Data for the Web

INSA Lyon

IT&CS department

MSIF

Part 4: NOSQL

Előd EGYED-ZSIGMOND

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Plan

- Introduction
- Core XML
- XML galaxy
- NOSQL
    - Introduction
    - Basic concepts
    - Column family
    - Key-Value Store
    - Graph DBMS
    - Document Store
- Conclusion

SQL Means More than SQL

- SQL stands for the query language
- But commonly refers to the traditional RDBMS:
  - Relational storage of data
    - Each tuple is stored consecutively
  - Joins as first-class citizens
    - In fact, normal forms prefer joins to maintenance
  - Strong guarantees on transaction management
    - No consistency worries when many transactions operate simultaneously on common data
- Focus on *scaling up*
  - That is, make a single machine do more, faster

Trends Drive Common Requirements

Social media + mobile computing + IoT

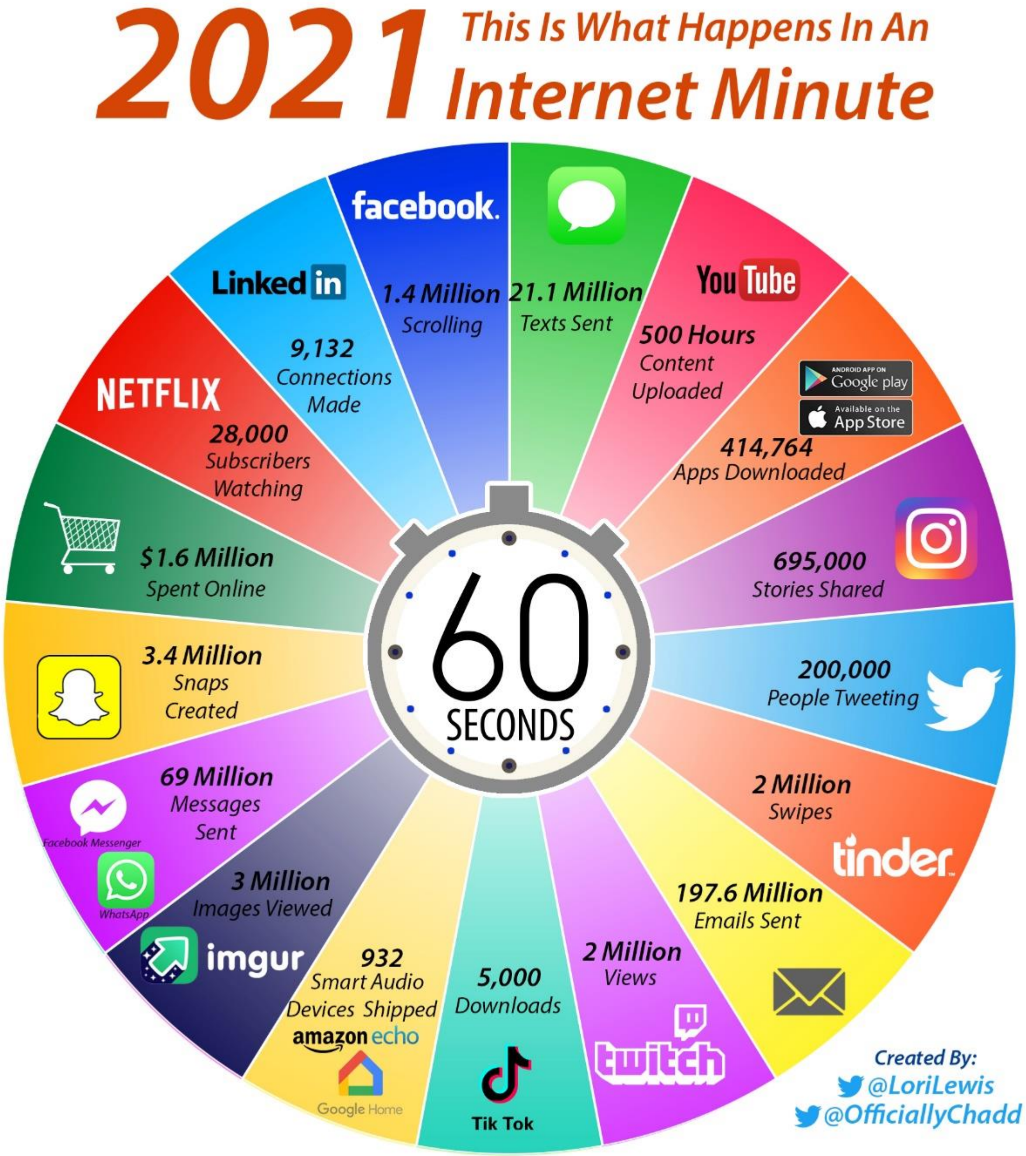
- Explosion in data, always available, constantly read and updated
- High load of simple requests of a common nature
- Some consistency can be compromised

Cloud computing + open source

- Affordable resources for management / analysis of data
- People of various skills / budgets need software solutions for distributed analysis of massive data

Database solutions need to *scale out* (use distribution, “scale horizontally”)

Scale out vs scale up



Emerging of Big Data (from wikipedia)

Science

- Large Hadron Collider -> 25 PB in 2012 200 PB after-replication

Government

- Utah Data Center being white constructed by NSA -> (Maybe) A Few exabytes

Business

- eBay -> 40bp Hadoop cluster for search and recommendation
- Walmart:> 1 million TranX per hour, DB> 2.5 petabytes
- Facebook -> 50 trillion pictures (in Haystack); Messaging 25 TB / month a while ago (inHBase)

The 4 V Big Data

- **V**olume
- **V**ariety and heterogeneity
- **V**elocity
- **V**eracity
- (**V**alue)

- ☞ **Access rights**

Compromises Required

What is needed for effective distributed, data- and user-intensive applications?

1. Use data models and storage that allow to avoid joins of big objects
2. Relax the guarantees on consistency

Plan

- Introduction
- Core XML
- XML galaxy

NOSQL

Introduction

Basic concepts

Column family

Key-Value Store

Graph DBMS

Document Store

NOSQL

Distributed DBs

Cloud

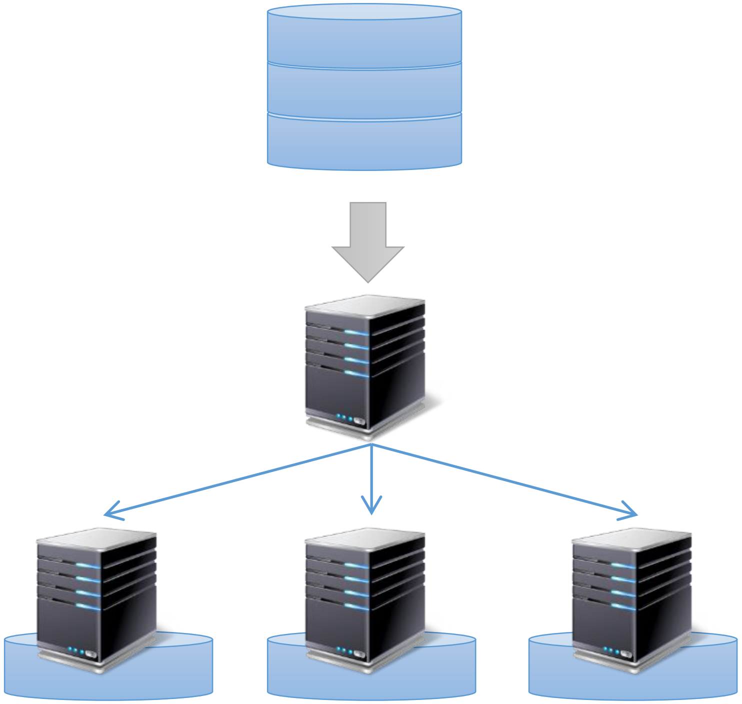
- Conclusion

Database Replication

- Data replication: storing the same data on several machines ("nodes")
- Useful for:
  - Availability (parallel requests are made against replicas)
  - Reliability (data can survive hardware faults)
  - Fault tolerance (system stays alive when nodes/network fail)
- Typical architecture: master-slave

Database Sharding

- Simply partitioning data across multiple nodes
- Useful for
  - Scaling (more data)
  - Availability



Horizontal / vertical partitioning

- "**Horizontal partitioning**", or sharding, is replicating [copying] the schema, and then dividing the data based on a shard key.

- "**Vertical partitioning**" involves dividing up the schema (and the data goes along for the ride).

NoSQL - Replication

A - H I - P Q - Z

Partitioning

AZ AZ AZ

Sharding

NoSQL - Replication

A - H
+
I - P

I - P
+
Q - Z

Q - Z
+
A - H

Partitioning + Sharding + Replication

Distributed File System

Do not move the data to the process ... move the process to the data!

- Store data on local disks of the cluster nodes
- Start the process on the node that owns the local data

Why?

- Bandwidth limited network
- Not enough RAM to hold all the data in the memory
- The disk access is slow, but the speed of the disc is reasonable

Creating a distributed file system

- GFS (Google File System) for MapReduce Google
- HDFS (Hadoop of Distributed File System) Hadoop

Map Reduce

Bring the process to the data, not the data to the process in a distributed environment

Huge amount of information

Distributed architecture

Parallelize calculations

Introduced long time ago (LISP, 1958)

Map

- map(m_f, [a_1, a_2, …a_n])
- -> [b_1, b_2, …, b_n]

- Accepts two arguments: a function and a list of values.
- Generates output by repeatedly applying the function on the list of values.

Map’s output is a list of values, which reduce can accept as one of its argument.

Reduce

- reduce(r_f, [b_1, b_2, …b_n])
- -> c

- Accepts two arguments: a function and a list of values.
- Generates output by reducing the list of input values using the function.

Map reduce : Analogy

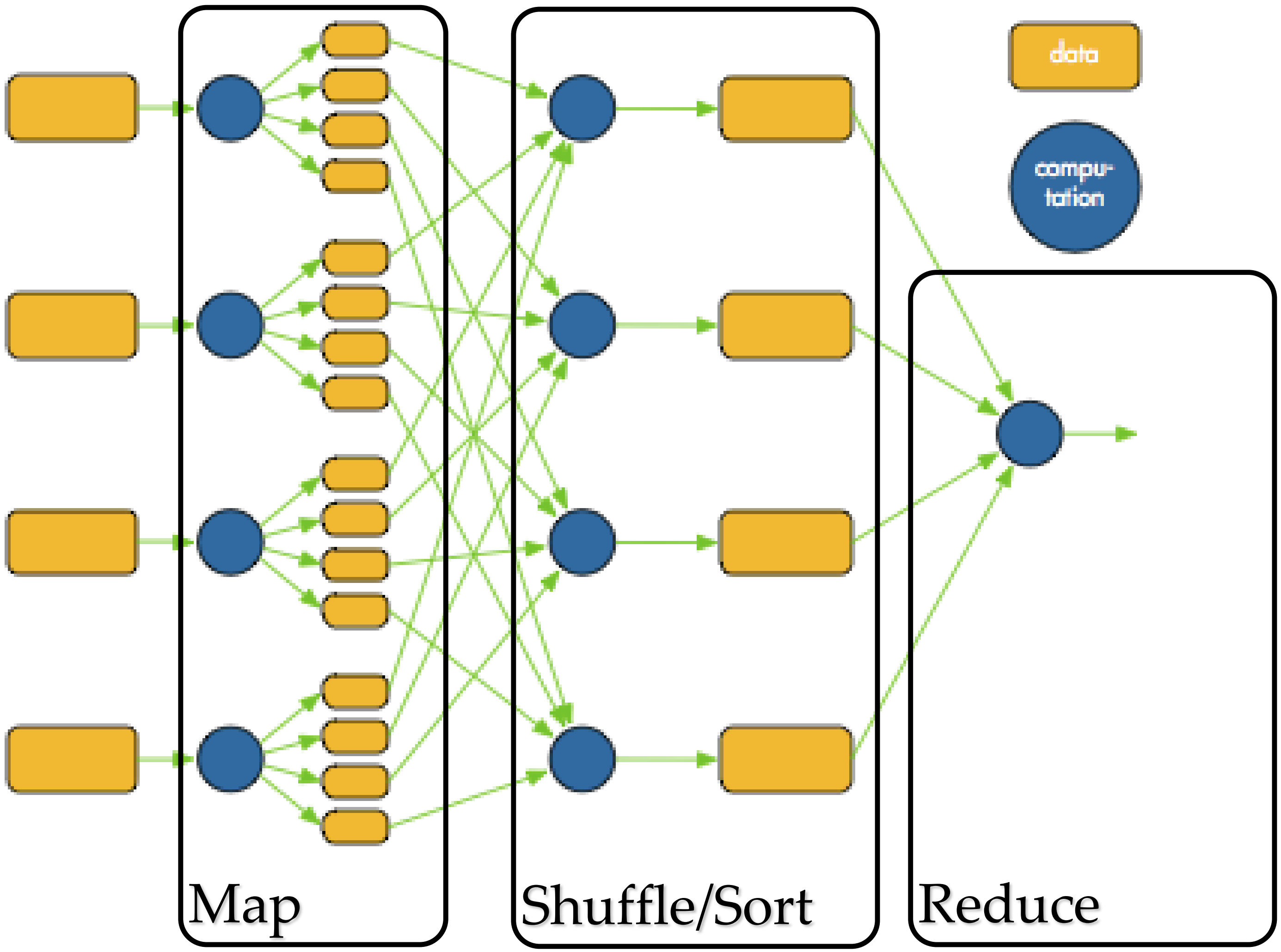
- Break large problem into small pieces
- Code m_f to solve one piece
- Run map to apply m_f on the small pieces and generate nuggets of solutions
- Code r_f to combine the nuggets
- Run reduce to apply r_f on the nuggets to output the complete solution

Map Sort Reduce

Mapreduce has three main phases

- Map (send each input record to a key)
- Sort (put all of one key in the same place)
    - handled behind the scenes
- Reduce (operate on each key and its set of values)
- Terms come from functional programming

Mapreduce overview



These 5 slides from : http://www.cs.cmu.edu/~wcohen/

Mapreduce: slow motion

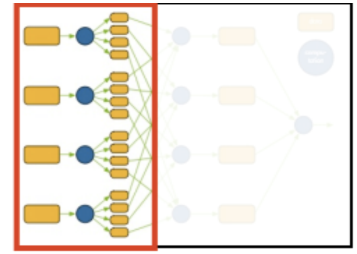
- The canonical mapreduce example is word count
- Example corpus:

