Beyond relational databases

Daniele Apiletti

http://ooros.com http://ict-mplane.eu http://ict-ontic.eu/

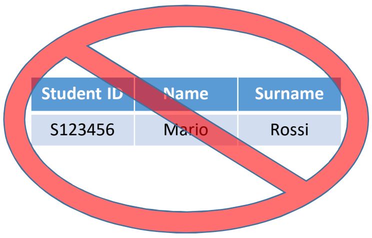
«NoSQL» birth

Not Only SQL

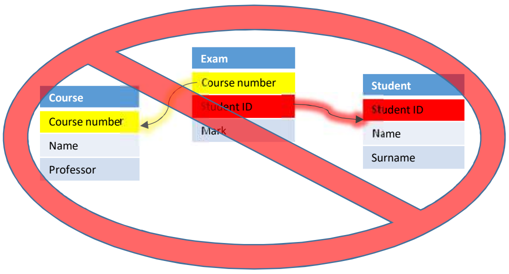
- In **1998** Carlo Strozzi's lightweight, open-source relational database that did not expose the standard SQL interface
- In **2009** Johan Oskarsson's (Last.fm) organizes an event to discuss recent advances on non-relational databases. A new, unique, short **hashtag** to promote the event on Twitter was needed.. **#NoSQL**

NoSQL main features

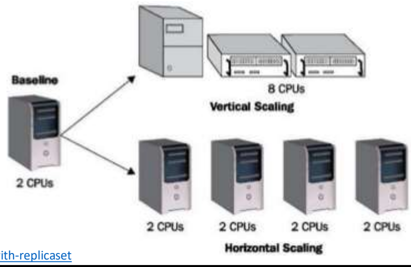
schema-less
(no tables, implicit schema)



no joins



horizontal scalability



http://www.slideshare.net/vivekparihar1/mongodb-scalability-and-high-availability-with-replicaset

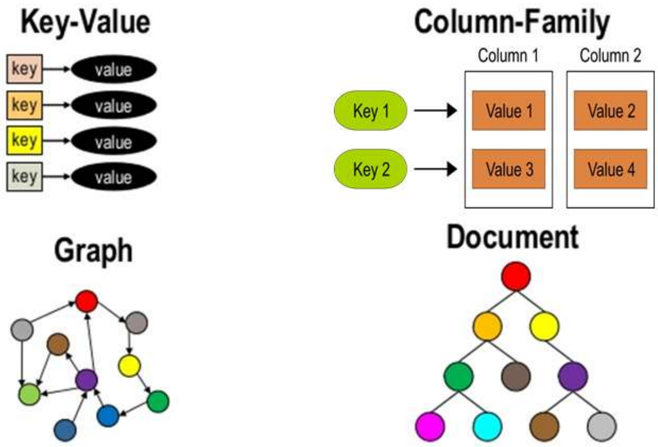
Comparison

Relational databases	Non-Relational databases
Table-based, each record is a structured row	**Specialized storage solutions**, e.g, document-based, key-value pairs, graph databases, columnar storage
Predefined **schema** for each table, changes allowed but usually blocking (expensive in distributed and live environments)	**Schema-less**, schema-free, schema change is dynamic for each document, suitable for semi-structured or **un-structured data**
Vertically scalable, i.e., typically scaled by increasing the power of the hardware	**Horizontally** scalable, NoSQL databases are scaled by increasing the databases servers in the pool of resources to reduce the load
Use **SQL** (Structured Query Language) for defining and manipulating the data, very powerful	**Custom query** languages, focused on collection of documents, graphs, and other specialized data structures

Comparison

Relational databases	Non-Relational databases
Suitable for **complex queries**, based on data **joins**	**No standard** interfaces to perform complex queries, **no joins**
Suitable for **flat** and structured data storage	Suitable for complex (e.g., **hierarchical**) data, similar to JSON and XML
Examples: MySql, **Oracle**, Sqlite, Postgres and Microsoft SQL Server	Examples: **MongoDB**, BigTable, Redis, Cassandra, Hbase and CouchDB

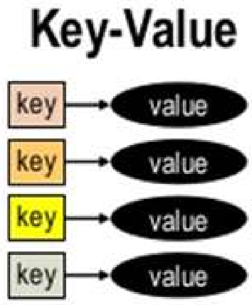
Types of NoSQL databases



http://www.slideshare.net/Couchbase/webinar-making-sense-of-nosql-applying-nonrelational-databases-to-business-needs

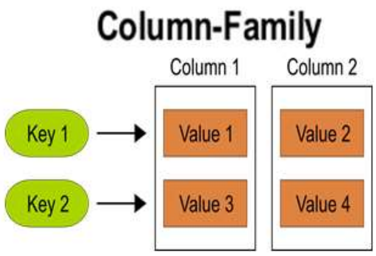
Key-values databases

- **Simplest** NoSQL data stores
- Match keys with values
- No structure
- Great **performance**
- Easily scaled
- Very fast
- Examples: Redis, Riak, **Memcached**



