibly other metadata records)
    - Scalar values, but also **flat sets, lists or maps** may be allowed

Wide Column Stores

- Query patterns

  Create, update or remove a row within a given column family

  - Select rows according to a row key or simple conditions
  - Reconstruct a record from columns

Wide Column Stores

- Suitable use cases
  - Event logging, content management systems, blogs, …
    - I.e. for structured flat data with similar schema
  - Queries that involve only a few columns
  - Analytical queries: aggregation (SUM, AVG, …), it allows for fast retrieval of columns of data.
  - Column-wise compression
- No suitable for
  - OLTP applications that insert records of data
    - Need to split records in columns
  - Queries against only a few rows – e.g. point queries

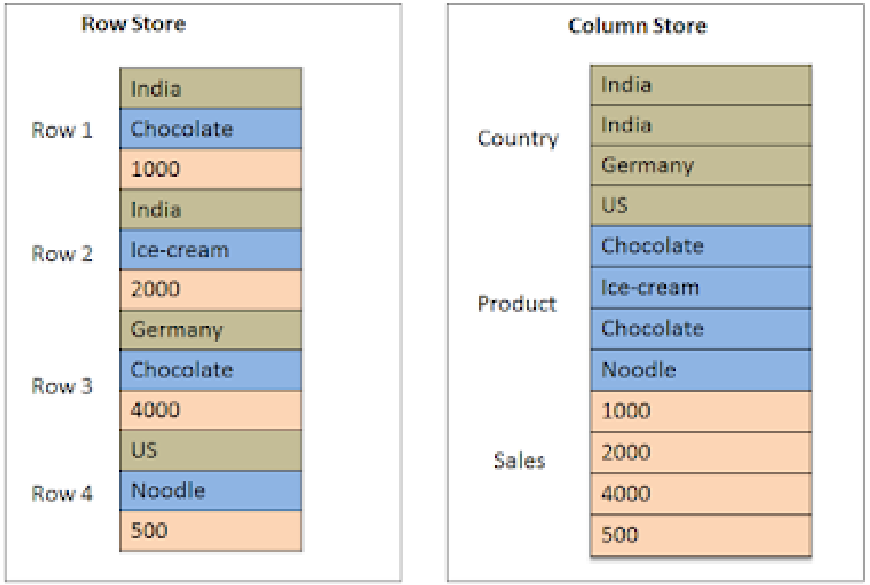
Row-Oriented vs Column Oriented

Rows stored sequentially

Table

	Country	Product	Sales
Row 1	India	Chocolate	1000
Row 2	India	Ice-cream	2000
Row 3	Germany	Chocolate	4000
Row 4	US	Noodle	500

```
Row1:India,Chocolate,1000;
Row2:India,Ice-cream,2000;
Row3:Germany,Chocolate,4000;
Row4:US,Noodle,500;
```

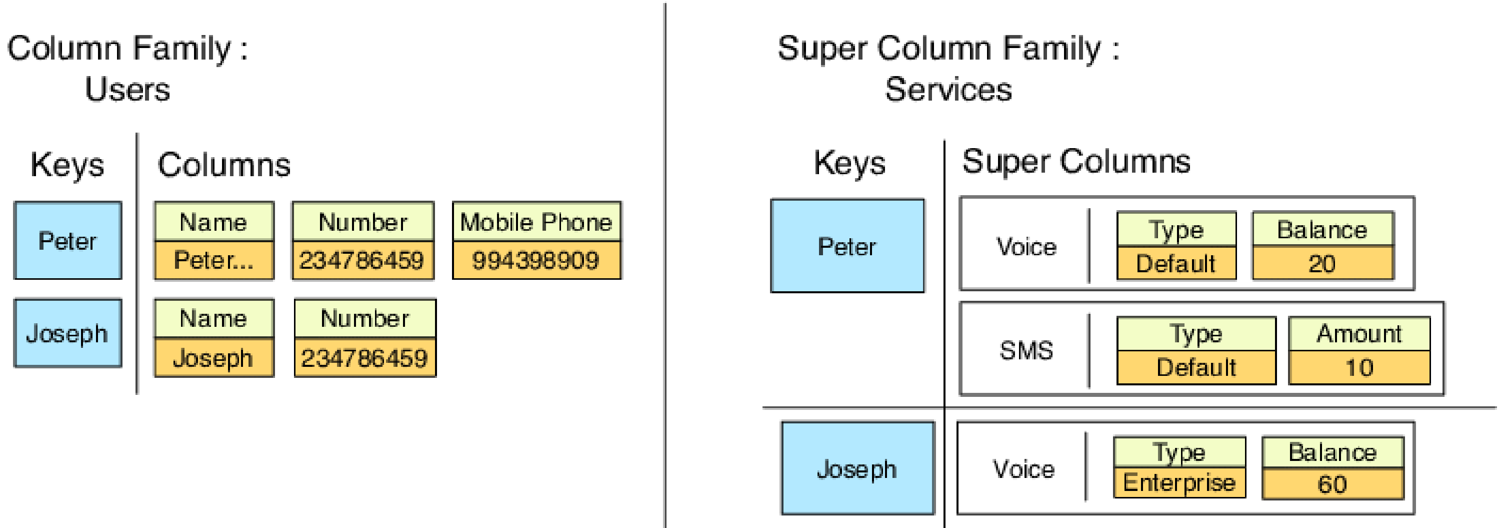



```
India:Row1,Row2; Germany: Row3; US: Row4;
Chocolate:Row1, Row3; Ice-Cream:Row2; Noodle: Row4;
1000:Row1; 2000:Row2; 4000:Row3; 500: Row4;
```

Column Values Stored sequentially, mapped to a RowID

Column Families

- A column family contains columns of related data.
  - a key–value pair, where the key is mapped to a value that **is a set** of columns.
  - In analogy with relational databases, a standard column family is as a "table", each key–value pair being a "row".
- A Super Column Family Contains Column Families