  Joe likes toast

  Jane likes toast with jam

  Joe burnt the toast

MR: slow motion: Map



Input

Joe likes toast
Map 1

Jane likes toast with jam
Map 2

Joe burnt the toast
Map 3

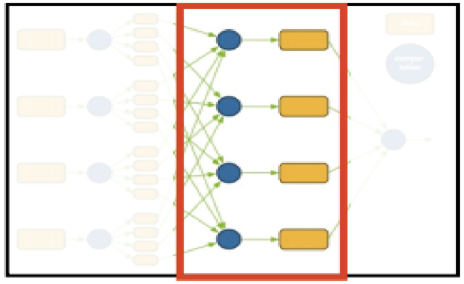
Output

Joe	1
likes	1
toast	1

Jane	1
likes	1
toast	1
with	1
jam	1

Joe	1
burnt	1
the	1
toast	1

MR: slow motion: Sort



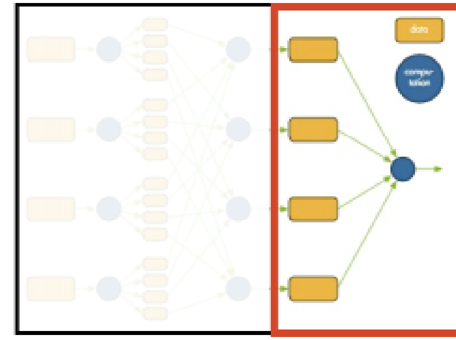
Input

Joe	1
likes	1
toast	1
Jane	1
likes	1
toast	1
with	1
jam	1
Joe	1
burnt	1
the	1
toast	1

Output

Joe	1
Joe	1
Jane	1
likes	1
likes	1
toast	1
toast	1
toast	1
with	1
jam	1
burnt	1
the	1

MR: slow mo: Reduce



Input

Joe	1	Reduce 1
Joe	1	
Jane	1	Reduce 2
likes	1	Reduce 3
likes	1	
toast	1	Reduce 4
toast	1	
toast	1	
with	1	Reduce 5
jam	1	Reduce 6
burnt	1	Reduce 7
the	1	Reduce 8

Output

Joe	2
Jane	1
likes	2
toast	3
with	1
jam	1
burnt	1
the	1

Map Reduce

Example uses:

- compute PageRank (matrix multiplication, …),
- build keyword indices,
- do data analysis of web click logs,
- financial artificial intelligence,
- geographical data
- relational algebra (select/project, …)
- …

Map Reduce

Operates at scales of 1000’s of machines

Handles failures seamlessly

Allows procedural code in map and reduce and allow data of any type

Not a solution to all problems!

Map Reduce

Here are a few examples of algorithms that might not be efficient when implemented using MapReduce:

- **Graph algorithms**: MapReduce is not well suited for algorithms that involve complex relationships between data elements, such as graph algorithms. The intermediate data representation required by MapReduce can make it difficult to capture the relationships between elements in a graph.
- **Real-time algorithms**: MapReduce is designed for batch processing, so it may not be suitable for real-time applications that require low latency.
- **Iterative algorithms**: MapReduce requires multiple rounds of data processing to arrive at the final result, which can be time-consuming for iterative algorithms.
- **Random access algorithms**: MapReduce is optimized for processing data in a sequential manner, so it may not be efficient for algorithms that require random access to data.
- **Algorithms with small datasets**: MapReduce is designed to handle large amounts of data, so it may not be efficient for small datasets. The overhead of the MapReduce framework can outweigh the benefits of parallel processing for small datasets.

NoSQL

- Not Only SQL
  - Not the other thing!
  - Term introduced by Carlo Strozzi in 1998 to describe an alternative database model
  - Became the name of a movement following Eric Evans’s reuse for a distributed-database event
- Seminal papers:
  - Google’s BigTable
    - Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, Gruber: Bigtable: A Distributed Storage System for Structured Data. OSDI 2006: 205-218
  - Amazon’s DynamoDB
    - DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, Vogels: Dynamo: Amazon's highly available key-value store. SOSP 2007: 205-220

NoSQL from nosql-database.org

"

- Next Generation Databases mostly addressing some of the points: being *non-relational*, *distributed*, *open-source* and *horizontally scalable*.
- The original intention has been modern web-scale databases. 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.
- So, 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.

"

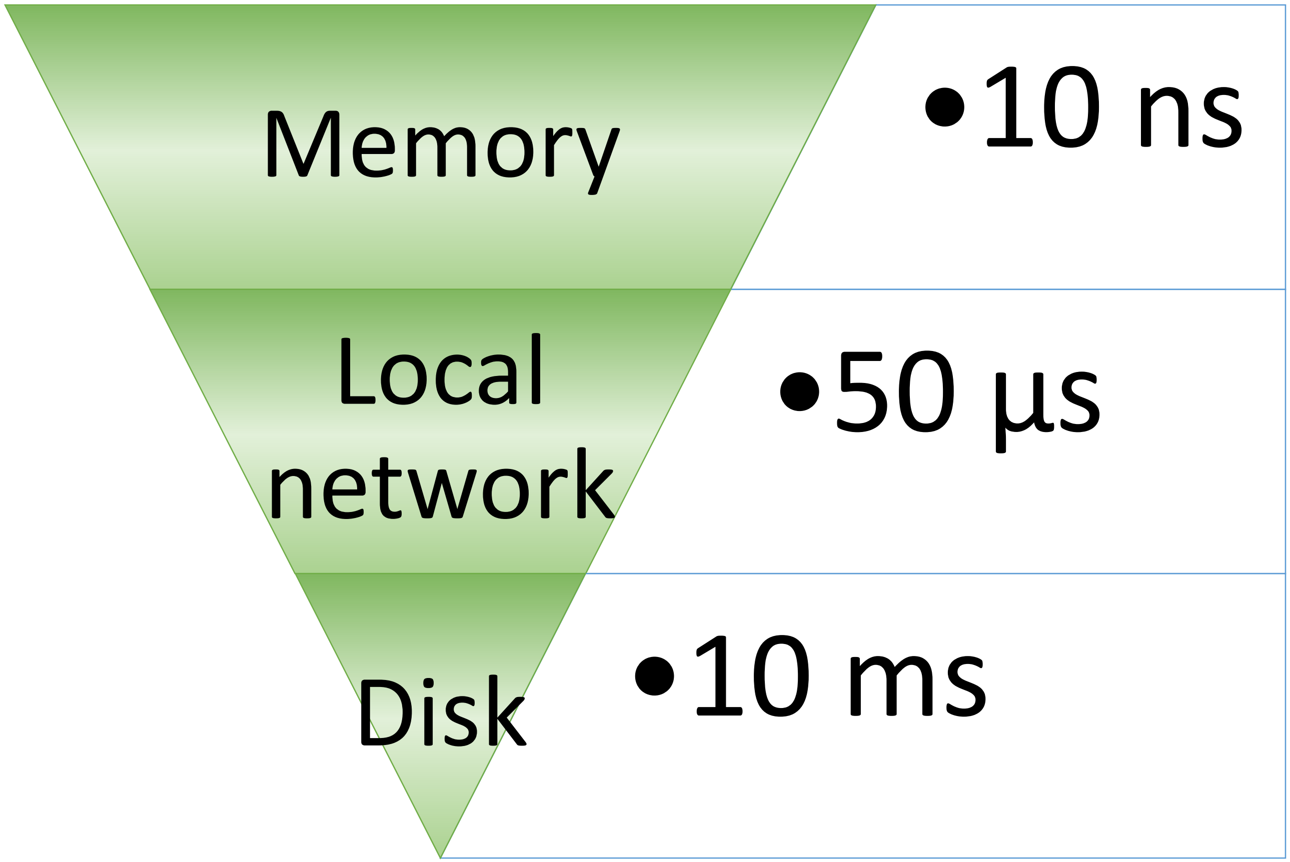
Common NoSQL Features

- Non-relational data models
- Flexible structure
  - No need to fix a schema, attributes can be added and replaced on the fly
- Massive read/write performance; availability via horizontal scaling
  - Replication and sharding (data partitioning)
  - Potentially thousands of machines worldwide
- Open source (very often)
- APIs to impose locality

Open Source

- Free software, source provided
  - Users have the right to use, modify and distribute the software
  - But restrictions may still apply, e.g., adaptations need to be opensource
- Idea: community development
  - Developers fix bugs, add features, ...
- *How can that work?*
  - See [Bonaccorsi, Rossi, 2003. Why open source software can succeed. *Research policy*, *32*(7), pp.1243-1258]
- A major driver of opensource is Apache

Performance - access time



It is faster to query another machine than read from the local disk

Transaction

- A sequence of operations (over data) viewed as a single higher-level operation
  - Transfer money from account 1 to account 2
- DBMSs execute transactions in parallel
  - No problem applying two “disjoint” transactions
  - But what if there are dependencies?
- Transactions can either commit (succeed) or abort (fail)
  - Failure due to violation of program logic, network failures, credit-card rejection, etc.
- DBMS should not expect transactions to succeed

Examples of Transactions

- Airline ticketing
  - Verify that the seat is vacant, with the price quoted, then charge credit card, then reserve
- Online purchasing
  - Similar
- "Transactional file systems" (MS NTFS)
  - Moving a file from one directory to another: verify file exists, copy, delete
- Textbook example: bank money transfer
  - Read from acct#1, verify funds, update acct#1, update acct#2

Predictable performance

Growing

1. Analytical thinking and innovation
2. Active learning and learning strategies
3. Creativity, originality and initiative
4. Technology design and programming
5. Critical thinking and analysis
6. Complex problem-solving
7. Leadership and social influence
8. Emotional intelligence
9. Reasoning, problem-solving and ideation
10. Systems analysis and evaluation

Declining

1. Manual dexterity, endurance and precision
2. Memory, verbal, auditory and spatial abilities
3. Management of financial, material resources
4. Technology installation and maintenance
5. Reading, writing, math and active listening
6. Management of personnel
7. Quality control and safety awareness
8. Coordination and time management
9. Visual, auditory and speech abilities
10. Technology use, monitoring and control

2022 Skills outlook

WORLD ECONOMIC FORUM
COMMITTED TO IMPROVING THE STATE OF THE WORLD

Forseeing failures

- The failure is the rule
- Amazon:
  - A data center of 100 000 disks
  - between 6000 and 10 000 discks down by year
  - (25 disks per day)
- The failure may have many sources
  - Server hardware (disk)
  - network equipment
  - power supply
  - software
  - software and OS updates.

NOSQL Advantages / Disadvantages

- Advantages of NoSQL DBMS
  - Performance (despite the amount of data)
  - Easily distributable
  - More flexible in case of failure

- Disadvantages of NoSQL DBMS
  - Less consistency of the database
  - No join mechanisms
  - More suited to semi-structured data
  - More oriented storage dedicated to one application

Quick reminder on the Cloud

Software as a Service

Platform as a Service

Infrastructure as a Service

open API

"Self Service"

"Pay as you go"

Abstraction of the location

Resource sharing

Elasticity

Purchasing: guarantees SLA

Service Level Agreement

- Availability near 99.9% (8h/year) for most players
  - Penalties as service extension for exceeding
- Failures in practice in 2009 at Amazon, Google, SalesForce
  - Down <2 days
- Different operating policies
  - Salesforce : Offer purely B2B
  - Google: very similar B2C and B2B
- Some youth in the business relationship ...

Purchasing: new terms

- Difficult to have a human partner (self service)
- Payment by credit card or PayPal : unusual…
- OPEX (operating expenses) / Subscription rather than CAPEX (investment expenses)
- Cost calculation not always trivial: cf. Amazon calculator
- Reduced costs not always proved

New acronyms!

- **ACID**

Atomic Consistent Isolated Durable

- **CAP** (Choose two out of three)

Consistent Available Partitionned

- **BASE**

BAsically available Soft State Eventually consistent

- **CRUD**

Create, Read, Update, Delete

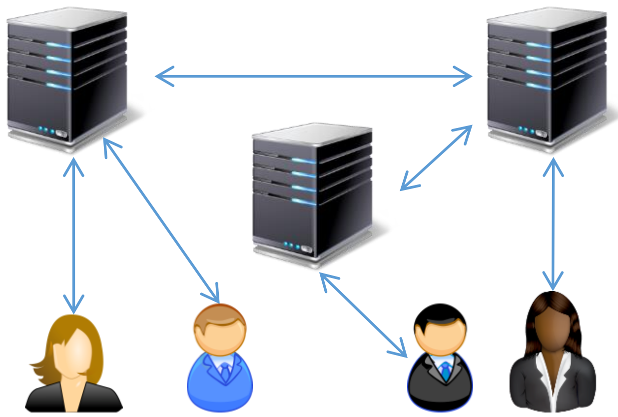
ACID transactions

- **A**tomicity
  - Either all operations applied or none are (hence, we need not worry about the effect of incomplete / failed transactions)
- **C**onsistency
  - Each transaction can start with a consistent database and is required to leave the database consistent
- **I**solation
  - The effect of a transaction should be as if it is the only transaction in execution (in particular, changes made by other transactions are not visible until committed)
- **D**urability
  - Once the system informs a transaction success, the effect should hold without regret, even if the database crashes (before making all changes to disk)

ACID May Be Overly Expensive

- In quite a few modern applications:
  - ACID contrasts with key desiderata: high volume, high availability
  - We can live with some errors, to some extent
  - Or more accurately, we prefer to suffer errors than to be significantly less functional
- *Can this point be made more "formal"?*