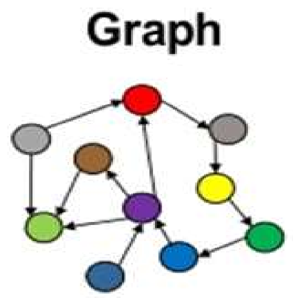
Column-oriented databases

- Store data in **columnar** format
- A column is a (possibly-complex) **attribute**
- Key-value pairs stored and retrieved on key in a parallel system (similar to **indexes**)
- **Rows** can be constructed from column values
- Column stores can produce row output (**tables**)
- Completely transparent to application
- Examples: Cassandra, Hbase, Hypertable, Amazon DynamoDB



Graph databases

- Based on graph theory
- Made up by **Vertex** and **Edges**
- Used to store information about **networks**
- Good fit for several real world applications
- Examples: Neo4J, Infinite Graph, OrientDB



Document databases

- Database stores and retrieves documents
- Keys are mapped to documents
- Documents are self-describing (**attribute=value**)
- Has hierarchical-tree nested data structures (e.g., maps, **lists**, datetime, …)
- **Heterogeneous** nature of documents
- Examples: **MongoDB**, CouchDB, RavenDB.

NoSQL example

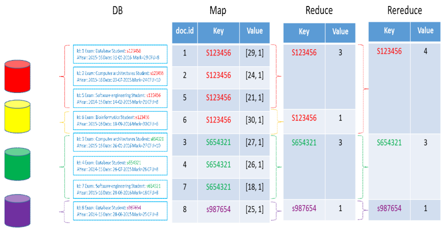
CouchDB

Cluster **O**f **U**nreliable
Commodity **H**ardware

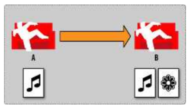
CouchDB
relax

CouchDB original home page

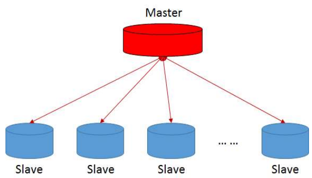
Document-oriented database can be queried and indexed in a **MapReduce** fashion



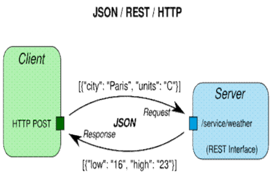
Offers incremental **replication** with bi-directional **conflict** detection and resolution



Written in Erlang, a robust functional programming language ideal for building **concurrent distributed systems**. Erlang allows for a flexible design that is **easily scalable** and readily extensible

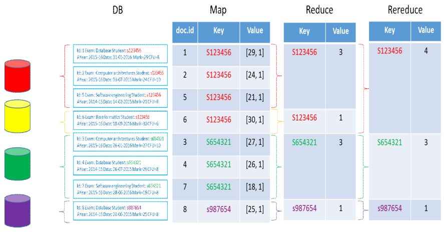


Provides a **RESTful JSON API** than can be accessed from any enviroment that allows **HTTP** requests

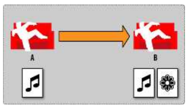


CouchDB original home page

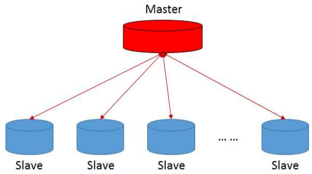
Document-oriented database can be queried and indexed in a **MapReduce** fashion



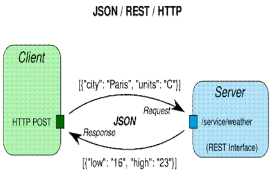
Offers incremental **replication** with bi-directional **conflict** detection and resolution



Written in Erlang, a robust functional programming language ideal for building **concurrent distributed systems**. Erlang allows for a flexible design that is **easily scalable** and readily extensible



Provides a **RESTful JSON API** than can be accessed from any enviroment that allows **HTTP** requests



MapReduce

a **scalable** distributed programming model to **process** Big Data

MapReduce

- Published in **2004** by **Google**
  - J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004
  - used to rewrite the production indexing system with 24 MapReduce operations (in August 2004 alone, 3288 TeraBytes read, 80k machine-days used, jobs of 10' avg)
- **Distributed** programming model
- Process large data sets with parallel algorithms on a **cluster** of common machines, e.g., PCs
- Great for **parallel** jobs requiring pieces of computations to be executed on all data records
- Move the computation (algorithm) to the data (remote node, PC, disk)
- Inspired by the map and reduce functions used in funcional programming

MapReduce: working principles

- Consists of two functions, a **Map** and a **Reduce**
  - The Reduce is optional
- **Map** function
  - Process each record (document)
  - Return a list of key-value pairs
- **Reduce** function
  - reduce the list of key-values returned by the map to a single value (it can be a complex value such as a map)

Map

- Map function are called once with each document as the argument:
  function(doc) {emit(key, value)}
- The function can choose to skip the document altogether or emit one or **more rows** as key/value pairs
- Map function may **not** depend on any information **outside the document.** This independence is what allows CouchDB views to be generated incrementally and **in parallel**