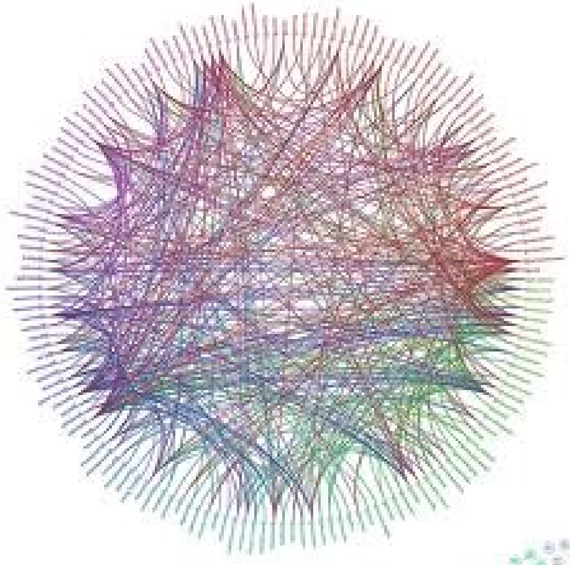


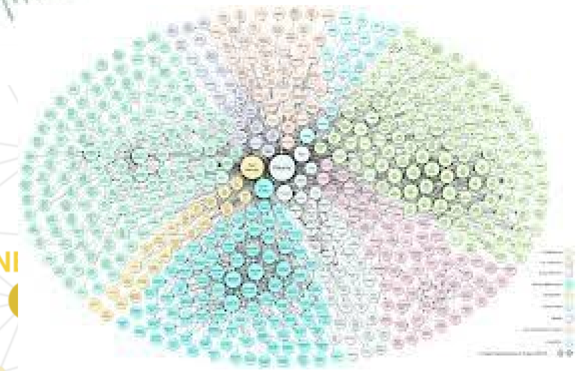
Wide Column Stores

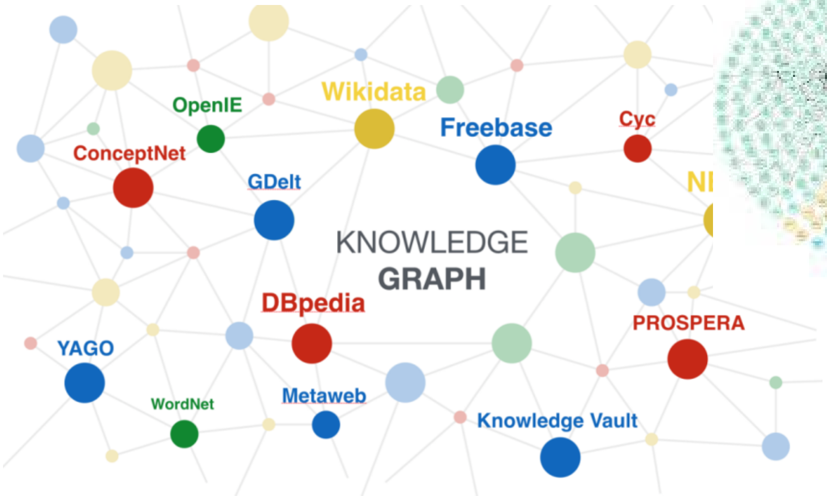
GRAPH STORES

Graphs

- Data graphs
  - Very big: billions of nodes and links
  - Many: millions of graphs
- Main applications
  - Social networks
    - Recommendation, sharing, sentiment analysis
  - Knowledge Graphs
    - Wikipedia, Google Knowledge Graph
  - Scientific networks
    - Biological networks
  - Web of Data
    - Linked Data







Graph Databases

- Data model

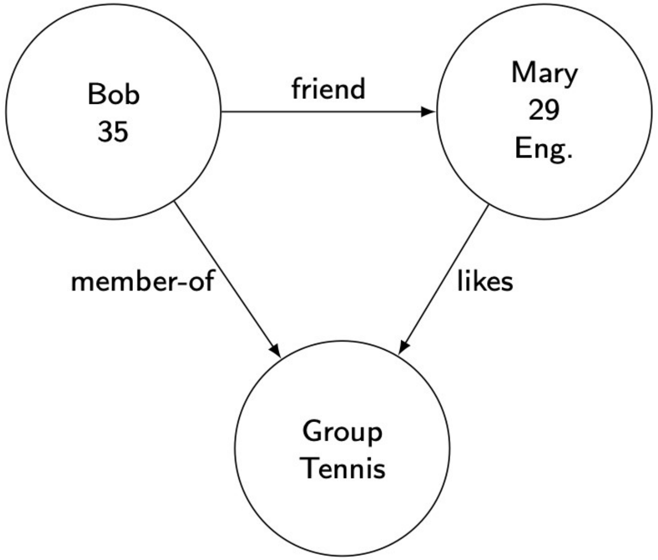
  Property graphs

  - **Directed / undirected graphs**, i.e. collections of …

    nodes (vertices) for real-world entities, and

    relationships (edges) between these nodes

  - Both the nodes and relationships can be associated with additional properties



- Types of databases
  - **Non-transactional** = small number of very large graphs
  - **Transactional** = large number of small graphs

Graph Databases

- Query patterns
  - Create, update or remove a node / relationship in a graph
    - *Add Mary as Friend to Peter, Get the address of Mary*
  - General **graph traversals**
    - *Get the Friend of Mary who is married to Anna*
  - **Sub-graph** queries
    - *Get All Friends of Mary who work in the same company with her*
  - **Similarity** based queries (approximate matching)
    - *Get the Friends of Mary whose names start from 'P'*
  - **Graph algorithms** (shortest paths, spanning trees, …)

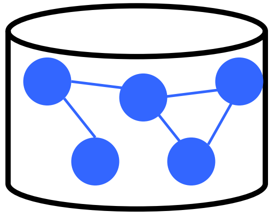
Graph Databases

- **Suitable use cases**
  - Social networks, routing, dispatch, and location-based services, recommendation engines, biological pathways, linguistic trees, …
    - I.e. simply **for graph structures**
- When not to use
  - **Extensive batch operations** are required
    - Multiple nodes / relationships are to be affected
  - **Only too large graphs** to be stored
    - Graph distribution is difficult or impossible at all