Simple Model of a Distributed Service

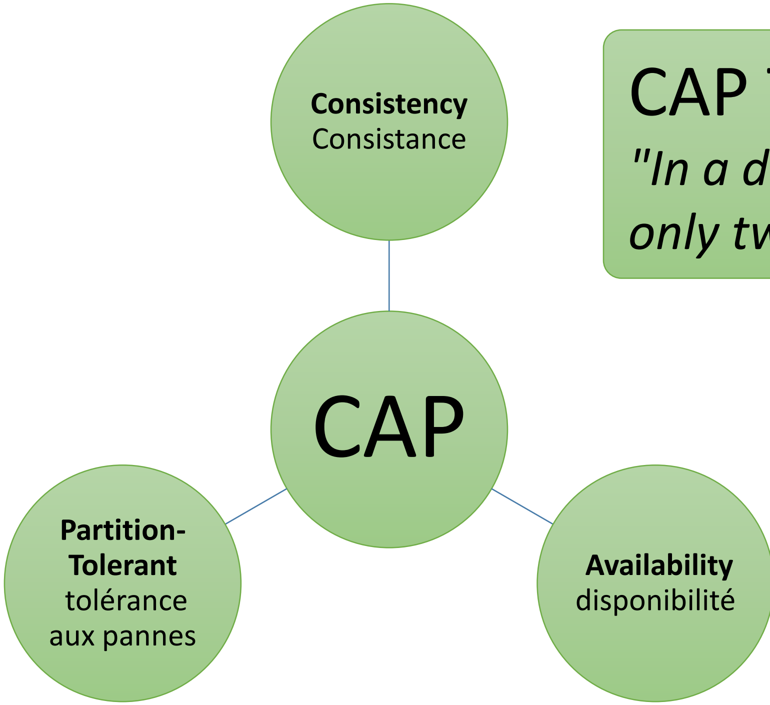


- Context: distributed service
  - e.g., social network
- Clients make get / set requests
  - e.g., setLike(user,post), getLikes(post)
  - Each client can talk to any server
- Servers return responses
  - e.g., ack, {$user_1$,....,$user_k$}
- Failure: the network may occasionally disconnect due to failures (e.g., switch down)
- Desiderata: **C**onsistency, **A**vailability, **P**artition tolerance

CAP Service Properties

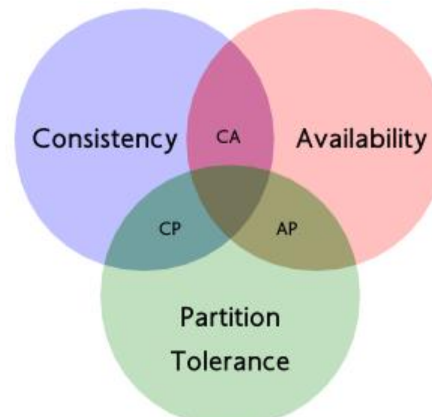
- **C**onsistency: every read (to any node) gets a response that reflects the most recent version of the data
  - More accurately, a transaction should behave as if it changes the entire state correctly in an instant
  - Idea similar to serializability
- **A**vailability: every request (to a living node) gets an answer: set succeeds, get retunes a value
- **P**artition tolerance: service continues to function on network failures
  - As long as clients can reach servers

CAP theorem



CAP Theorem

"In a distributed architecture, it is possible to ensure only two of the three CAP properties ".



Cloud Actors usually prefer A and P properties
-> Horizontal Scalability
-> Banalisation server components

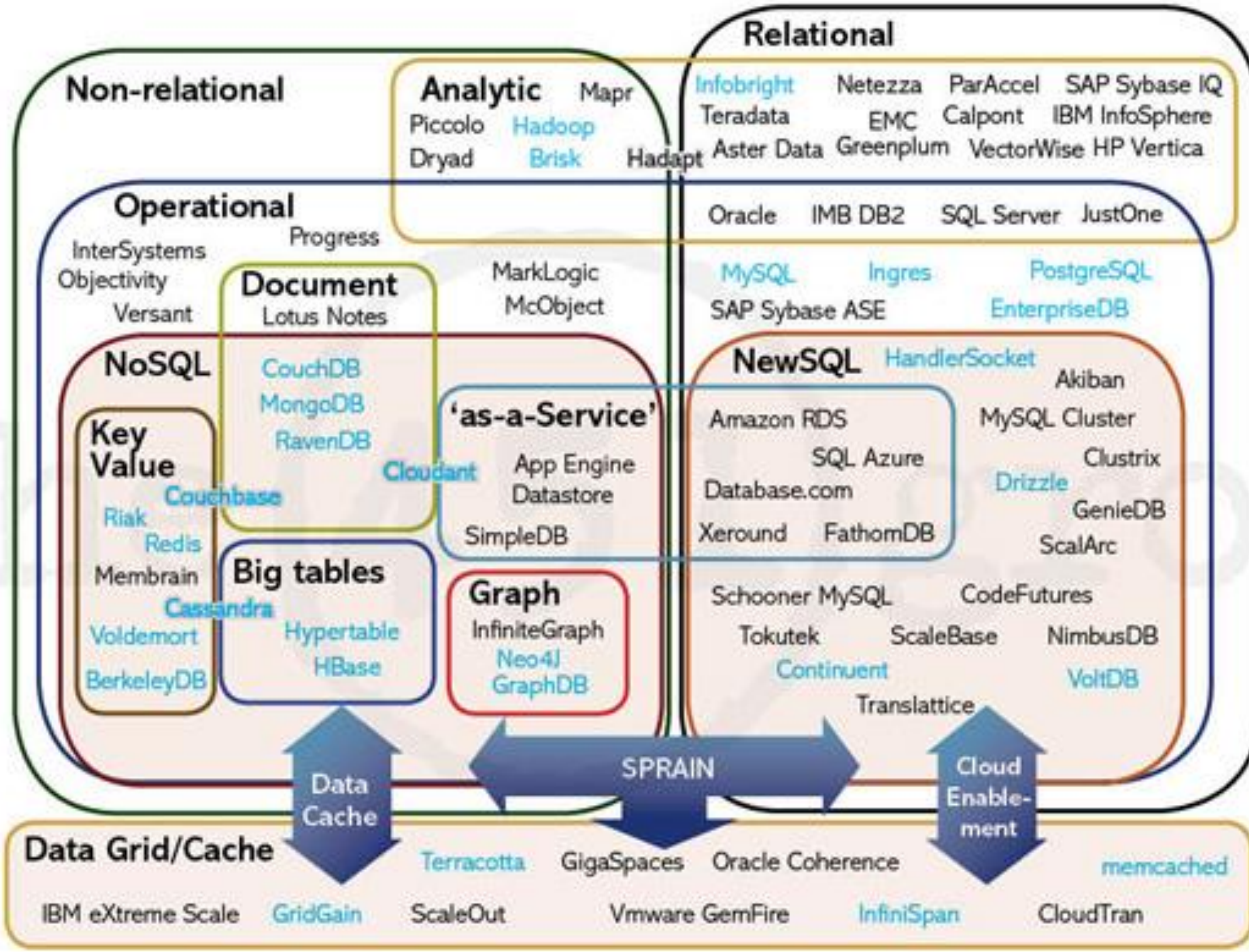
Amazon dilemma

When a customer clicks
the button "buy"
Should we?

The BASE Model

- Applies to distributed systems of type AP
- **B**asic **A**vailability
  - Provide high availability through distribution
- **S**oft state
  - Inconsistency (stale answers) allowed
- **E**ventual consistency
  - If updates stop, then after some time consistency will be achieved
    - Achieved by protocols to propagate updates and verify correctness of propagation (gossip protocols)
- Philosophy: best effort, optimistic, staleness and approximation allowed

DBMS-s



https://451research.com/

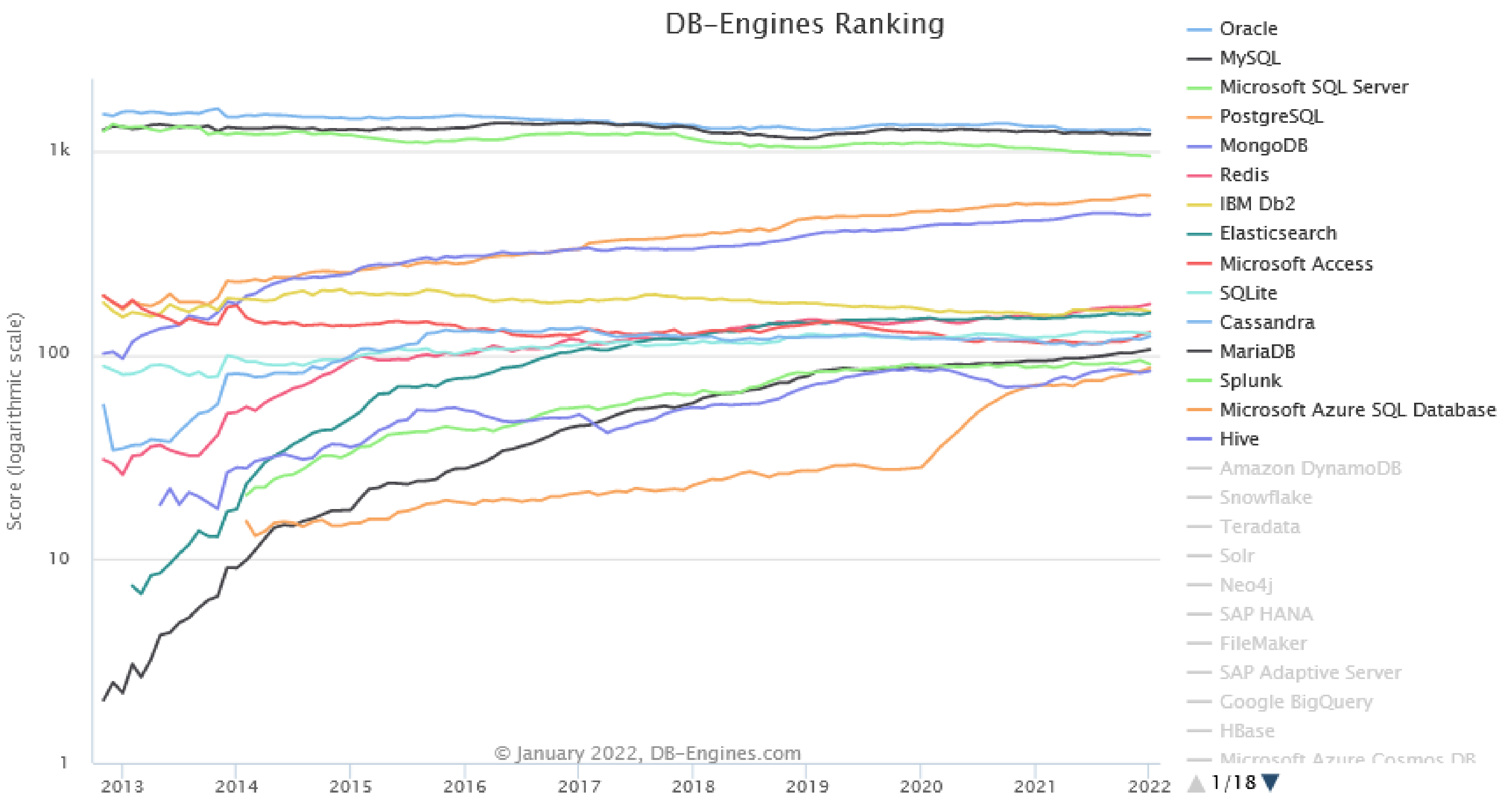
DBMSs

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

383 systems in ranking, January 2022

Rank Jan 2022	Rank Dec 2021	Rank Jan 2021	DBMS	Database Model	Score Jan 2022	Score Dec 2021	Score Jan 2021
1.	1.	1.	Oracle	Relational, Multi-model	1266.89	-14.85	-56.05
2.	2.	2.	MySQL	Relational, Multi-model	1206.05	+0.01	-46.01
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	944.81	-9.21	-86.42
4.	4.	4.	PostgreSQL	Relational, Multi-model	606.56	-1.66	+54.33
5.	5.	5.	MongoDB	Document, Multi-model	488.57	+3.89	+31.34
6.	6.	7.	Redis	Key-value, Multi-model	177.98	+4.44	+22.97
7.	7.	6.	IBM Db2	Relational, Multi-model	164.20	-2.98	+7.03
8.	8.	8.	Elasticsearch	Search engine, Multi-model	160.75	+3.03	+9.50
9.	10.	11.	Microsoft Access	Relational	128.95	+2.96	+13.61
10.	9.	9.	SQLite	Relational	127.43	-1.25	+5.54
11.	11.	10.	Cassandra	Wide column	123.55	+4.35	+5.47
12.	12.	12.	MariaDB	Relational, Multi-model	106.42	+2.06	+12.63
13.	13.	13.	Splunk	Search engine	90.45	-3.87	+2.79
14.	14.	15.	Microsoft Azure SQL Database	Relational, Multi-model	86.32	+3.07	+14.96
15.	15.	16.	Hive	Relational	83.45	+1.52	+13.02
16.	16.	17.	Amazon DynamoDB	Multi-model	79.85	+2.23	+10.72
17.	17.	37.	Snowflake	Relational	76.82	+5.79	+61.30
18.	18.	14.	Teradata	Relational, Multi-model	69.13	-1.17	-3.46
19.	20.	20.	Solr	Search engine, Multi-model	58.53	+0.80	+6.04
20.	19.	19.	Neo4j	Graph	58.03	0.00	+4.25
21.	21.	21.	SAP HANA	Relational, Multi-model	56.92	+2.34	+6.05
22.	22.	22.	FileMaker	Relational	55.86	+1.99	+8.47
23.	23.	18.	SAP Adaptive Server	Relational, Multi-model	51.05	-0.33	-3.56
24.	24.	24.	Google BigQuery	Relational	45.62	-0.18	+9.62
25.	25.	23.	HBase	Wide column	43.99	-1.55	-2.29

DBMS trends



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

Plan

- Introduction
- Core XML
- XML galaxy

NOSQL

Introduction

Basic concepts

Column family

Key-Value Store

Graph DBMS

Document Store

- Conclusion

Column Stores

- Common idea: don’t keep a row in a consecutive block, split via projection
  - Column store: each column is independent; column-family store: each column family is independent
- Both provide some major efficiency benefits in common read-mainly workloads
  - Given a query, load to memory only the relevant columns
  - Columns can often be highly compressed due to value similarity
  - Effective form for sparse information (no NULLs, no space)
- Column-family store is handled differently from RDBs, often requiring a designated query language

Examples Systems

- Column store (SQL):
  - MonetDB (started 2002, Univ. Amsterdam)
  - VectorWise (spawned from MonetDB)
  - Vertica (M. Stonebraker)
  - SAP Sybase IQ
  - Infobright
- Column-family store (NOSQL):
  - Google's BigTable (main inspiration to column families)
  - Apache HBase (used by Facebook, LinkedIn, Netflix...)
  - Hypertable
  - Apache Cassandra
    - Read more : http://wiki.apache.org/cassandra/GettingStarted

Example: Apache Cassandra

- Initially developed by Facebook
  - Open-sourced in 2008
- Used by 1500+ businesses, e.g., Comcast, eBay, GitHub, Hulu, Instagram, Netflix, Best Buy, ...
- Column-family store
  - Supports key-value interface
  - Provides a SQL-like CRUD interface: CQL
- Uses Bloom filters
  - An interesting membership test that can have false positives but never false negatives, well behaves statistically
- BASE consistency model (_AP)
  - Gossip protocol (constant communication) to establish consistency
  - Ring-based replication model

Cassandra: Outline

- **Extension of Bigtable with aspects of Dynamo**
- **Motivations:**
  - **High Availability**
  - **High Write Throughput**
  - **Fail Tolerance**

Cassandra: Data Model

Table is a multi dimensional map indexed by key (row key).