Map example

- Example database, a collection of docs describing university exam records

Id: 1 Exam: Database Student: s123456 AYear: 2015-16 Date: 31-01-2016 Mark=29 CFU=8	Id: 2 Exam: Computer architectures Student: s123456 AYear: 2015-16 Date: 03-07-2015 Mark=24 CFU=10	Id: 3 Exam: Computer architectures Student: s654321 AYear: 2015-16 Date: 26-01-2016 Mark=27 CFU=10	Id: 4 Exam: Database Student: s654321 AYear: 2014-15 Date: 26-07-2015 Mark=26 CFU=8
Id: 5 Exam: Software engineering Student: s123456 AYear: 2014-15 Date: 14-02-2015 Mark=21 CFU=8	Id: 6 Exam: Bioinformatics Student: s123456 AYear: 2015-16 Date: 18-09-2016 Mark=30 CFU=6	Id: 7 Exam: Software engineering Student: s654321 AYear: 2015-16 Date: 28-06-2016 Mark=18 CFU=8	Id: 8 Exam: Database Student: s987654 AYear: 2014-15 Date: 28-06-2015 Mark=25 CFU=8

Map example (1)

- List of exams and corresponding mark

```
Function(doc){
    emit(doc.exam, doc.mark);
}
```

Id: 2
Exam: Computer architectures
Student: s123456
AYear: 2015-16
Date: 03-07-2015
Mark=24
CFU=10

Id: 3
Exam: Computer architectures
Student: s654321
AYear: 2015-16
Date: 26-01-2016
Mark=27
CFU=10

Id: 4
Exam: Database
Student: s654321
AYear: 2014-15
Date: 26-07-2015
Mark=26
CFU=8

Id: 1
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

Id: 5
Exam: Software engineering
Student: s123456
AYear: 2014-15
Date: 14-02-2015
Mark=21
CFU=8

Id: 8
Exam: Database
Student: s987654
AYear: 2014-15
Date: 28-06-2015
Mark=25
CFU=8

Id: 7
Exam: Software engineering
Student: s654321
AYear: 2015-16
Date: 28-06-2016
Mark=18
CFU=8

Id: 6
Exam: Bioinformatics
Student: s123456
AYear: 2015-16
Date: 18-09-2016
Mark=30
CFU=6

Result:

doc.id	Key	Value
6	Bioinformatics	30
2	Computer architectures	24
3	Computer architectures	27
1	Database	29
4	Database	26
8	Database	25
5	Software engineering	21
7	Software engineering	18

Map example (2)

- Ordered list of exams, academic year, and date, and select their mark

```
Function(doc) {
    key = [doc.exam, doc.AYear]
    value = doc.mark
    emit(key, value);
}
```

Id: 2
Exam: Computer architectures
Student: s123456
AYear: 2015-16
Date: 03-07-2015
Mark=24
CFU=10

Id: 3
Exam: Computer architectures
Student: s654321
AYear: 2015-16
Date: 26-01-2016
Mark=27
CFU=10

Id: 4
Exam: Database
Student: s654321
AYear: 2014-15
Date: 26-07-2015
Mark=26
CFU=8

Id: 1
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

Id: 5
Exam: Software engineering
Student: s123456
AYear: 2014-15
Date: 14-02-2015
Mark=21
CFU=8

Id: 8
Exam: Database
Student: s987654
AYear: 2014-15
Date: 28-06-2015
Mark=25
CFU=8

Id: 7
Exam: Software engineering
Student: s654321
AYear: 2015-16
Date: 28-06-2016
Mark=18
CFU=8

Id: 6
Exam: Bioinformatics
Student: s123456
AYear: 2015-16
Date: 18-09-2016
Mark=30
CFU=6

Result:

doc.id	Key	Value
6	Bioinformatics, 2015-16	30
2	Computer architectures, 2015-16	24
3	Computer architectures, 2015-16	27
4	Database, 2014-15	26
8	Database, 2014-15	25
1	Database, 2015-16	29
5	Software engineering, 2014-15	21
7	Software engineering, 2015-16	18

Map example (3)

- Ordered list of students, with mark and CFU for each exam

```
Function(doc) {
    key = doc.student
    value = [doc.mark, doc.CFU]
    emit(key, value);
}
```

Id: 2
Exam: Computer architectures
Student: s123456
AYear: 2015-16
Date: 03-07-2015
Mark=24
CFU=10

Id: 3
Exam: Computer architectures
Student: s654321
AYear: 2015-16
Date: 26-01-2016
Mark=27
CFU=10

Id: 4
Exam: Database
Student: s654321
AYear: 2014-15
Date: 26-07-2015
Mark=26
CFU=8

Id: 1
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

Id: 5
Exam: Software engineering
Student: s123456
AYear: 2014-15
Date: 14-02-2015
Mark=21
CFU=8

Id: 8
Exam: Database
Student: s987654
AYear: 2014-15
Date: 28-06-2015
Mark=25
CFU=8

Id: 7
Exam: Software engineering
Student: s654321
AYear: 2015-16
Date: 28-06-2016
Mark=18
CFU=8

Id: 6
Exam: Bioinformatics
Student: s123456
AYear: 2015-16
Date: 18-09-2016
Mark=30
CFU=6

Result:

doc.id	Key	Value
1	S123456	[29, 8]
2	S123456	[24, 10]
5	S123456	[21, 8]
6	S123456	[30, 6]
3	S654321	[27, 10]
4	S654321	[26, 8]
7	S654321	[18, 8]
8	s987654	[25, 8]

Reduce

- Documents emitted by the map function are **sorted by key**
  - some platforms (e.g. Hadoop) allow you to specifically define a **shuffle phase** to manage the distribution of map results to reducers spread over different nodes, thus providing a fine-grained control over **communication costs**
- Reduce **inputs** are the map outputs: a **list** of key-value documents
- Each reduce-function call **outputs** one key-value document
- The most simple SQL-equivalent operations performed by means of reducers are **«group by» aggregations**, but reducers are very flexible functions that can execute even **complex operations**
- **Re-reduce**: reduce functions can be called on their own results

MapReduce example (1)

- Map - List of exams and corresponding mark

```
Function(doc){
    emit(doc.exam, doc.mark);
}
```

- Reduce - Compute the average mark for each exam

```
Function(key, values){
    S = sum(values);
    N = len(values);
    AVG = S/N;
    return AVG;
}
```

id: 1 DOC
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

The reduce function receives:
- **key**=Bioinformatics, **values**=[30]
- ...
- **key**=Database, **values**=[29,26,25]
- ...