Graph Databases

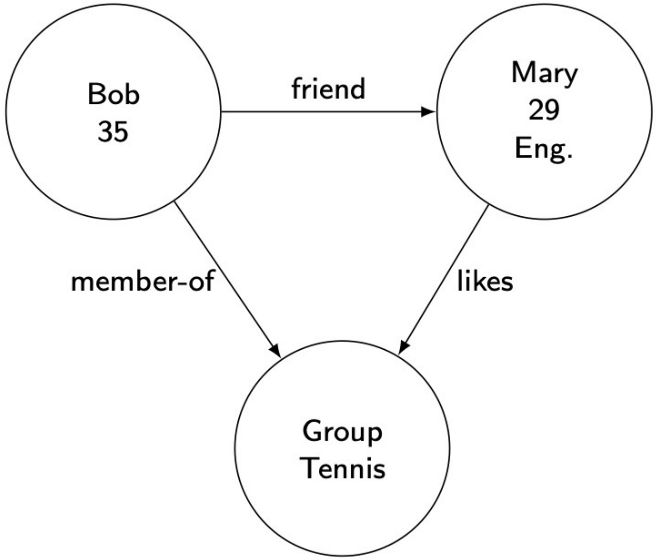
Neo4J

- Direct support of graphs
  - Data model, API, query language
  - Implemented by linked lists on disk
  - Optimized for graph processing
  - Transactions
- Implemented on SN cluster
  - Asymmetric replication
  - Graph partitioning
    - Data “Fabrics”



Neo4J – data model

- Nodes
- Links between nodes
- Properties on nodes and links



A Neo4j transaction

```
NeoService neo = … // factory
Transaction tx = neo.beginTx();
Node n1 = neo.CreateNode();
n1.setProperty("name", "Bob");
n1.setProperty("age", 35);
Node n2 = neo.createNode();
n2.setProperty("name", "Mary");
n2.setProperty("age", 29);
n2.setProperty("job", "engineer");
n1.createRelationshipTo(n2, RelTypes.friend);
tx.Commit();
Node n3 = ...
```

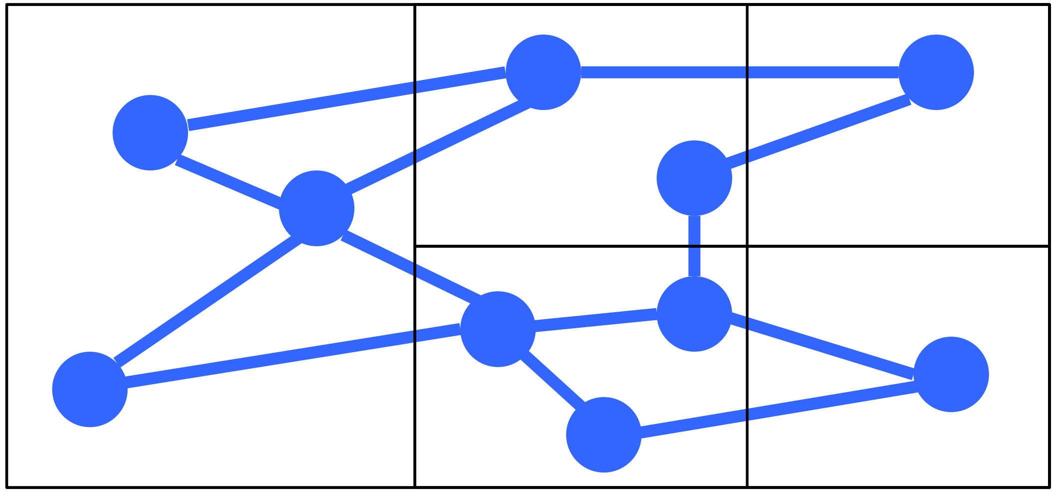
Neo4J - Cypher Query Language

- Java API (navigational)
- Cypher query language. It is a declarative, SQL-inspired language for describing visual patterns in graphs
  - Queries and updates with graph traversals
- Example Cypher query that returns the (indirect) friends of Bob whose name starts with "M"

```
MATCH (bob:Person {name = 'Bob'})-[:friend]-> follower:Person
      WHERE follower.name =~ 'M*' (or follower.name STARTS WITH 'M')
      RETURN follower.name
```

- Support of SPARQL for RDF data

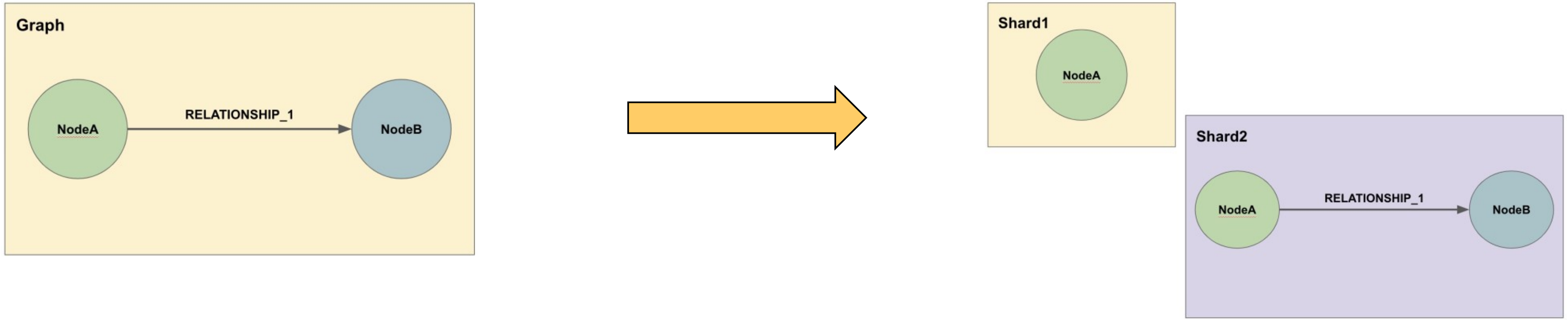
Graph Partitioning



- Objective: get balanced partitions
  - NP-hard problem: no optimal algorithm
  - Solutions: approximate, heuristics, based on the graph topology