Columns are grouped into Column Families.

2 Types of Column Families

- Simple
- Super (nested Column Families)

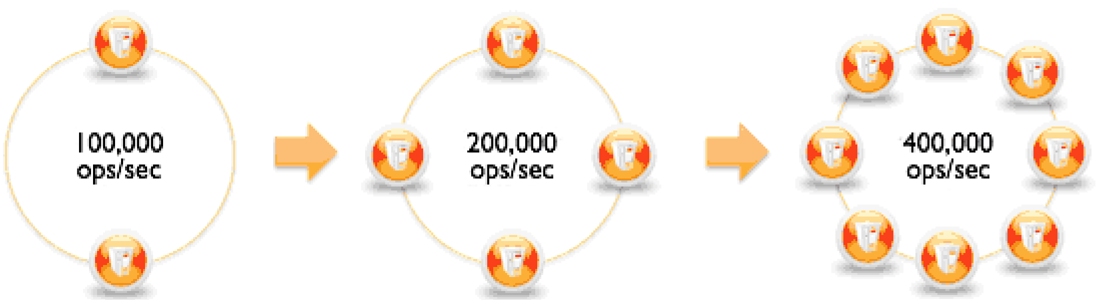
Each Column has

- Name
- Value
- Timestamp

Cassandra architecture

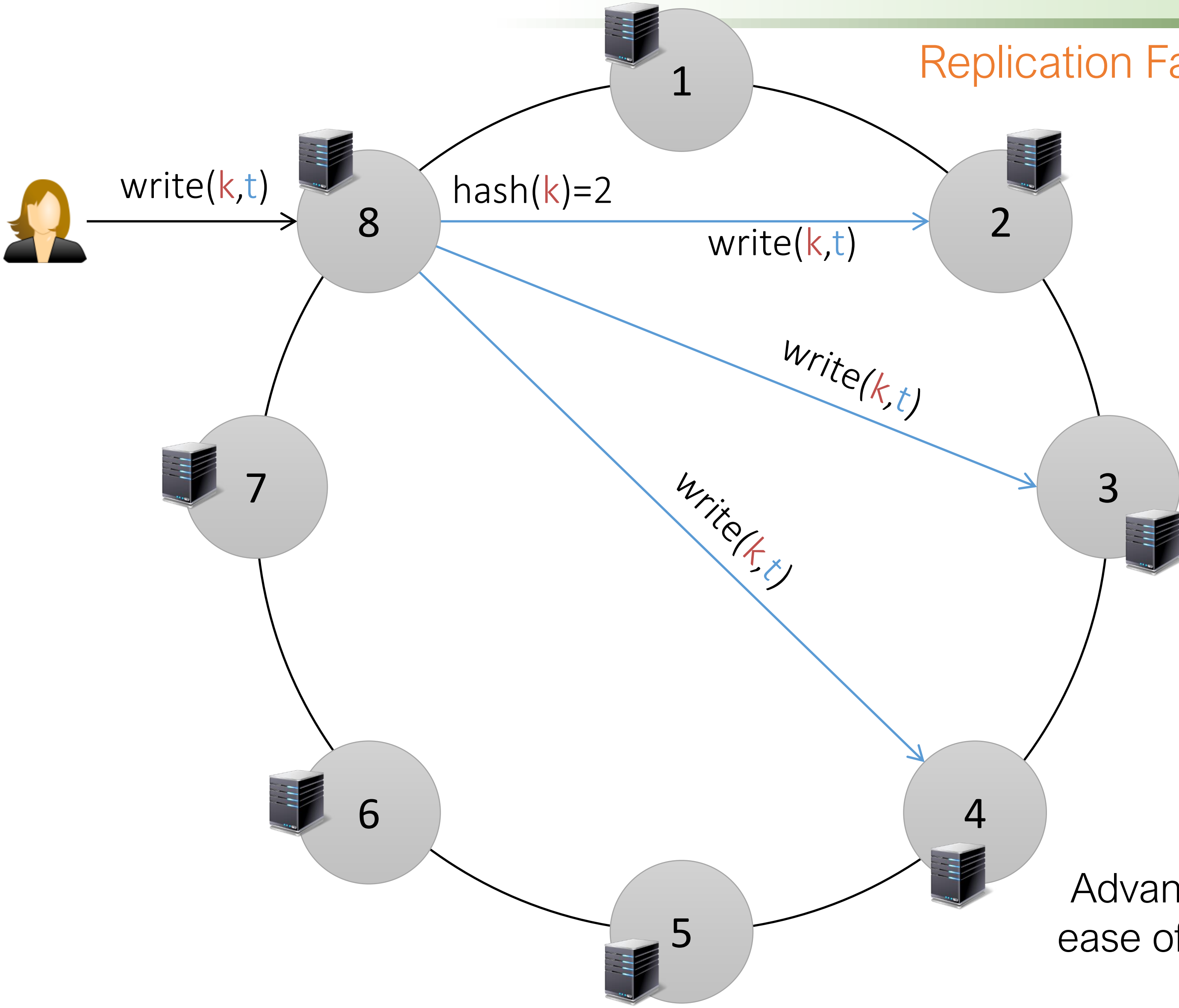
Cassandra has a "masterless" architecture.

Cassandra provides customizable replication, storing redundant copies of data across nodes that participate in a Cassandra ring.



Cassandra’s Ring Model

Replication Factor = 3



Advantage: Flexibility / ease of cluster redesign

Plan

- Introduction
- Core XML
- XML galaxy

NOSQL

- Introduction
- Basic concepts
- Column family
- **Key-Value Store**
- Graph DBMS
- Document Store

- Conclusion

Key-Value Stores

- Essentially, big distributed hash maps
- Origin attributed to Dynamo – Amazon's DB for world-scale catalog/cart collections
  - But Berkeley DB has been here for >20 years
- Store pairs ⟨key,opaque-value⟩
  - Opaque means that DB does not associate any structure/semantics with the value; *oblivious* to values
  - This may mean more work for the user: retrieving a large value and parsing to extract an item of interest
- Sharding via partitioning of the key space
  - Hashing, gossip and remapping protocols for load balancing and fault tolerance

Example Databases

- Berkley DB (Oracle)
  - consistent
  - Master / slave
- memcached
  - memcachedb = memcached + BerkeleyDB
- membase (Couchbase.org)
  - Erlang
- Riak
  - Coherent
  - Erlang
- **Redis** (vmware) next slides
  - Coherent
  - in memory ; asynchronous disk write
  - evolved types (map list) and advanced operations associated

- Dynamo (Amazon)
  - Indirect Use tools with Amazon AWS
- Voldemort (LinkedIn)
- GigaSpace
- Infinityspan (RedHat, JBoss)
  - Hibernate GMO

Redis (REmote DIctionary Server)

- Basically a data structure for strings, numbers, hashes, lists, sets
- Ultra-fast in-memory key-value data store
- Simplistic “transaction” management
  - Queuing of commands as blocks, really
  - Among ACID, only Isolation guaranteed
    - A block of commands that is executed sequentially; no transaction interleaving; no roll back on errors
- In-memory store
  - Persistence by periodical saves to disk
- Comes with
  - A command-line API
  - Clients for different programming languages
    - Perl, PHP, Rubi, Tcl, C, C++, C#, Java, R, …

Example of Redis Commands

key	value

```
get x
>> 10
```

```
hget h y
>> 5
```

```
hkeys p:22
>> name , age
```

```
smembers s
>> 20 , Jean
```

```
scard s
>> 2
```

```
llen l
>> 3
```

```
lrange l 1 2
>> a , b
```

```
lindex l 2
>> b
```

```
lpop l
>> c
```

```
rpop l
>> b
```

Example of Redis Commands

	Commands	key	value
(simple value)	set x 10	x	10
(hash table)	hset h y 5	h	y→5
	hset h1 name two hset h1 value 2	h1	name→two value→2
	hmset p:22 name Jean age 25	p:22	name→Jean age→25
(set)	sadd s 20 sadd s Jean sadd s Jean	s	{20,Jean}
(list)	rpush l a rpush l b lpush l c	l	(c,a,b)

```
get x
>> 10

hget h y
>> 5

hkeys p:22
>> name , age

smembers s
>> 20 , Jean

scard s
>> 2

llen l
>> 3

lrange l 1 2
>> a , b

lindex l 2
>> b

lpop l
>> c

rpop l
>> b
```

Additional Notes

- A key can be any <256MB binary string
  - For example, JPEG image
- Some key operations:
  - List all keys: `keys *`
  - Remove all keys: `flushall`
  - Check if a key exists: `exists k`
- You can configure the persistency model
  - `save m k` means save every `m` seconds if at least `k` keys have changed
- Redis is not a database
  - It complements your existing data storage layer
  - E.g. StackOverflow uses Redis for data caching

What is redis not good for

1. Neither SQL nor NoSQL
2. Need ACID Transaction
3. Every byte is precious
4. Single threading
5. Memory problem
6. **Security**

Plan

- Introduction
- Core XML
- XML galaxy

NOSQL

Introduction

Basic concepts

Column store

Key-Value Store

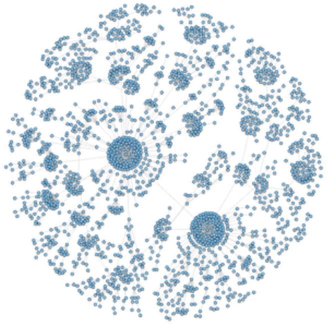
Graph DBMS

Document Store

- Conclusion

Graph Databases

- Restricted case of a relational schema:
  - Nodes (+labels/properties)
  - Edges (+labels/properties)
- Motivated by the popularity of network/communication oriented applications
- Efficient support for graph-oriented queries
  - *Reachability*, *graph patterns*, *path patterns*
  - Ordinary RDBs either not support or inefficient for such queries
    - Path of length k is a k-wise self join; yet a very special one...
- Specialized languages for graph queries
  - For example, pattern language for paths
- Plus distributed, 2-of-CAP, etc.
  - Depending on the design choices of the vendor



Example Databases

- Graph with nodes/edges marked with labels and properties (labeled property graph)
  - Sparksee (DEX) (Java, 1st release 2008)
  - neo4j (Java, 1st release 2010)
  - InfiniteGraph (Java/C++, 1st release 2010)
  - OrientDB (Java, 1st release 2010)
- Triple stores: Support W3C RDF and SPARQL, also viewed as graph databases
  - MarkLogic, AllegroGraph, Blazegraph, IBM SystemG, Oracle Spatial & Graph, OpenLink Virtuoso, ontotext

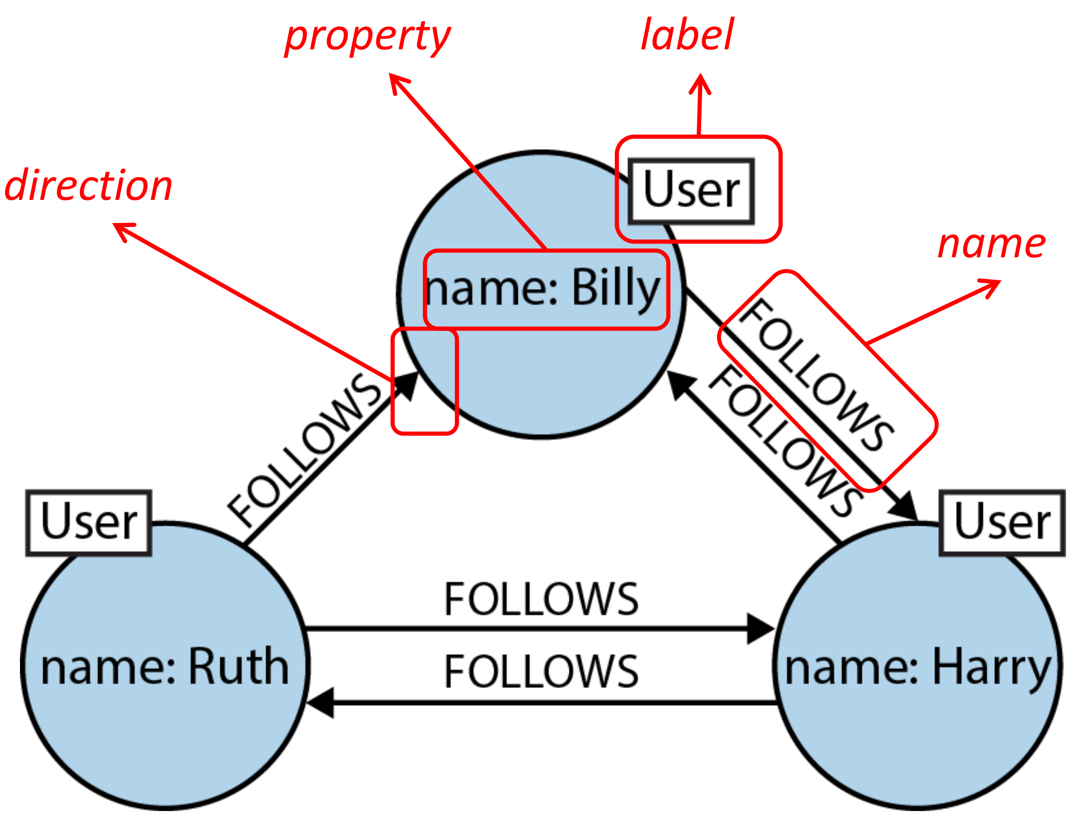
neo4j

- Open source, written in Java
  - First version released 2010
- Supports the Cypher query language
- Clustering support
  - Replication and sharding through master-slave architectures
- Used by ebay, WJeanrt, Cisco, National Geographic, TomTom, Lufthansa, ...