Map

doc.id	Key	Value
6	Bioinformatics	30
2	Computer architectures	24
3	Computer architectures	27
1	Database	29
4	Database	26
8	Database	25
5	Software engineering	21
7	Software engineering	18

Reduce

Key	Value
Bioinformatics	30
Computer architectures	25.5
Database	26.67
Software engineering	19.5

MapReduce example (2)

- Map - List of exams and corresponding mark

```
Function(doc){
    emit(
     [doc.exam, doc.AYear],
     doc.mark
    );
}
```

- Reduce - Compute the average mark for each exam and academic year

```
Function(key, values){
    S = sum(values);
    N = len(values);
    AVG = S/N;
    return AVG;
}
```

Reduce is the same as before

id: 1 DOC
Exam: Database
Student: s123456
AYear: 2015-16
Date: 31-01-2016
Mark=29
CFU=8

The reduce function receives:
- **key**=[Database, 2014-15], **values**=[26,25]
- **key**=[Database, 2015-16], **values**=[29]
- ...

Map

doc.id	Key	Value
6	Bioinformatics, 2015-16	30
2	Computer architectures, 2015-16	24
3	Computer architectures, 2015-16	27
4	Database, 2014-15	26
8	Database, 2014-15	25
1	Database, 2015-16	29
5	Software engineering, 2014-15	21
7	Software engineering, 2015-16	18

Reduce

Key	Value
[Bioinformatics, 2015-16]	30
[Computer architectures, 2015-16]	25.5
[Database, 2014-15]	25.5
[Database, 2015-16]	29
[Software engineering, 2014-15]	21
[Software engineering, 2015-16]	18

Rereduce

- Average mark the for each exam (group level=1) – **same Reduce** as before

DB

Id: 1 Exam: Database Student: s123456 AYear: 2015-16 Date: 31-01-2016 Mark=29 CFU=8	Id: 8 Exam: Database Student: s987654 AYear: 2014-15 Date: 28-06-2015 Mark=25 CFU=8
Id: 6 Exam: Bioinformatics Student: s123456 AYear: 2015-16 Date: 18-09-2016 Mark=30 CFU=6	Id: 4 Exam: Database Student: s654321 AYear: 2014-15 Date: 26-07-2015 Mark=26 CFU=8
Id: 5 Exam: Software engineering Student: s123456 AYear: 2014-15 Date: 14-02-2015 Mark=21 CFU=8	Id: 7 Exam: Software engineering Student: s654321 AYear: 2015-16 Date: 28-06-2016 Mark=18 CFU=8
Id: 3 Exam: Computer architectures Student: s654321 AYear: 2015-16 Date: 26-01-2016 Mark=27 CFU=10	Id: 2 Exam: Computer architectures Student: s123456 AYear: 2015-16 Date: 03-07-2015 Mark=24 CFU=10

Map

doc.id	Key	Value
6	Bioinformatics, 2015-16, 18-09-2016	30
2	Computer architectures, 2015-16, 03-07-2015	24
3	Computer architectures, 2015-16, 26-01-2016	27
4	Database, 2014-1015, 26-07-2015	26
8	Database, 2014-15, 28-06-2015	25
1	Database, 2015-16, 31-01-2016	29
5	Software engineering, 2014-15, 14-02-2015	21
7	Software engineering, 2015-16, 28-06-2016	18

Reduce

Key	Value
[Bioinformatics, 2015-16]	30
[Computer architectures, 2015-16]	25.5
[Database, 2014-15]	25.5
[Database, 2015-16]	29
[Software engineering, 2014-15]	21
[Software engineering, 2015-16]	18

Rereduce

Key	Value
Bioinformatics	30
Computer architectures	25.5
Database	27.25
Software engineering	19.5

MapReduce example (3a)

- Map - Ordered list of students, with mark and CFU for each exam

```
Function(doc) {
    key = doc.student
    value = [doc.mark, doc.CFU]
    emit(key, value);
}
```

- Reduce - Average CFU-weighted mark for each student

```
Function(key, values){
    S = sum([ X*Y for X,Y in values ]);
    N = sum([ Y for X,Y in values ]);
    AVG = S/N;
    return AVG;
}
```

key = S123456,
values = [(29,8), (24,10), (21,8)...]
X = 29, 24, 21, … → mark
Y = 8, 10, 8, … →CFU

The reduce function receives:
- **key**=S123456, **values**=[(29,8), (24,10), (21,8)...]
- ...
- **key**=s987654, **values**=[(25,8)]