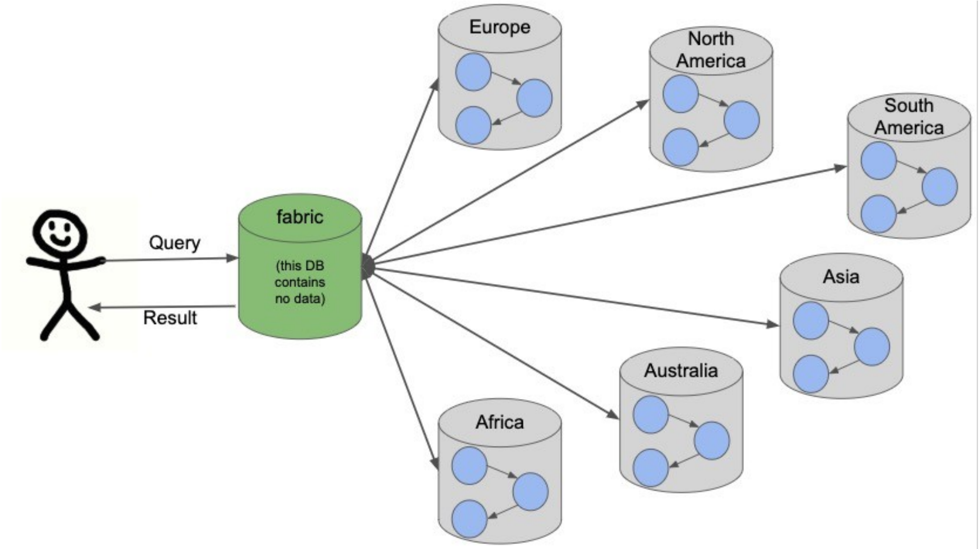
Graph Sharding

- Allows users to break a larger graph down into individual, smaller graphs and store them in separate databases.
- For graphs that are highly-connected, this means some level of data redundancy to maintain the relationships between entities.



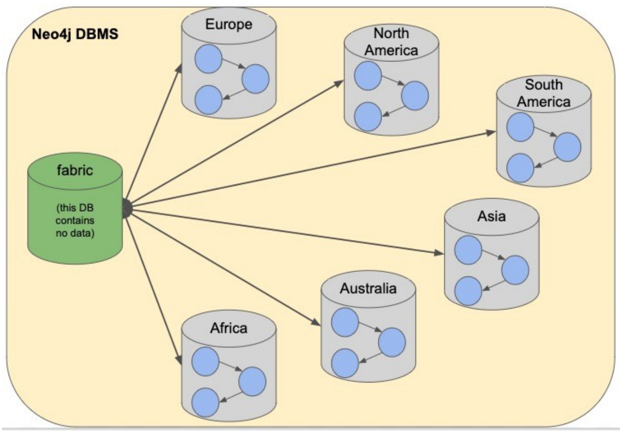
Fabric Database

- A virtual database that does not store data, but acts as the entry point into the rest of the graphs
- Queries coming from users or applications will hit the fabric database first, then get routed to the instance or instances required to answer the query.

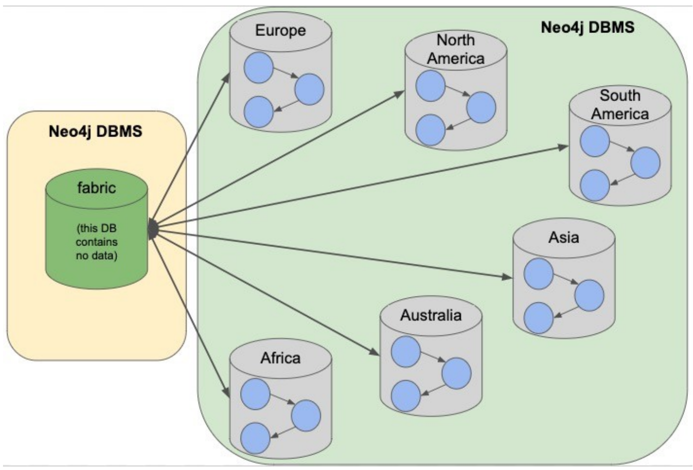


Neo4J – Alternative architectures

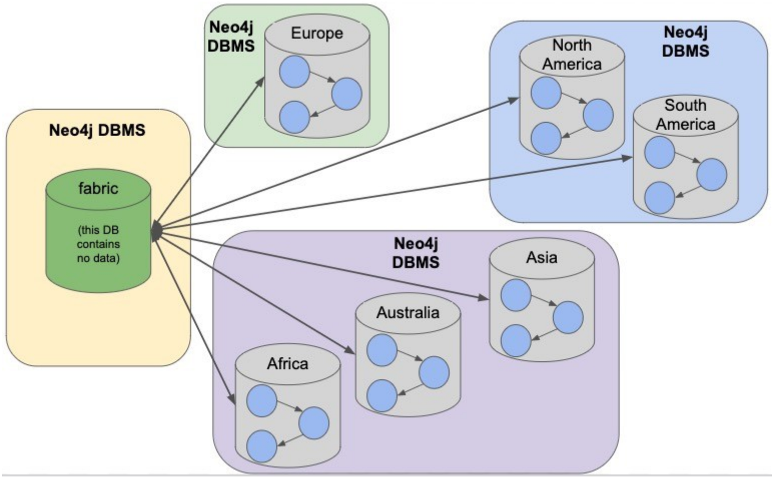
A single DBMS for everything



Fabric database in separate DBMS



Scale out in multiple DBMS



ARE THERE MORE?

Native XML Databases

- Data model
  - **XML documents**
    - Tree structure with nested elements, attributes, and text values (beside other less important constructs)
    - Documents are organized into collections
- Query languages
  - **XPath**: *XML Path Language* (navigation)
  - **XQuery**: *XML Query Language* (querying)

Native XML Databases

MarkLogic™

OPENLINK® VIRTUOSO UNIVERSAL SERVER

sedna
Native XML Database System

Tamino XML Server

existdb

RDF Stores

- Data model
  - **Resource Description Framework (RDF) triples**
    - Components: **subject**, **predicate**, and **object**
    - Each triple represents a statement about a real-world entity
  - Triples can be viewed as **graphs**
    - **Vertices** for subjects and objects
    - **Edges** directly correspond to individual statements
- Query language
  - **SPARQL**: *SPARQL Protocol and RDF Query Language*

RDF Stores

More details in Coming Lecture

Multi – model or Polystores

- Support multiple data models against a single, integrated backend.
  - E.g., Document + relational
  - Document + graph + key–value
  - Document + relational + key–value
  - …
- Multi-model support is either within the DB engine (native) or via different layers on top of the engine (layered architecture).
  - E.g., user relational table to store graphs

What about NewSQL DBs?