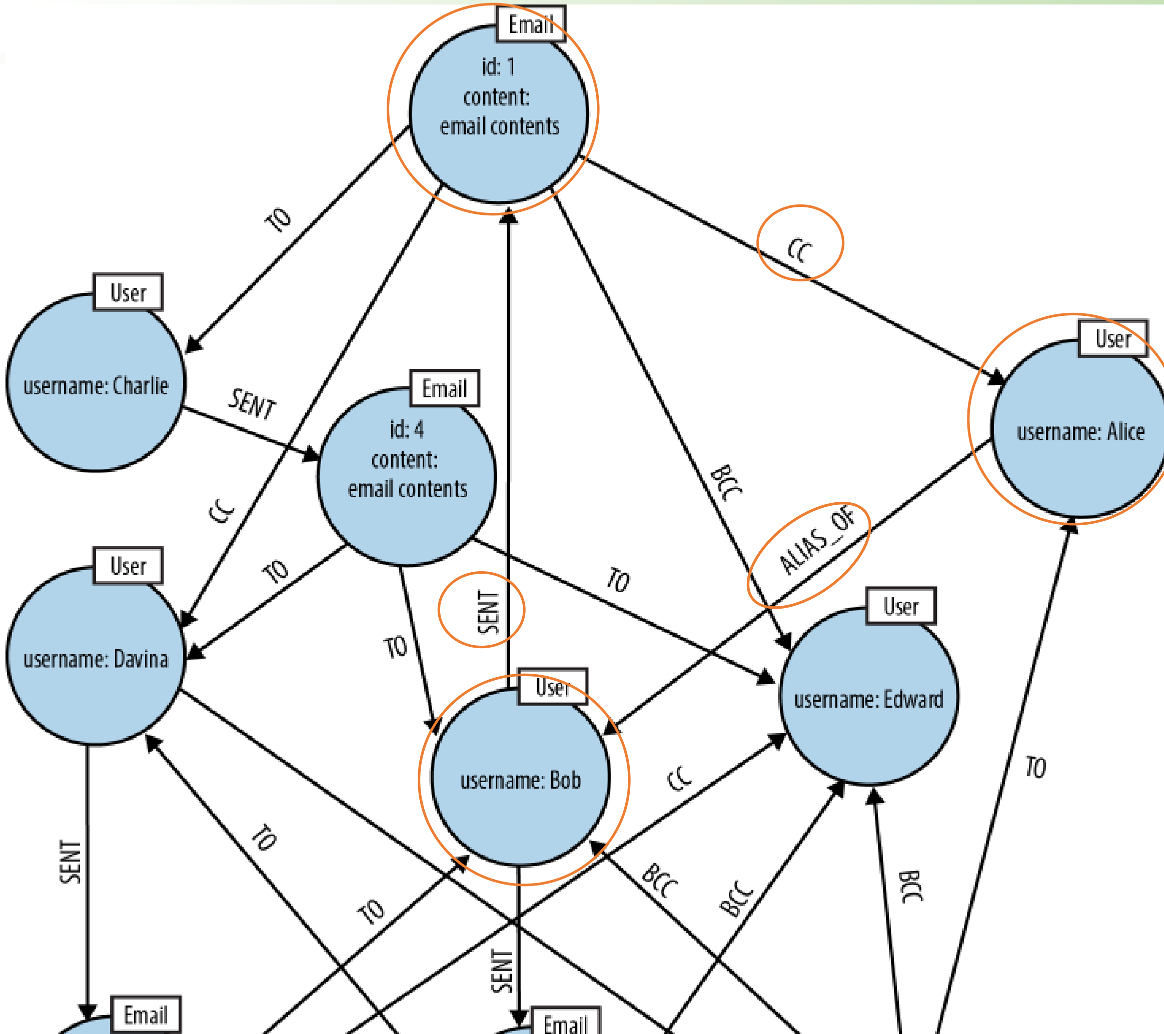
The Graph Data Model in Cypher

- Labeled property graph model
- Node
  - Has a set of *labels* (typically one label)
  - Has a set of *properties* key:value (where value is of a primitive type or an array of primitives)
- Edge (relationship)
  - Directed: node→node
  - Has a *name*
  - Has a set of *properties* (like nodes)

Example: Cypher Graph for Social Networks



Query Example

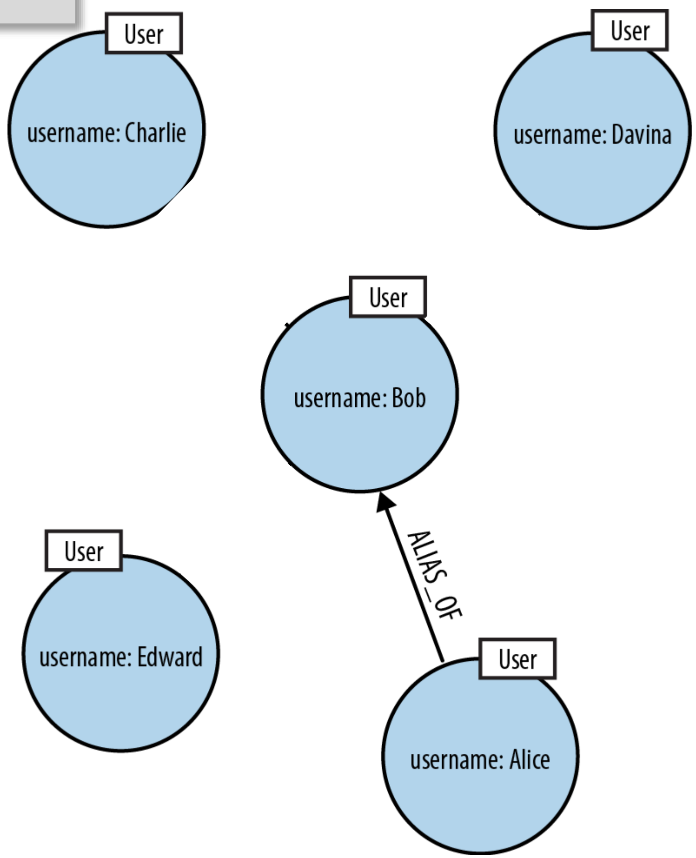


email
Node{id:"1",content:"..."}

```
MATCH (bob:User{username:'Bob'})-[:SENT]->(email)-[:CC]->(alias),
      (alias)-[:ALIAS_OF]->(bob)
RETURN email
```

Creating Graph Data

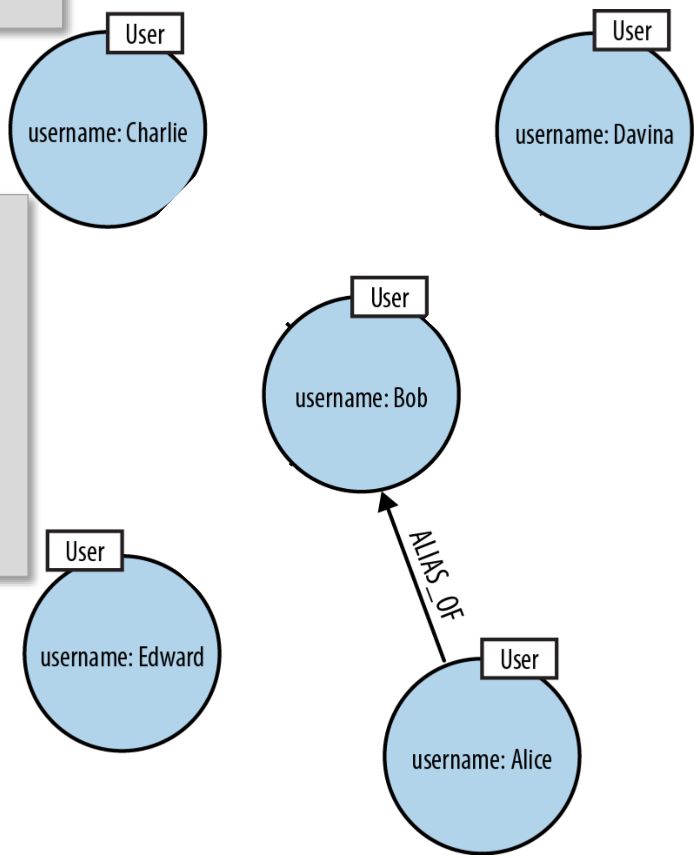
```
CREATE (alice:User {username:'Alice'}),
       (bob:User {username:'Bob'}),
       (charlie:User {username:'Charlie'}),
       (davina:User {username:'Davina'}),
       (edward:User {username:'Edward'}),
       (alice)-[:ALIAS_OF]->(bob)
```



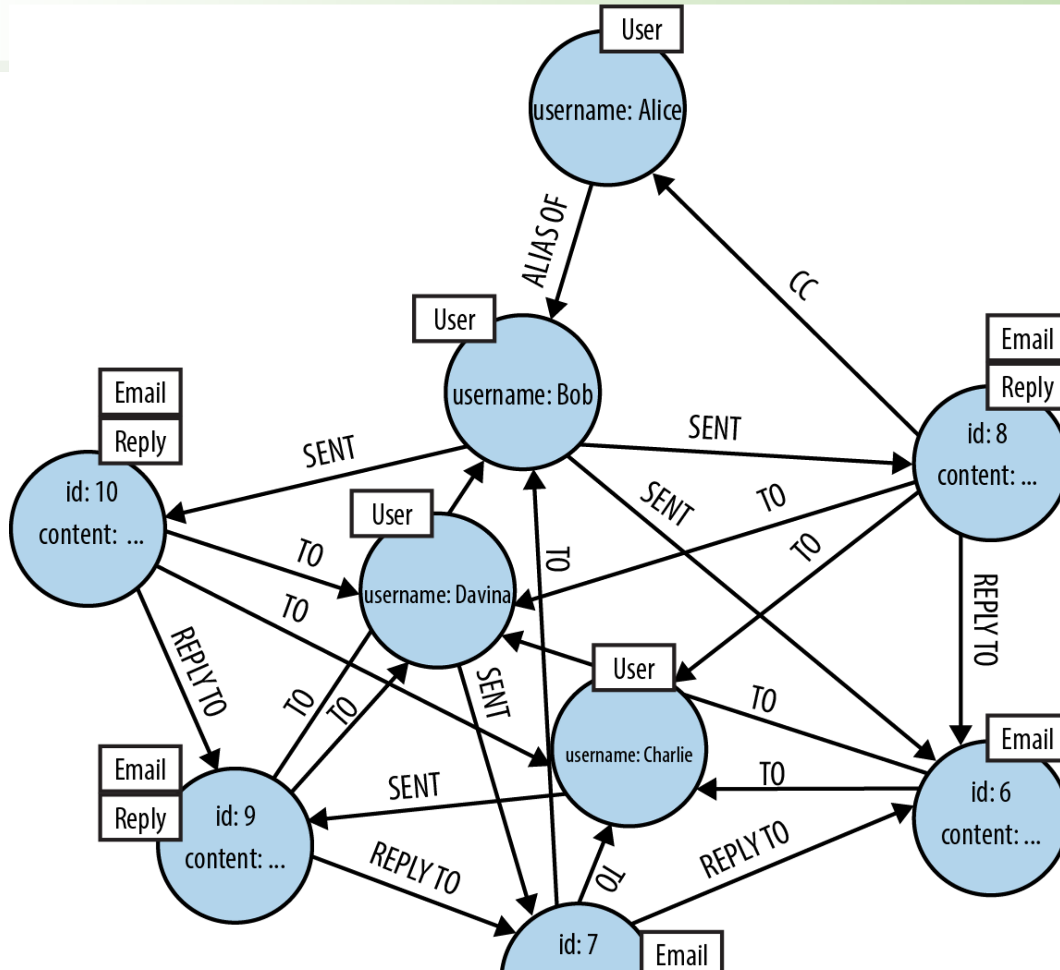
Creating Graph Data

```
CREATE (alice:User {username:'Alice'}),
       (bob:User {username:'Bob'}),
       (charlie:User {username:'Charlie'}),
       (davina:User {username:'Davina'}),
       (edward:User {username:'Edward'}),
       (alice)-[:ALIAS_OF]->(bob)
```

```
MATCH  (bob:User {username:'Bob'}),
       (charlie:User {username:'Charlie'}),
       (davina:User {username:'Davina'}),
       (edward:User {username:'Edward'})
CREATE  (bob)-[:EMAILED]->(charlie),
        (bob)-[:CC]->(davina),
        (bob)-[:BCC]->(edward)
```



Another Example



replier	depth
Davina	1
Bob	1
Charlie	2
Bob	3

```
MATCH p = (email:Email {id:'6'})
      <-[:REPLY_TO*1..4]-(:Reply)<-[:SENT]-(replier)
RETURN replier.username AS replier, length(p) - 1 AS depth
ORDER BY depth
```

Plan

- Introduction
- Core XML
- XML galaxy

NOSQL

- Introduction
- Basic concepts
- Column store
- Key-Value Store
- Graph DBMS
- **Document Store**

- Conclusion

Document Stores

- Similar in nature to key-value store, but value is tree structured as a *document*
- Motivation: avoid joins; ideally, all relevant joins already encapsulated in the document structure
- A document is an atomic object that cannot be split across servers
  - But a document *collection* will be split
- Moreover, transaction atomicity is typically guaranteed within a single document
- Model generalizes column-family and key-value stores

Example Databases

- MongoDB
  - Next slides
- Apache CouchDB
  - Emphasizes Web access
- RethinkDB
  - Optimized for highly dynamic application data
- RavenDB
  - Deigned for .NET, ACID
- Clusterpoint Server
  - XML and JSON, a combined SQL/JavaScript QL
- OrientDB
  - Java embeddable
- Terrastore

mongoDB

- Open source, 1st release 2009, document store
  - Actually, an extended format called BSON (binary JSON) for typing and better compression
- Supports replication (master/slave), sharding
  - Developer provides the “shard key” – collection is partitioned by ranges of values of this key
- Consistency guarantees, CP of CAP
- Used by Adobe (experience tracking), Craigslist, eBay, FIFA (video game), LinkedIn, McAfee
- Provides connector to Hadoop
  - Cloudera provides the MongoDB connector in distributions