Map

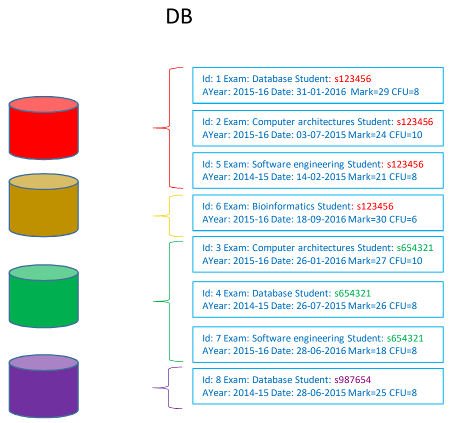
doc.id	Key	Value
1	S123456	[29, 8]
2	S123456	[24, 10]
5	S123456	[21, 8]
6	S123456	[30, 6]
3	S654321	[27, 10]
4	S654321	[26, 8]
7	S654321	[18, 8]
8	s987654	[25, 8]

Reduce

Key	Value
S123456	25.6
S654321	23.9
s987654	25

MapReduce example (3b)

- Compute the number of exams for each student
- Technological view of data distribution among different nodes



Map

doc.id	Key	Value
1	S123456	[29, 1]
2	S123456	[24, 1]
5	S123456	[21, 1]
6	S123456	[30, 1]
3	S654321	[27, 1]
4	S654321	[26, 1]
7	S654321	[18, 1]
8	s987654	[25, 1]

Reduce

Key	Value
S123456	3
S123456	1
S654321	3
s987654	1

Rereduce

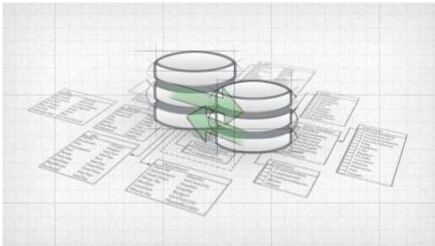
Key	Value
S123456	4
S654321	3
s987654	1

Views (indexes)

- The only way to **query** CouchDB is to build a view on the data
- A view is produced by a MapReduce
- The predefined view for each database has
  - the document ID as **key**,
  - the whole document as **value**
  - no Reduce
- CouchDB views are **materialized** as values **sorted by key**
  - allows the same DB to have **multiple primary indexes**
- When writing CouchDB map functions, your primary goal is to build an index that **stores related data under nearby keys**

Replication

Same data
in **different** places
(content and schema)

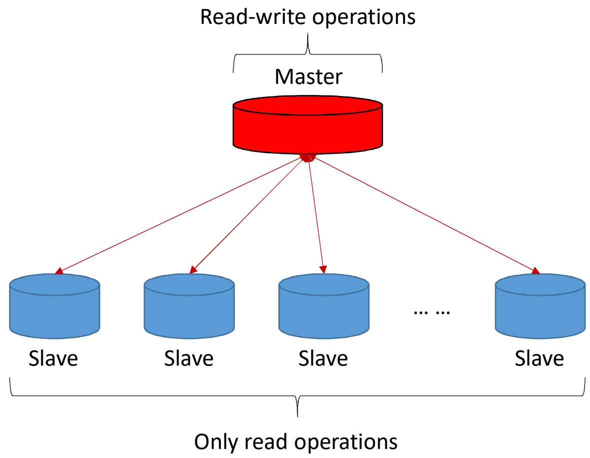


Replication

- **Same** data
  - portions of the whole dataset (chunks)
- in **different** places
  - local and/or remote servers, clusters, data centers
- Goals
  - Redundancy helps surviving failures (availability)
  - Better performance
- Approaches
  - Master-Slave replication
  - A-Synchronous replication

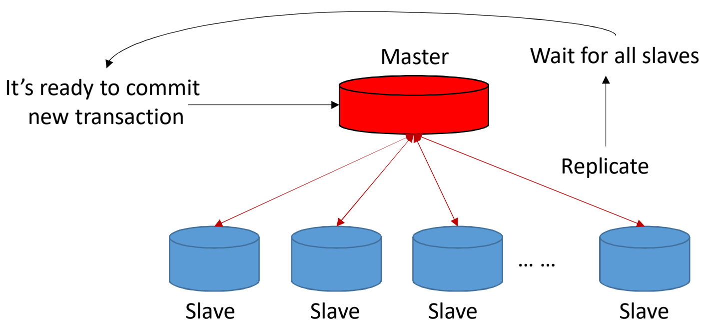
Master-Slave replication

- Master-Slave
  - A **master** server takes all the writes, updates, inserts
  - One or more **Slave** servers take all the reads (they can't write)
  - Only read **scalability**
  - The master is a single point of **failure**
- CouchDB supports Master-Master replication



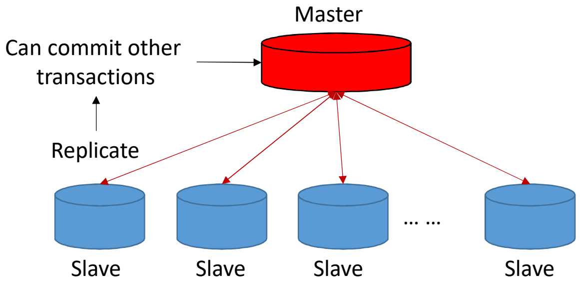
Synchronous replication

- Before committing a transaction, the Master **waits** for (all) the Slaves to commit
- Similar in concept to the **2-Phase Commit** in relational databases
- **Performance** killer, in particular for replication in the cloud
- Trade-off: wait for a subset of Slaves to commit, e.g., the **majority** of them