- **NewSQL** is a class of **relational database management** systems for online transaction processing (**OLTP**) workloads.
  - online **scalability** of NoSQL databases
  - Support of **SQL**
  - Maintaining the **ACID** guarantees
- Distributed architectures & distributed query processing.
- Optimized SQL engines with advanced statistics
- Transparent sharding

BLOG@CACM

New SQL: An Alternative to NoSQL and Old SQL For New OLTP Apps

By Michael Stonebraker
June 16, 2011
Comments (11)

VIEW AS: SHARE:

Historically, Online Transaction Processing (OLTP) was performed by customers submitting traditional transactions (order something, withdraw money, cash a check, etc.) to a relational DBMS. Large enterprises might have dozens to hundreds of these systems. Invariably, enterprises wanted to consolidate the information in these OLTP systems for business analysis, cross selling, or some other purpose. Hence, Extract-Transform-and-Load (ETL) products were used to convert OLTP data to a common format and load it into a data warehouse. Data warehouse activity rarely shared machine resources with OLTP because of lock contention in the DBMS and because business intelligence (BI) queries were so resource-heavy that they got in the way of timely responses to transactions.

VOLTDB

NUODB

CockroachDB

TIBCO ActiveSpaces®

Amazon Aurora

Google Cloud

PingCAP

TiDB

Cloud Spanner

NewSQL

- Pros NoSQL
  - Scalability
    - Often by relaxing strong consistency
  - Performance
  - Practical APIs for programming
- Pros Relational
  - Strong consistency
  - Transactions
  - Standard SQL
    - Makes it easy for tool vendors (BI, analytics, …)
- NewSQL = NoSQL/relational hybrid

SUMMARY OF FEATURES OF NOSQL DATABASES

Features of NoSQL Databases

- **Data model**
  - Traditional approach: relational model
  - (New) possibilities:
    - **Key-value, document, wide column, graph**
    - Object, XML, RDF, …
  - Goal
    - Respect the real-world nature of data
      - (i.e. data structure and mutual relationships)

Features of NoSQL Databases

- **Aggregate structure**
  - Aggregate definition
    - Data unit with a complex structure
    - **Collection of related data pieces we wish to treat as a unit**
      - (with respect to data manipulation and data consistency)
  - Examples
    - **Value** part of key-value pairs in key-value stores
    - **Document** in document stores
    - **Row** of a **column family** in wide column stores

Features of NoSQL Databases

- **Aggregate structure**
  - Types of systems
    - **Aggregate-ignorant**: relational, graph
      - It is not a bad thing, it is a feature
    - **Aggregate-oriented**: key-value, document, wide column
  - Design notes
    - No universal strategy how to draw **aggregate boundaries**
      Atomicity of database operations:
      - just a single aggregate at a time

Features of NoSQL Databases

- **Elastic scaling**
  - Traditional approach: scaling-up
    - Buying bigger servers as database load increases
  - New approach: scaling-out
    - Distributing database data across multiple hosts
      - – Graph databases (unfortunately): difficult or impossible at all
- **Data distribution**
  - Sharding
    - Particular ways how database data is split into separate groups
  - Replication
    - Maintaining several data copies (performance, recovery)

Features of NoSQL Databases

- **Automated processes**
  - Traditional approach
    - Expensive and highly trained database administrators
- New approach: **automatic recovery, distribution, tuning, … Relaxed consistency**
  - Traditional approach
    - **Strong consistency** (ACID properties and transactions)
  - New approach
    - **Eventual consistency** only (BASE properties)
      - I.e. we have to make trade-offs because of the data distribution

Features of NoSQL Databases

- **Schemalese-ness**
  - Relational databases
    - Database schema present and **strictly enforced**
  - NoSQL databases
    - **Heterogeneous, Relaxed schema** or **completely missing**
    - Consequences: **higher flexibility**
      - Dealing with **non-uniform data**
      - **Structural changes** cause no overhead
    - However: there is (usually) an **implicit schema**
      - We must know the data structure at the **application level** anyway