MongoDB Data Model

- JavaScript Object Notation (JSON) model
- *Database* = set of named *collections* → *generalizes relation*
- *Collection* = sequence of *documents* → *generalizes tuple*
- *Document* = {attribute_1:value_1,...,attribute_k:value_k}
- *Attribute* = string ($\text{attribute}_i \neq \text{attribute}_j$)
- *Value* = primitive value (string, number, date, ...), or a document, or an *array*
- *Array* = [value_1,...,value_n]

- Key properties: hierarchical (like XML), no schema
  - Collection docs may have different attributes

MongoDB vs Relational DBMS

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
Partition	Sharding
Clustering	ReplicaSet
Joining	Linking & Embedding

Data Example

Collection **inventory**

```
{
	item: "ABC2",
	details: { model: "14Q3", manufacturer: "M1 Corporation" },
	stock: [ { size: "M", qty: 50 } ],
	category: "clothing”
}

{
	item: "MNO2",
	details: { model: "14Q3", manufacturer: "ABC Company" },
	stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "L", qty: 1 } ],
	category: "clothing”
}
```

```
db.inventory.insert(
  {
   item: "ABC1",
   details: {model: "14Q3",manufacturer: "XYZ Company"},
   stock: [ { size: "S", qty: 25 }, { size: "M", qty: 50 } ],
   category: "clothing"
  }
)
```

Document insertion

(docs.mongodb.org)

Example of a Simple Query

Collection orders

```
{
    _id: "a",
    cust_id: "abc123",
    status: "A",
    price: 25,
    items: [ { sku: "mmm", qty: 5, price: 3 },
             { sku: "nnn", qty: 5, price: 2 } ]
}
{
    _id: "b",
    cust_id: "abc124",
    status: "B",
    price: 12,
    items: [ { sku: "nnn", qty: 2, price: 2 },
             { sku: "ppp", qty: 2, price: 4 } ]
}
```

```
db.orders.find(
    { status: "A" },                      → selection
    { cust_id: 1, price: 1, _id: 0 }      → projection
)
```

In SQL it would look like this:

```
SELECT cust_id, price
  FROM orders
WHERE status="A"
```

```
{
    cust_id: "abc123",
    price: 25
}
```

JSON

JSON " **J**ava**S**cript **O**bject **N**otation "is a formatted exchange data readable by a human and interpreted by a machine.

Based on JavaScript, it is completely independent of programming languages

Two structures:

- A unordered collection of key/values ➔ **Object**
- An ordered collection of objects ➔ **Array**

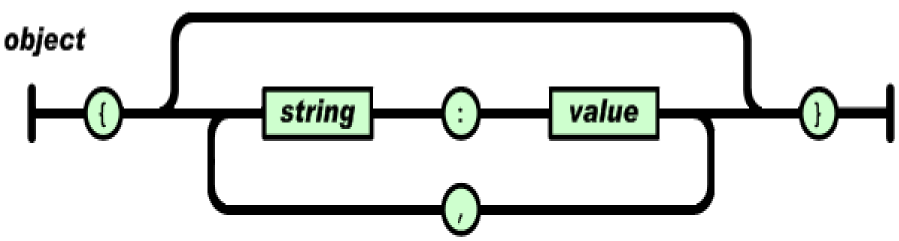
Basic constructs (recursive)

- Base values
  number, string, boolean, …
- Objects { }
  sets of label-value pairs
- Arrays [ ]
  lists of values

JSON

Object

Starts with a " **{** " and ends with" **}**" and consists of an unordered list of keys/value pairs . A key is followed by "**:**" and the key / value pairs are separated by " **,** "

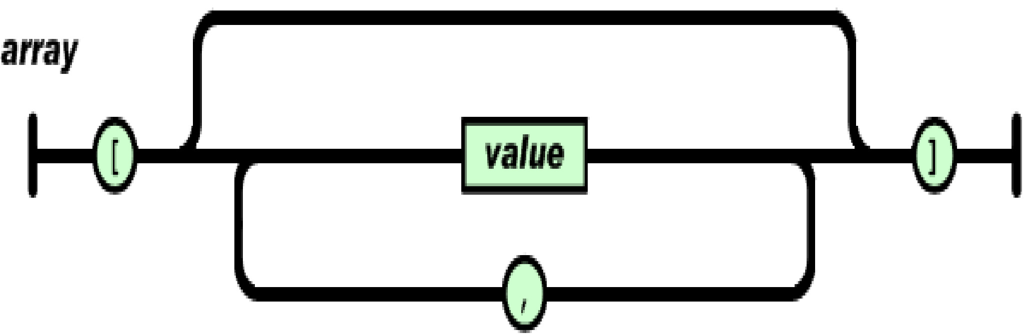



```
{ "Id": 51,
"name": "Mathematics 1 ", "resume":
"Summary of math ", "isbn":
"123654",
"category":
    {
    "id ": 2, "name": "Mathematics",
    "description": "Description of
    mathematics "
    } ,
"amount": 42,
"Photo": ""
}
```

JSON

ARRAY

ordered list of objects that begin with " [" and end with"] ". Objects are separated from each other by" , ".

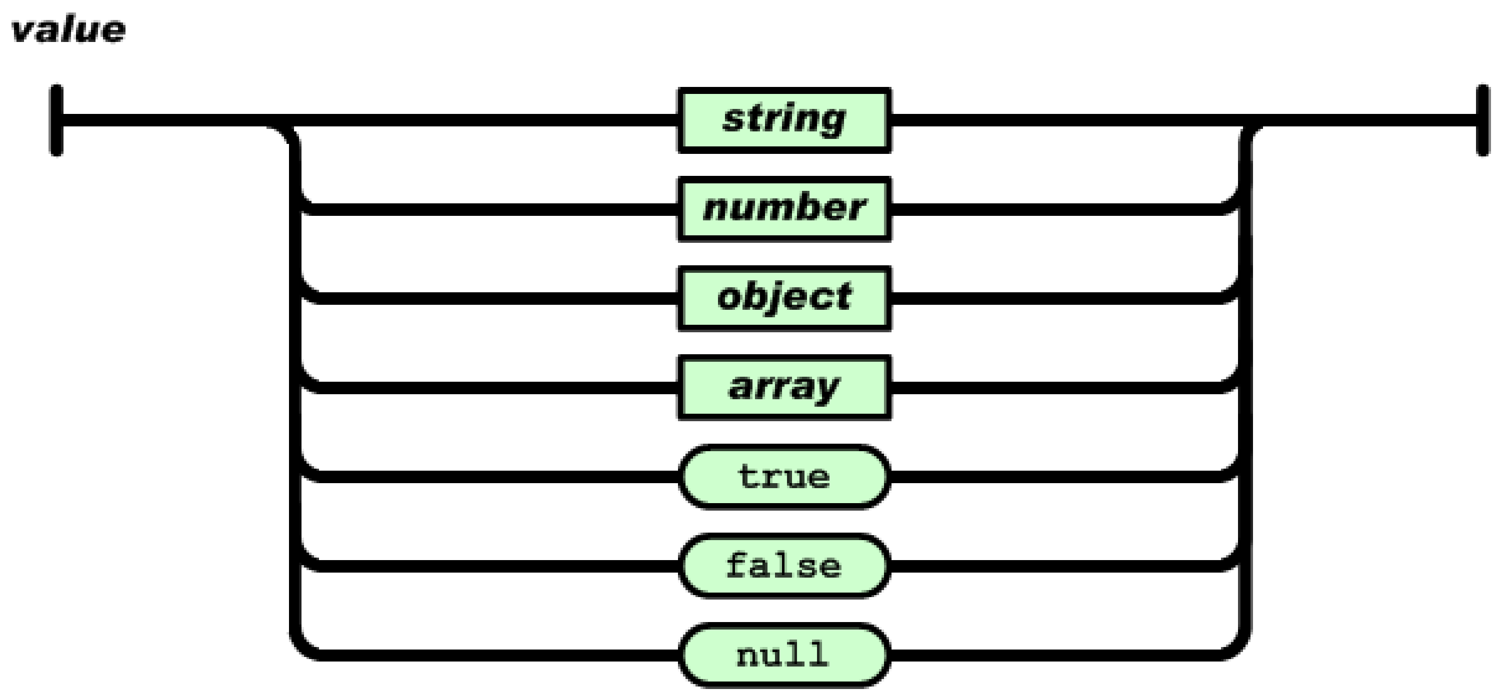



```
[
{ "Id": 51,
"name": "Mathematics 1 ",
"resume": "Resume of math",
"isbn": "123654",
"amount": 42,
"Photo": ""
}
{ "Id": 102,
"name": "Mathematics 1 ",
"resume": "Resume of math",
"isbn": "12365444455",
"amount": 42,
"Photo": ""
}
]
```

JSON

Value

An object can be either a string between "" or a number (integer, decimal) or boolean (true, false) or null or an object.



Much Like XML

- Plain text formats
- “Self-describing“ (human readable)
- Hierarchical (Values can contain lists of objects or values)

Not Like XML

- Lighter and faster than XML
- JSON uses typed objects. All XML values are type-less strings and must be parsed at runtime.
- Less syntax, no semantics
- Properties are immediately accessible to JavaScript code

Knocks against JSON

- Lack of namespaces
- No inherit validation (XML has DTD and templates, but there is JSONlint)
- Not extensible
- It’s basically just ***not*** XML

MongoDB Some terms

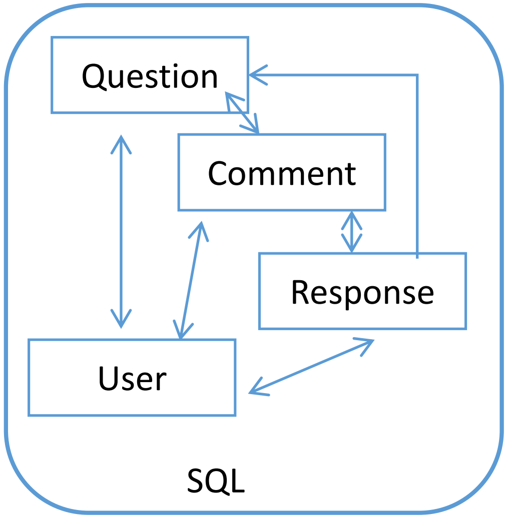
- **Cluster MongoDB**: Set of processes spread over multiple nodes, covers all routers (mongos) + config servers (mongod) + shards (mongod).
- **Router**: Treating process requests from client drivers, research data in the shards.
- **Shard**: Set of processes mongodA primary, secondary n => replication data type master-slave, one arbiter (failure of one or more mongod).
- **MongoDB DataBase**: The basic data are distributed by shardingon the cluster nodes; in oneshardThe base is duplicated by replication; a base consists of a set of collections.
- **Collection**: A collection contains a set of documents in JSON format.

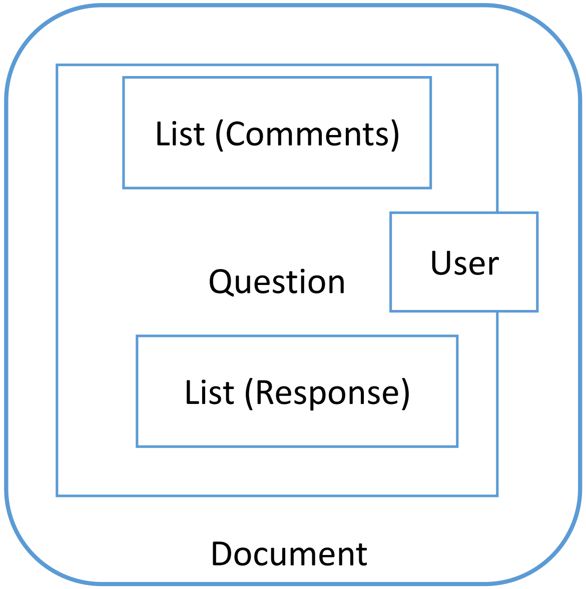
Understanding MongoDb components

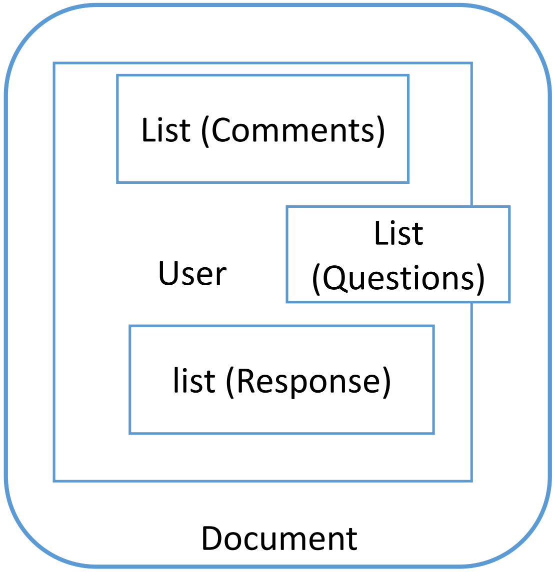
Component Set	Binaries
Server	mongod.exe
Router (Sharding service)	mongos.exe
Client	mongo.exe
Monitoring Tools	mongostat.exe, mongotop.exe
ImportExportTools	mongodump.exe, mongorestore.exe, mongoexport.exe, mongoimport.exe
MiscellaneousTools	bsondump.exe, mongofiles.exe, mongooplog.exe, mongoperf.exe monoreplay.exe , mongoldap.exe

De normalizing

- Relational / Document







Why use MongoDB?

You must store unstructured data

You have a very high write load (without transactions)

You need to handle more reads & writes than a single server can handle

You need a solution that can easily scale-out(sharding)

You work with tables with very inconsistent schemas

You need high availability solution built-in (ReplicaSets)

You need high performance (most of the data is stored in ram)

You need built in geospatial functions

Why you wouldn't want to use MongoDB?