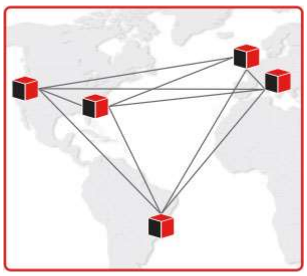
Asynchronous replication

- The Master commits **locally**, it does not wait for any Slave
- Each Slave independently fetches updates from Master, which may **fail**…
  - IF no Slave has replicated, then you've **lost the data** committed to the Master
  - IF some Slaves have replicated and some haven't, then you have to **reconcile**
- Faster and **un**reliable



Distributed databases

Different autonomous machines, working **together** to manage the same **dataset**

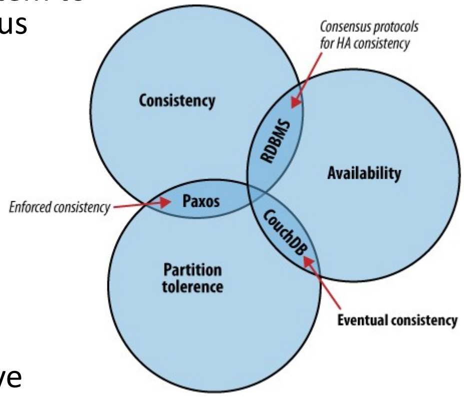


Key features of distributed databases

- There are 3 typical problems in distributed databases:
  - **Consistency**
    - All the distributed databases provide the same data to the application
  - **Availability**
    - Database failures (e.g., master node) do not prevent survivors from continuing to operate
  - **Partition** tolerance
    - The system continues to operate despite arbitrary message loss, when connectivity failures cause network partitions

CAP Theorem

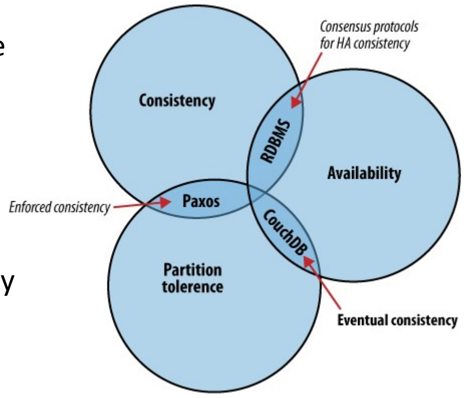
- The CAP theorem, also known as Brewer's theorem, states that it is impossible for a distributed system to **simultaneously** provide **all three** of the previous guarantees
- The theorem began as a **conjecture** made by University of California in 1999-2000
  - Armando Fox and Eric Brewer, "Harvest, Yield and Scalable Tolerant Systems", Proc. 7th Workshop Hot Topics in Operating Systems (HotOS 99), IEEE CS, 1999, pg. 174-178.
- In 2002 a formal proof was published, establishing it as a **theorem**
  - Seth Gilbert and Nancy Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59
- In 2012, a follow-up by Eric Brewer, "CAP twelve years later: How the "rules" have changed"
  - IEEE Explore, Volume 45, Issue 2 (2012), pg. 23-29.



http://guide.couchdb.org/editions/1/en/consistency.html#figure/1

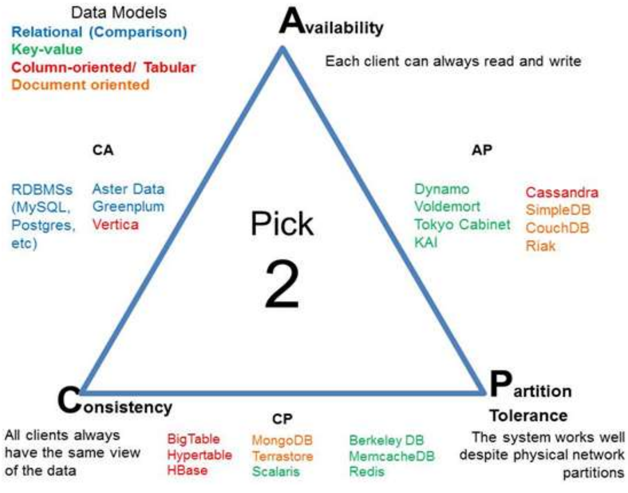
CAP Theorem

- The easiest way to understand CAP is to think of **two nodes** on opposite sides of a **partition**.
- Allowing at least one node to update state will cause the nodes to become **inconsistent**, thus forfeiting C.
- If the choice is to preserve consistency, one side of the partition must act as if it is **unavailable**, thus forfeiting A.
- Only when no network **partition** exists, is it possible to preserve both consistency and availability, thereby forfeiting P.
- The general belief is that for wide-area systems, **designers cannot forfeit P** and therefore have a difficult choice between C and A.



http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

CAP Theorem



http://blog.flux7.com/blogs/nosql/cap-theorem-why-does-it-matter

CA without P (local consistency)

- **Partitioning** (communication breakdown) causes a failure.
- We can still have **Consistency** and **Availability** of the data shared by agents **within each Partition**, by ignoring other partitions.
  - Local rather than global consistency / availability
- Local consistency for a partial system, 100% availability for the partial system, and no partitioning does not exclude several partitions from existing with their own “internal” CA.
- So partitioning means having **multiple independent systems** with 100% CA that do not need to interact.