Features of NoSQL Databases

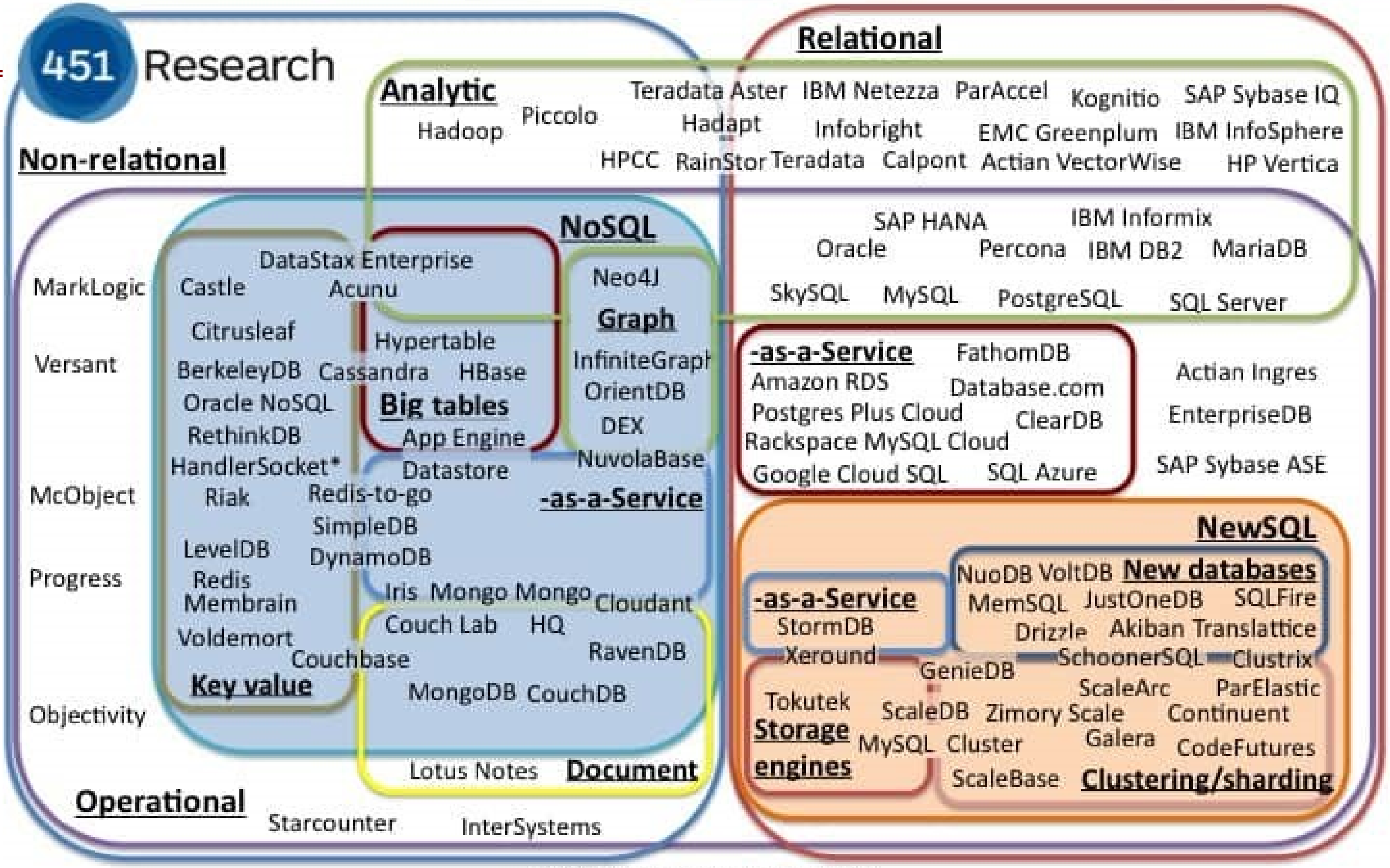
- **Open source**
  - Community and enterprise versions (with extended features or extent of support)
- **Simple APIs**
  - State-less application interfaces (HTTP)

Which Data Store for What?

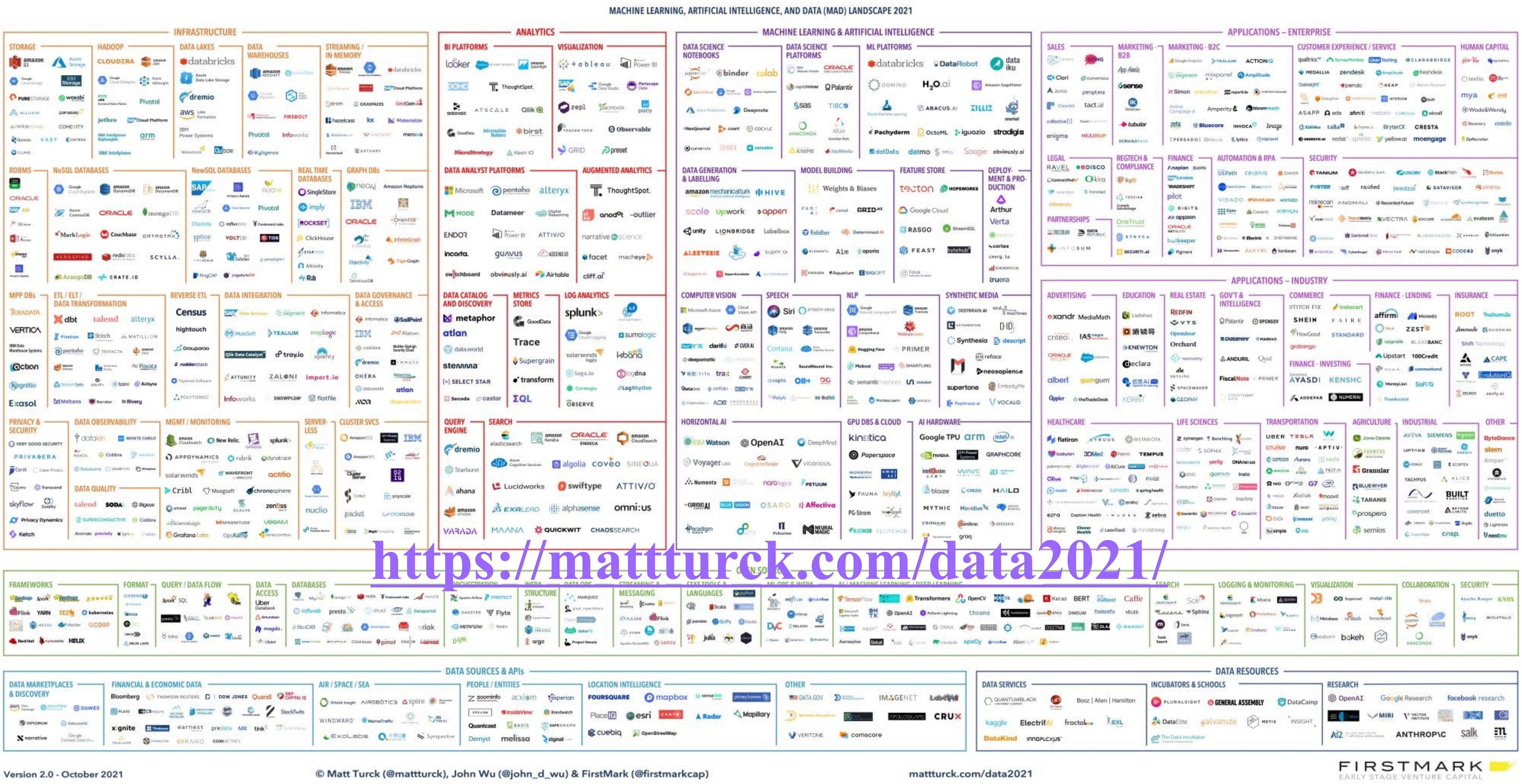
Category	Requirements
Key-value	• Access by key • Flexibility (no schema) • Very high scalability and performance
Document	• Web content management • Flexibility (no schema) • Limited transactions
Column	• Very big collections • Analytical tasks • Scalability and high availability
Graph	• Efficient storage and management of large graphs
Multimodel	• Integrated key-value, document and graph management
NewSQL	• ACID transactions , flexibility and scalability • SQL and key-value access

WHAT IS NEXT?