- No support for transactions
- Limited support for joins
- No support for triggers
- Document size limit (16 mb)
- Your data is relational
- You don't want duplicate data.

Resources

Official site

www.mongodb.org

Documents

https://docs.mongodb.com/

Elasticsearch

Text based document search engine

- Distributed RESTful search server
- Document oriented
- Domain Driven
- Schema less
- Restful
- Easy to scale horizontally

Elasticsearch Evaluating IR

Evaluating Information retrieval

tp: count of true positives
fp: count of false positives
fn: count of false negatives

Precision

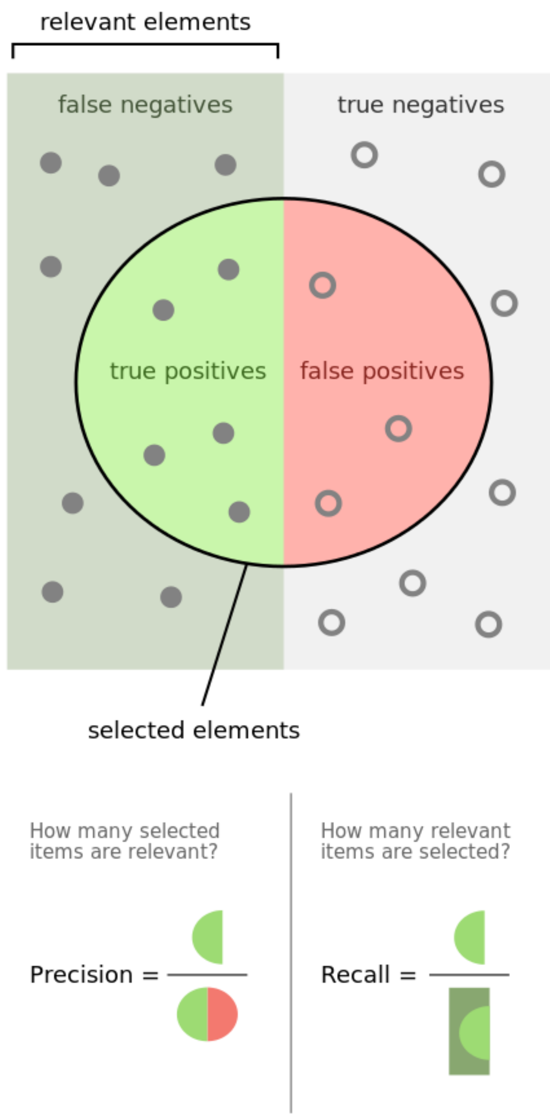
$$P = \frac{tp}{tp+fp}$$

Recall

$$R = \frac{tp}{tp+fn}$$

F measure

$$F = 2 * \frac{P*R}{P+R}$$



Source : wikipedia

Elasticsearch F-measure example

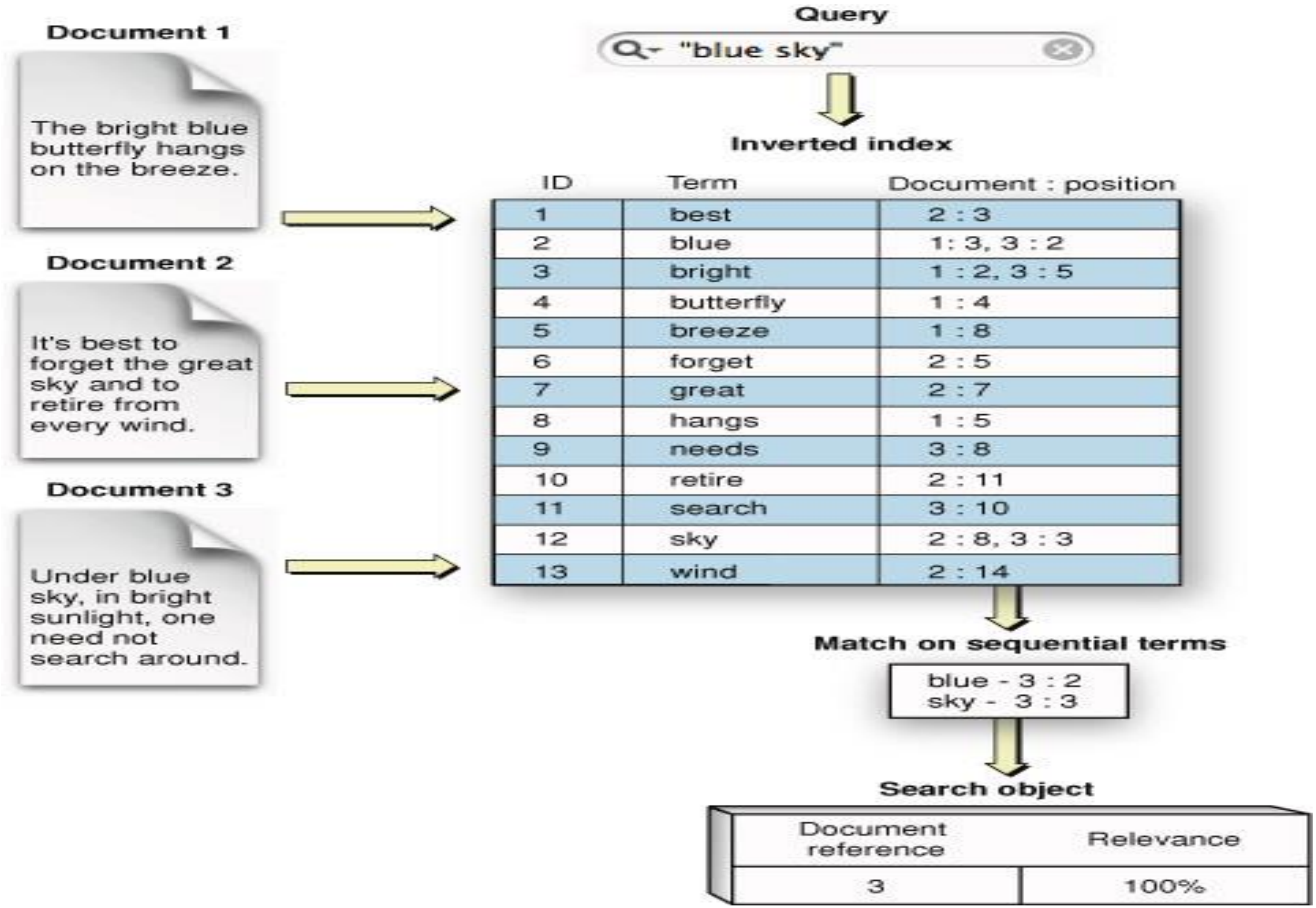
	relevant	**not relevant**	
retrieved	20	40	60
not retrieved	60	1,000,000	1,000,060
	80	1,000,040	1,000,120

$P = 20/(20 + 40) = 1/3$

$R = 20/(20 + 60) = 1/4$

$$F_1 = 2\frac{1}{\frac{1}{\frac{1}{3}}+\frac{1}{\frac{1}{4}}} = 2/7$$

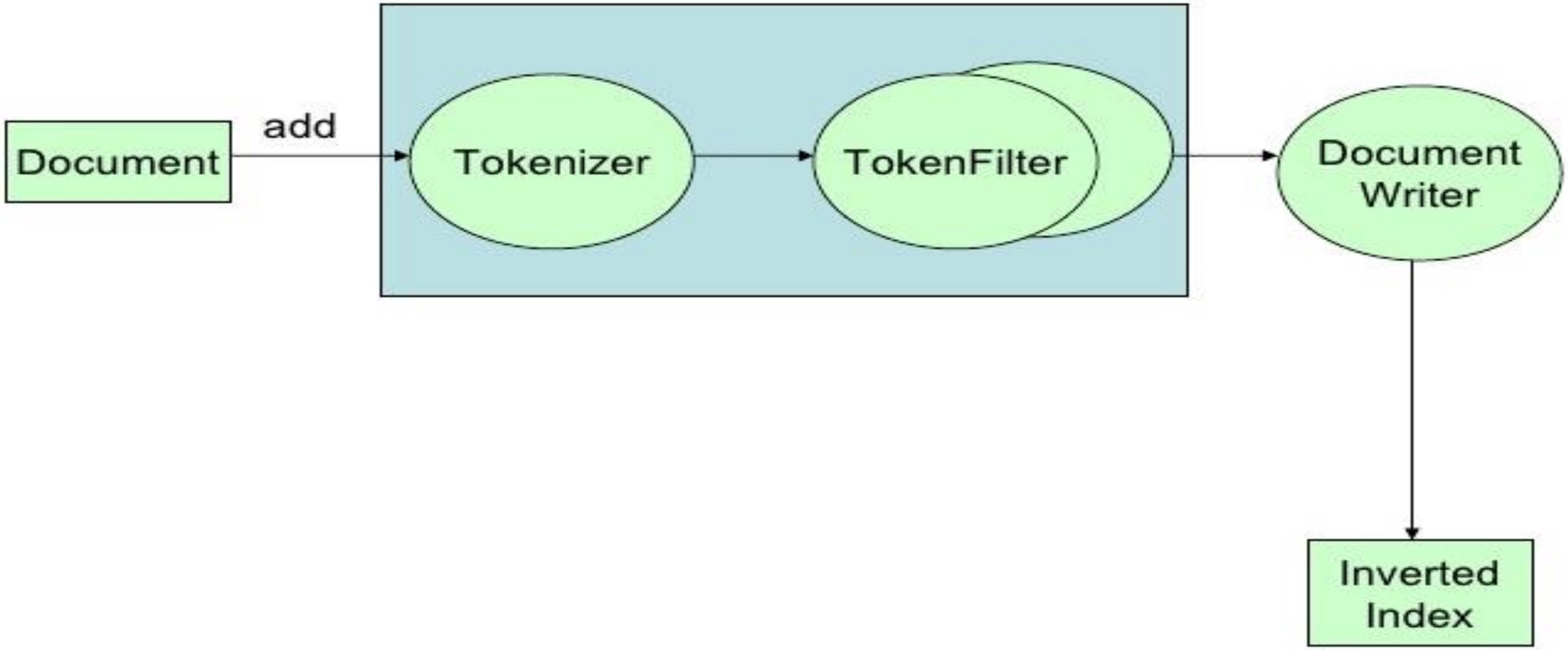
Elasticsearch Inverted Index



https://developer.apple.com/library/mac/documentation/userexperience/conceptual/SearchKitConcepts/searchKit_basics/searchKit_basics.html

Indexing Pipeline

- Analyzer : create tokens using a Tokenizer and/or applying Filters (Token Filters)
-



Credit : http://www.slideshare.net/otisg/lucene-introduction

Analysis Process - Tokenizer

WhitespaceAnalyzer

Simplest built-in analyzer

The quick brown fox jumps over the lazy dog.

[The] [quick] [brown] [fox] [jumps] [over] [the] [lazy] [dog.]

Tokens

Analysis Process - Tokenizer

SimpleAnalyzer

Lowercases, split at non-letter boundaries

The quick brown fox jumps over the lazy dog.

⇩

[the] [quick] [brown] [fox] [jumps] [over] [the] [lazy] [dog]

Tokens

Term weighting – TF.iDF

- How important is a term *t* in a document *d*
- Intuition 1: More times a term is present in a document, more important it is
  - Term frequency (tf)
- Intuition 2: If a term is present in many documents, it is less important particularly to any one of them
  - Document frequency (df)
- Combining the two: tf.idf (term frequency × inverse document frequency)
- There are various ways to calculate the exact values of both statistics.

Term weighting – TF.iDF

The tf–idf is the product of two statistics, *term frequency* and *inverse document frequency*.

- **term frequency** tf(t,d)
  - If we denote the raw count of term t in document d by $f_{t,d}$, then the simplest tf scheme is tf(t,d) = $f_{t,d}$.
  - We can normalize this value with :

$$\mathrm{tf}(t,d) = \log(1 + f_{t,d})$$

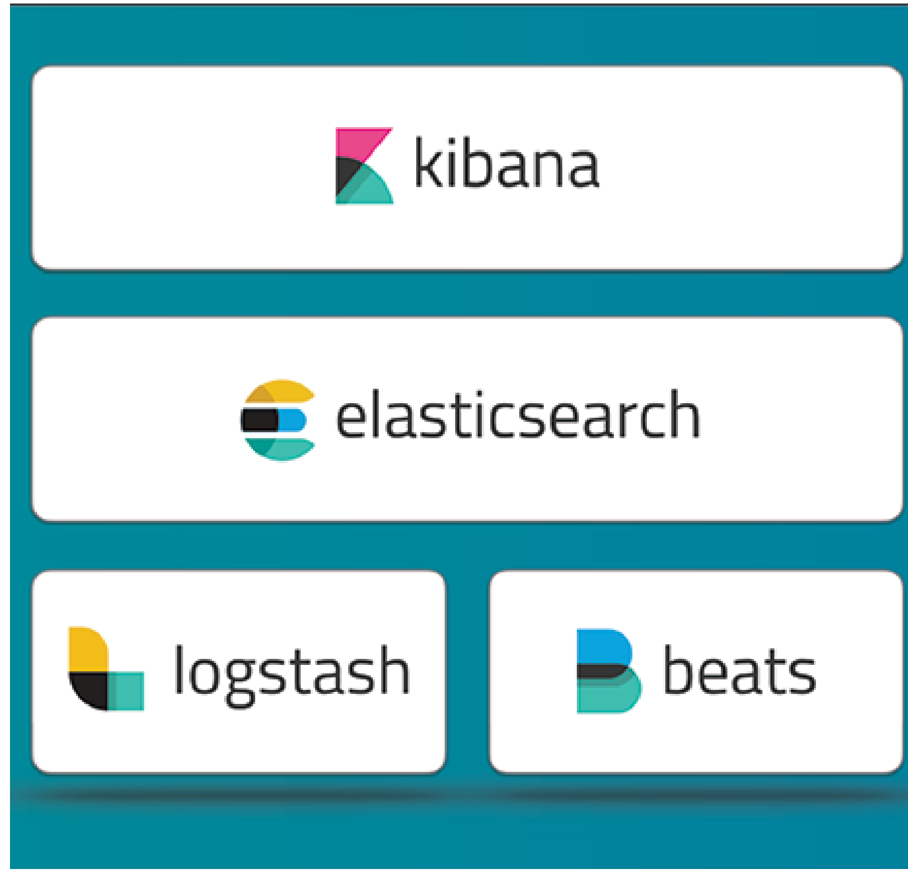
- **inverse document frequency** idf(t,D)

$$Idf(t,D)=log\,\frac{|D|}{|\{d\in D; t\in d\}|} \quad \text{or} \quad Idf(t,D)=log\,\frac{|D|}{1+|\{d\in D; t\in d\}|}$$