CP without A (transaction locking)

- A system is allowed to *not* answer requests at all (turn off “A”).
- We claim to tolerate **partitioning/faults**, because we simply block all responses if a partition occurs, assuming that we cannot continue to function correctly without the data on the other side of a partition.
- Once the partition is healed and **consistency** can once again be verified, we can restore availability and leave this mode.
- In this configuration there are global consistency, and global correct behaviour in partitioning is to **block access to replica sets** that are not in synch.
- In order to tolerate P at any time, we must sacrifice A at any time for global consistency.
- This is basically the **transaction lock**.

AP without C (best effort)

- If we don't care about **global consistency** (i.e. simultaneity), then every part of the system can make available what it knows.
- Each part might be able to answer someone, even though the system as a whole has been broken up into incommunicable regions (**partitions**).
- In this configuration without consistency means without the assurance of global consistency **at all times**.

A consequence of CAP

"Each node in a system should be able to make decisions purely based on **local state**. If you need to do something under high load with **failures** occurring and you need to reach agreement, you're lost. If you're concerned about **scalability**, any algorithm that forces you to run agreement will eventually become your **bottleneck**. Take that as a given."

Werner Vogels, Amazon CTO and Vice President

Beyond CAP

- The "2 of 3" view is misleading on several fronts.
- First, because **partitions** are rare, there is little reason to forfeit C or A when the system is not partitioned.
- Second, the **choice between C and A** can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved.
- Finally, all three **properties are more continuous than binary**. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.

ACID versus BASE

- ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum
- ACID properties focus on **consistency** and are the traditional approach of databases
- BASE properties focus on high **availability** and to make explicit both the choice and the spectrum
- **BASE**: Basically Available, Soft state, Eventually consistent, work well in the presence of **partitions** and thus promote **availability**

ACID

- The four ACID properties are:
  - **Atomicity (A)** All systems benefit from atomic operations, the database transaction must completely succeed or fail, partial success is not allowed
  - **Consistency (C)** During the database transaction, the database progresses from a valid state to another. In ACID, the C means that a transaction pre-serves all the database rules, such as unique keys. In contrast, the C in CAP refers only to single copy consistency.
  - **Isolation (I)** Isolation is at the core of the CAP theorem: if the system requires ACID isolation, it can operate on at most one side during a partition, because a client's transaction must be isolated from other client's transaction
  - **Durability (D)** The results of applying a transaction are permanent, it must persist after the transaction completes, even in the presence of failures.

BASE

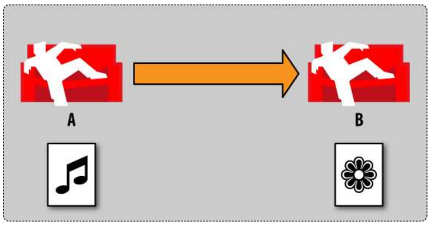
- **Basically Available**: the system provides availability, in terms of the CAP theorem
- **Soft state:** indicates that the state of the system may change over time, even without input, because of the eventual consistency model.
- **Eventual consistency:** indicates that the system will become consistent over time, given that the system doesn't receive input during that time
- Example: DNS – Domain Name Servers
  - DNS is not multi-master

How the rules have changed

- Any networked shared-data system can have **only 2 of 3** desirable properties at the **same time**
- Explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some **trade-off of all three**
- CAP prohibits only a tiny part of the design space:
  - **perfect** availability (A) and consistency (C)
  - in the presence of partitions (P), which are **rare**
- Although designers need to choose between consistency and availability when partitions are present, there is an incredible range of **flexibility for handling partitions** and recovering from them
- Modern CAP goal should be to maximize combinations of consistency (C) and availability (A) that make sense for the **specific application**

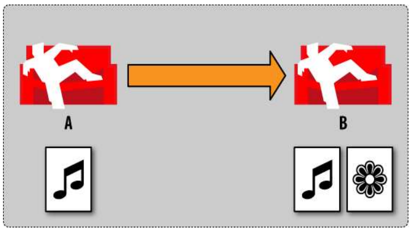
Conflict resolution problem

- There are two customers, A and B
- A books a hotel room, the last avaible room
- B does the same, on a different node of the system, which was **not consistent**



Conflict resolution problem

- The hotel room document is affected by two **conflicting updates**
- Applications should solve the conflict with custom logic (it's a business decision)
- The database can
  - **Detect** the conflict
  - Provide a local **solution**, e.g., latest version is saved as the winning version



Conflict

- CouchDB guarantees that **each instance** that sees the **same conflict** comes up with the **same winning** and losing **revisions**.
- It does so by running a **deterministic algorithm** to pick the winner.
  - The revision with the longest revision history list becomes the winning revision.
  - If they are the same, the _rev values are compared in ASCII sort order, and the highest wins.