The evolving database landscape



For a ranked list of DB engines: https://db-engines.com/en/ranking/

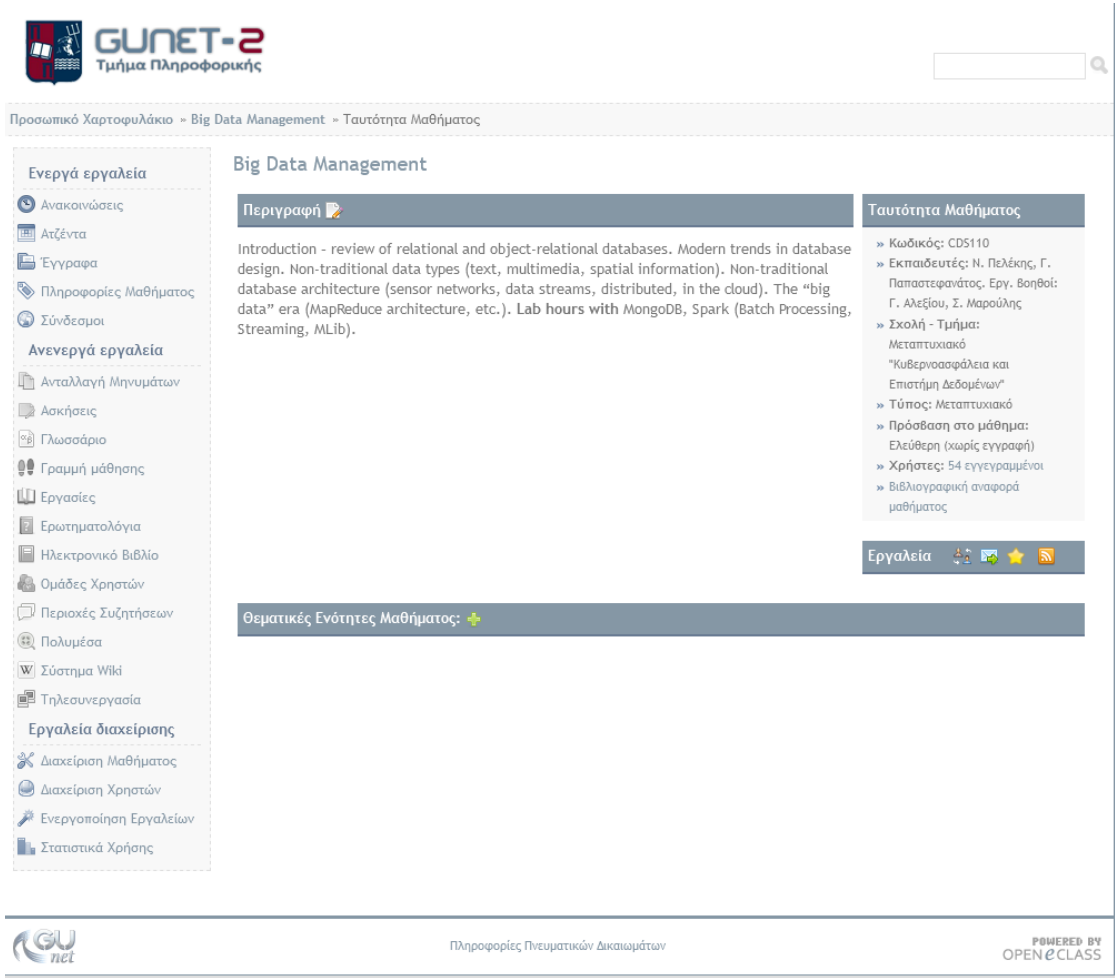


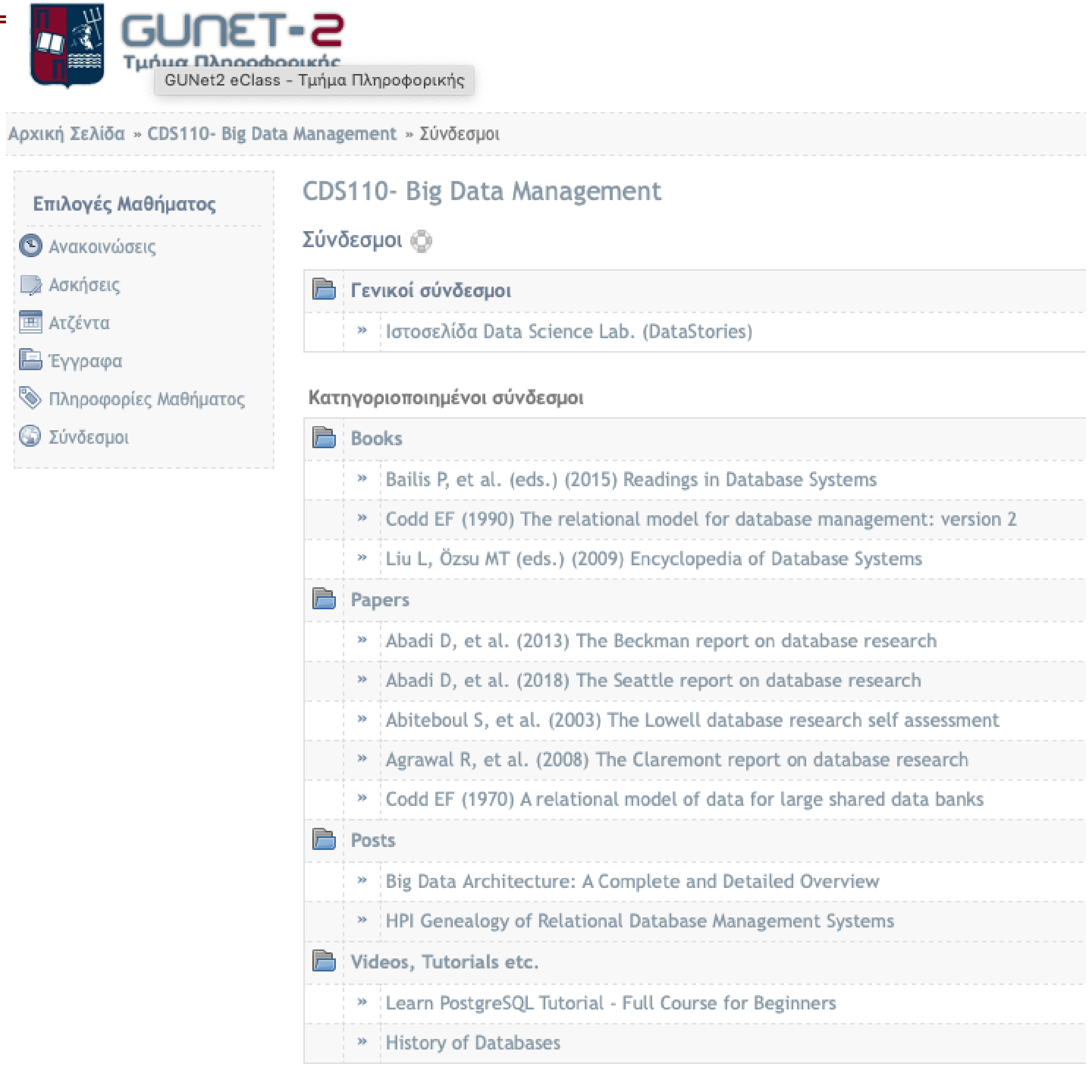
https://mattturck.com/data2021/

Things to study

- CAP Theorem: https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
- M. Stonebraker and U. Cetintemel, ""One size fits all": an idea whose time has come and gone," 21st International Conference on Data Engineering (ICDE'05), 2005, pp. 2-11, doi: 10.1109/ICDE.2005.1.
- New SQL: An Alternative to NoSQL and Old SQL For New OLTP Apps By Michael Stonebraker (June 16, 2011) https://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext
- For a ranked list of all DB engines: https://db-engines.com/en/ranking/
- Principles of Distributed Database Systems. M. Tamer Özsu, Patrick Valduriez, https://cs.uwaterloo.ca/~ddbook/
  - Chapter11: NoSQL, NewSQL and Polystores

Thank you


GUNET-2
Τμήμα Πληροφορικής
Προσωπικό Χαρτοφυλάκιο » Big Data Management » Ταυτότητα Μαθήματος
Ενεργά εργαλεία
Ανακοινώσεις
Ατζέντα
Έγγραφα
Πληροφορίες Μαθήματος
Σύνδεσμοι
Ανενεργά εργαλεία
Ανταλλαγή Μηνυμάτων
Ασκήσεις
Γλωσσάριο
Γραμμή μάθησης
Εργασίες
Ερωτηματολόγια
Ηλεκτρονικό Βιβλίο
Ομάδες Χρηστών
Περιοχές Συζητήσεων
Πολυμέσα
Σύστημα Wiki
Τηλεσυνεργασία
Εργαλεία διαχείρισης
Διαχείριση Μαθήματος
Διαχείριση Χρηστών
Ενεργοποίηση Εργαλείων
Στατιστικά Χρήσης
Big Data Management
Περιγραφή
Introduction - review of relational and object-relational databases. Modern trends in database design. Non-traditional data types (text, multimedia, spatial information). Non-traditional database architecture (sensor networks, data streams, distributed, in the cloud). The “big data” era (MapReduce architecture, etc.). Lab hours with MongoDB, Spark (Batch Processing, Streaming, MLib).