  - Where
    - D is the collection of documents

The ELK stack

- ELK stack=
  - **E**lasticsearch
  - **L**ogstash
  - **K**ibana
- 3 separate pieces of software
  - But they are designed to fit together
- URL: www.elastic.co



Elasticsearch Basic Concepts

Cluster

consists of one or more nodes

Index

is like a 'database' in a relational database

Node

is a running instance of elasticsearch

Type

is like a 'table' in a relational database

Document

is like a row in a table in a relational database

Field

is similar to a column in a table in RDB

Elasticsearch queries

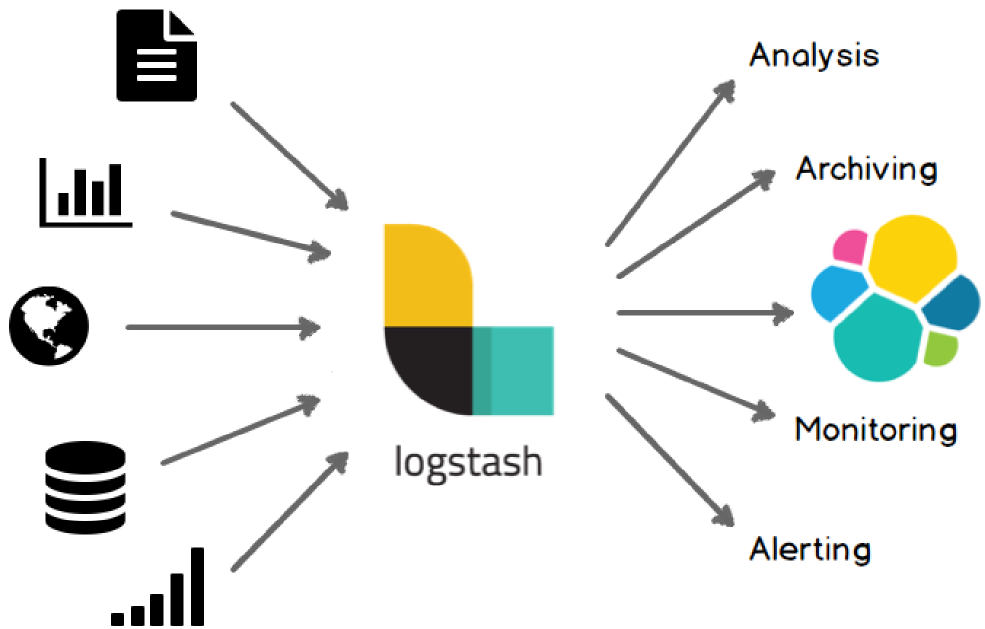
```
PUT /customer/_doc/1
        { "name": "John Doe" }

GET /customer/_doc/1
```


```
{
"_index" : "customer",
"_type" : "_doc",
"_id" : "1",
"_version" : 1,
"_seq_no" : 26,
"_primary_term" : 4,
"found" : true,
"_source" : { "name": "John Doe" }
}
```

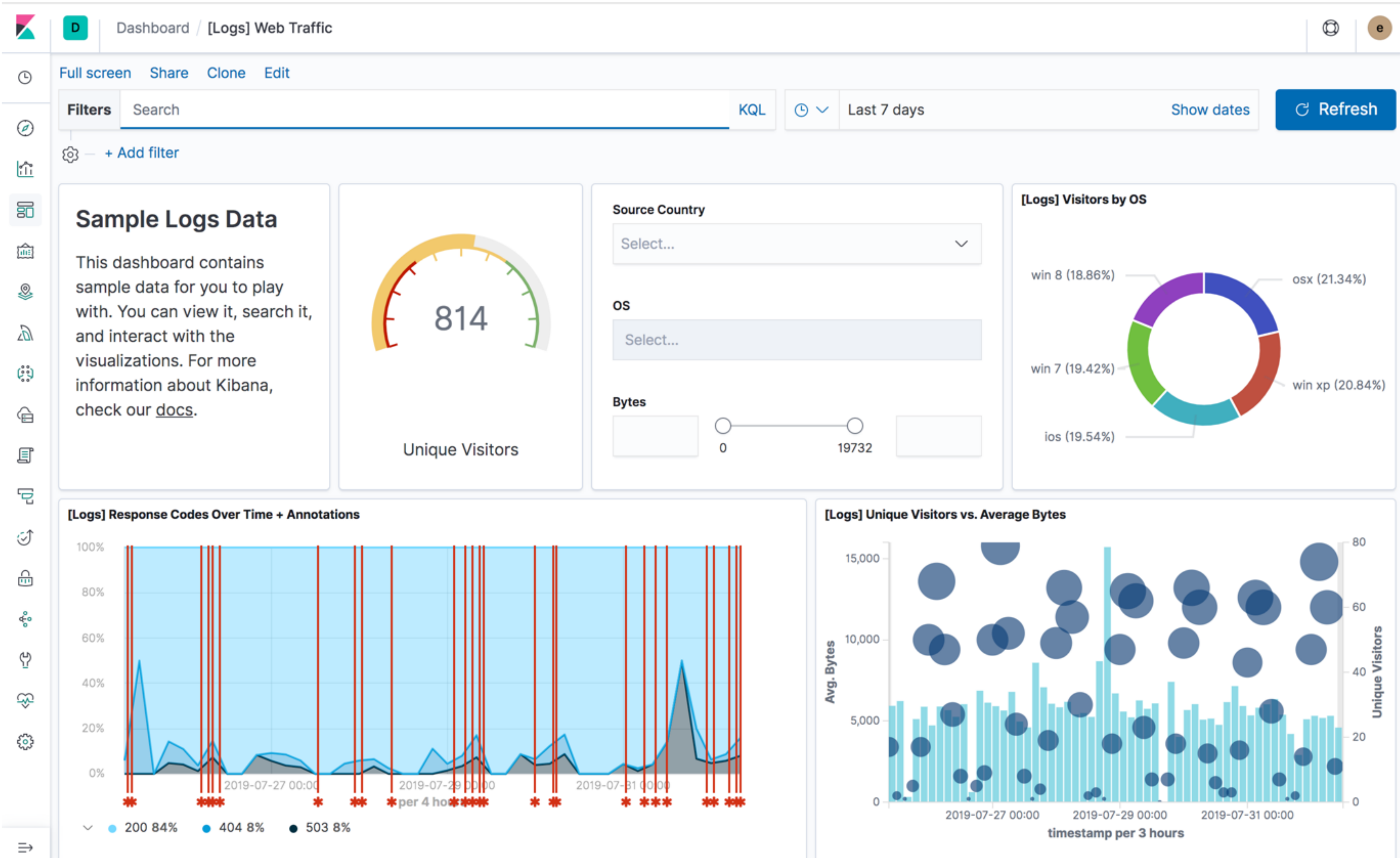
Logstash

Very simple to handle ETL tool



https://www.elastic.co/guide/en/logstash/current/introduction.html

Kibana



https://www.elastic.co/guide/en/kibana/current/dashboard.html

Plan

- Introduction
- XML Core
- XML galaxy

NOSQL

Introduction

Basic concepts

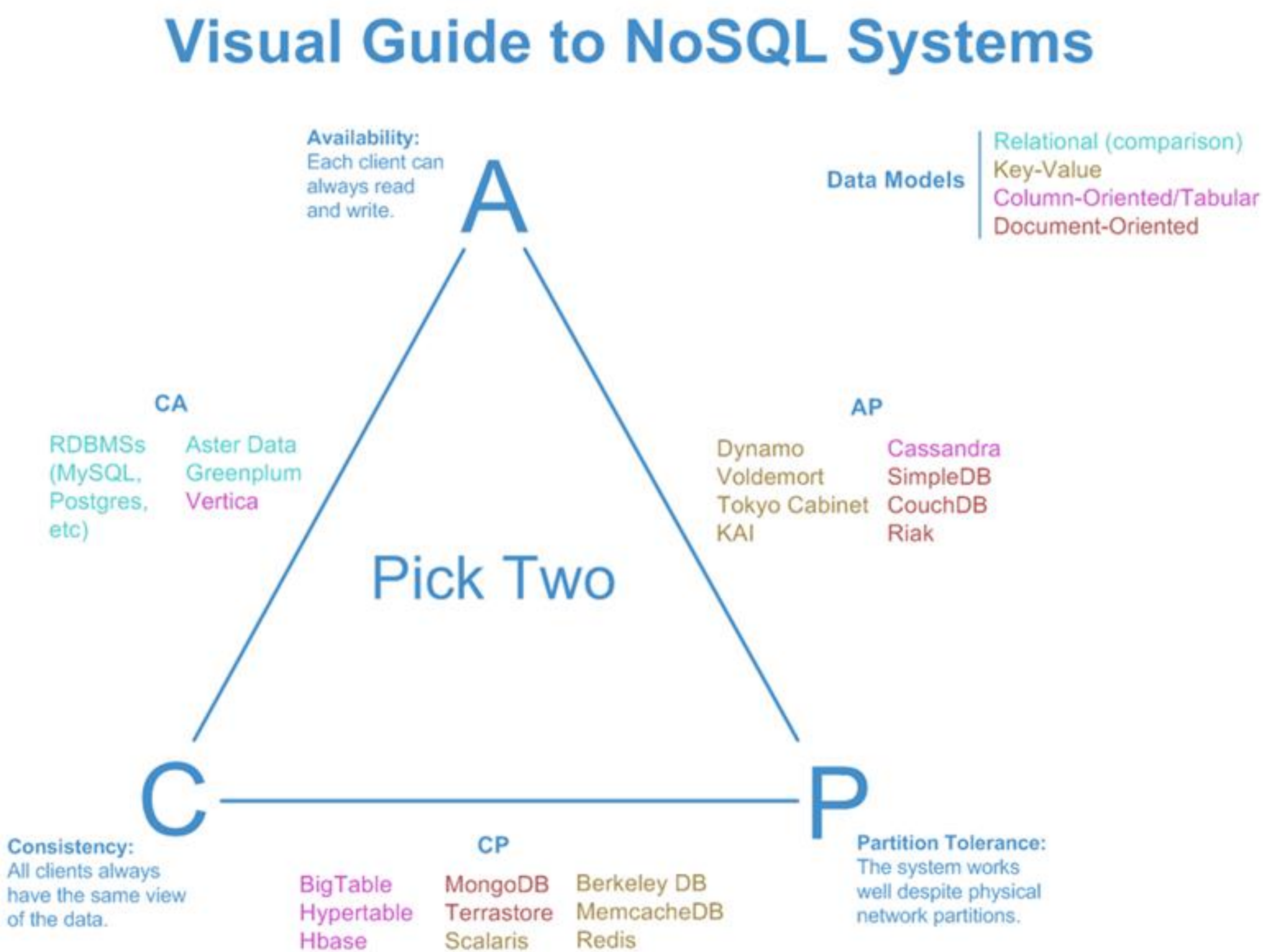
Column store

Key-Value Store

Graph DBMS

Document Store

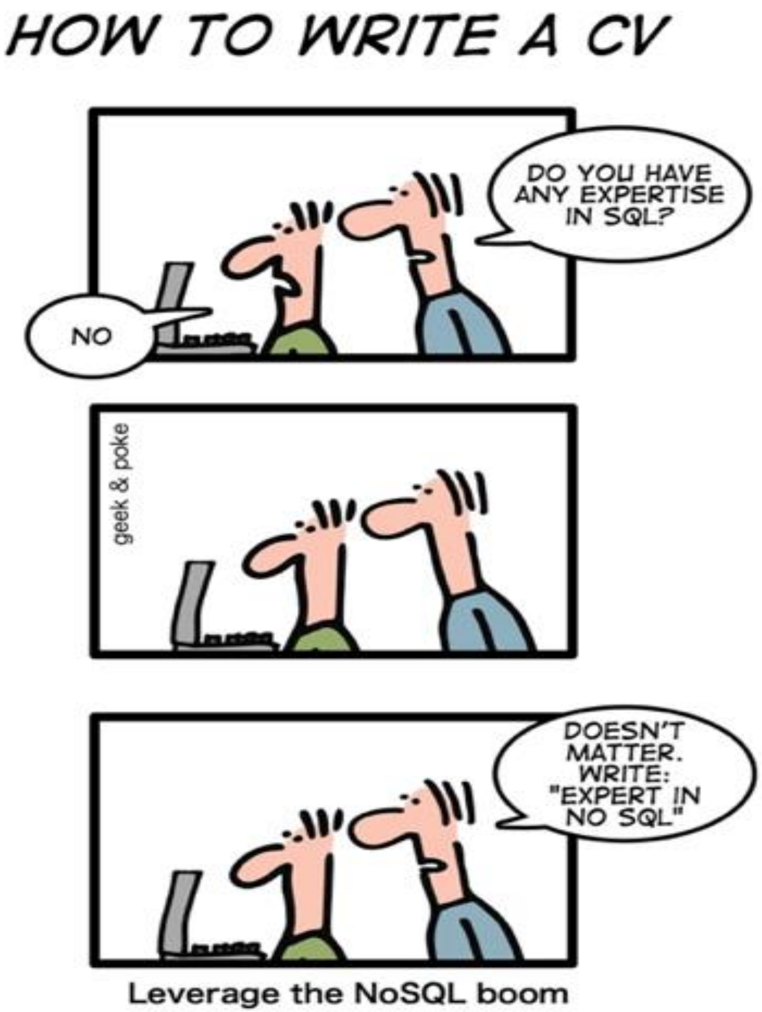
- Conclusion



http://www.samuel-berthe.fr/blog/couchbase-server-101-it-rocks

Conclusion

Use relational database when possible!



http://geekandpoke.typepad.com/geekandpoke/2011/01/nosql.html

References

- http://nosql.developpez.com/
- http: //news.humancoders.com/t/nosql/
- http://nosql-database.org/