Ταυτότητα Μαθήματος
» Κωδικός: CDS110
» Εκπαιδευτές: Ν. Πελέκης, Γ. Παπαστεφανάτος. Εργ. Βοηθοί: Γ. Αλεξίου, Σ. Μαρούλης
» Σχολή - Τμήμα: Μεταπτυχιακό "Κυβερνοασφάλεια και Επιστήμη Δεδομένων"
» Τύπος: Μεταπτυχιακό
» Πρόσβαση στο μάθημα: Ελεύθερη (χωρίς εγγραφή)
» Χρήστες: 54 εγγεγραμμένοι
» Βιβλιογραφική αναφορά μαθήματος
Εργαλεία
Θεματικές Ενότητες Μαθήματος:
Πληροφορίες Πνευματικών Δικαιωμάτων
POWERED BY OPEN eCLASS


GUNET-2
GUNet2 eClass - Τμήμα Πληροφορικής
Αρχική Σελίδα » CDS110- Big Data Management » Σύνδεσμοι
Επιλογές Μαθήματος
Ανακοινώσεις
Ασκήσεις
Ατζέντα
Έγγραφα
Πληροφορίες Μαθήματος
Σύνδεσμοι
CDS110- Big Data Management
Σύνδεσμοι
Γενικοί σύνδεσμοι
» Ιστοσελίδα Data Science Lab. (DataStories)
Κατηγοριοποιημένοι σύνδεσμοι
Books
» Bailis P, et al. (eds.) (2015) Readings in Database Systems
» Codd EF (1990) The relational model for database management: version 2
» Liu L, Özsu MT (eds.) (2009) Encyclopedia of Database Systems
Papers
» Abadi D, et al. (2013) The Beckman report on database research
» Abadi D, et al. (2018) The Seattle report on database research
» Abiteboul S, et al. (2003) The Lowell database research self assessment
» Agrawal R, et al. (2008) The Claremont report on database research
» Codd EF (1970) A relational model of data for large shared data banks
Posts
» Big Data Architecture: A Complete and Detailed Overview
» HPI Genealogy of Relational Database Management Systems
Videos, Tutorials etc.
» Learn PostgreSQL Tutorial - Full Course for Beginners
» History of Databases