HTTP API

a «**web**» database,
no ad-hoc **client**
required

HTTP RESTful API

- How to **get** a document? Use your browser and write its **URL**
  - http://localhost:5984/test/some_doc_id
- Any application and language can access **web data**
  - GET /somedatabase/some_doc_id HTTP/1.0
  - HEAD /somedatabase/some_doc_id HTTP/1.0
    - HTTP/1.1 200 OK
- **Write** a document by means of PUT HTTP request (specify revision to avoid conflicts)
  - PUT /somedatabase/some_doc_id HTTP/1.0
    - HTTP/1.1 201 Created
    - HTTP/1.1 409 Conflict

HTTP RESTful API

- **Delete** a document
  - DELETE /somedatabase/**some_doc_id**?**rev**=1582603387
    - HTTP/1.1 200 OK
- Parametric multi-document fetch
  - GET /somedatabase/**_alldocs**?**startkey**=doc2&**endkey**=doc3
  - GET /somedatabase/**_alldocs**?**startkey**=doc2&**limit**=2&**descending**=true..
- All that were updated and deleted,
  in the order these actions were executed (LOG)
  - GET /somedatabase/**_all_docs_by_seq**

MongoDB

The **leading** NoSQL database currently on the market

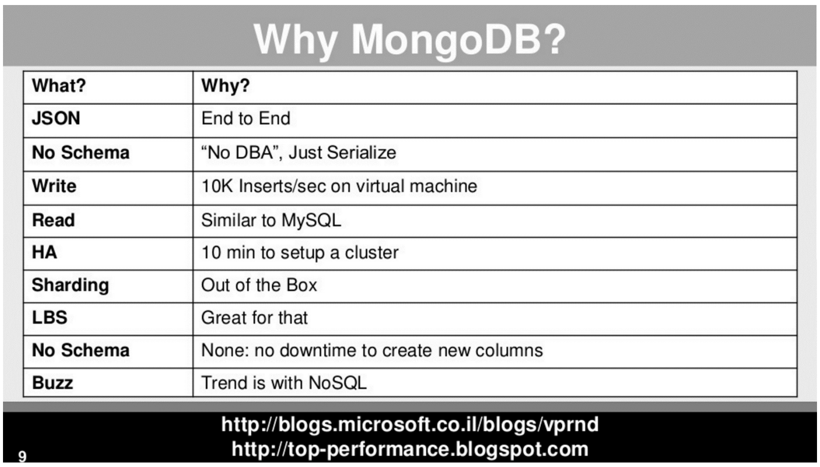
mongoDB

MongoDB - intro

- Full of **features**, beyond NoSQL
- High **performance** and natively **scalable**
- Open source
- 311$ millions in funding
- 500+ employees
- **2000+ customers**

http://www.slideshare.net/mongodb/introduction-to-mongodb-56807822

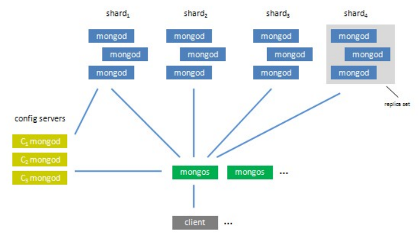
MongoDB - why



Why MongoDB?

What?	Why?
JSON	End to End
No Schema	"No DBA", Just Serialize
Write	10K Inserts/sec on virtual machine
Read	Similar to MySQL
HA	10 min to setup a cluster
Sharding	Out of the Box
LBS	Great for that
No Schema	None: no downtime to create new columns
Buzz	Trend is with NoSQL

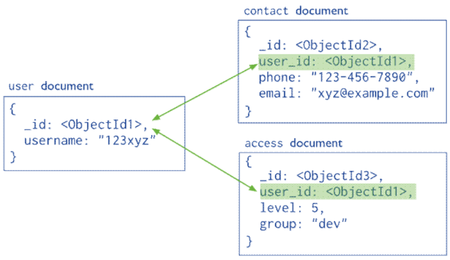
http://blogs.microsoft.co.il/blogs/vprnd
http://top-performance.blogspot.com
9



MongoDB – Document Data Design

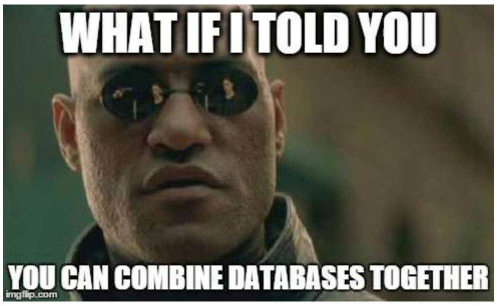
- High-level, business-ready representation of the data
- Flexible and rich, adapting to most use cases
- Mapping into developer-language objects
  - year, month, day, timestamp,
  - lists, sub-documents, etc.

- BUT
- Relations among documents / records are inefficient, and leads to de-normalization
  - Object(ID) reference, with **no native join**
- Temptation to go too much schema-free / non-relational even with structured relational data



«So, which database should I choose?»

- If you're building an app today, then there might be a need for **using two or more databases** at the same time
- If your app does (text) search you might have to implement **ElasticSearch**
- for non-relational data-storage, **MongoDB** works the best
- if you're building an IoT which has sensors pumping out a ton of data, shoot it into **Cassandra**
- Implementing multiple databases to build one app is called "**Polyglot Persistence**"



https://blog.cloudboost.io/why-you-should-not-use-only-mongodb/

A design recipe

A notable example of NoSQL design

Design recipe: banking account

- Banks are serious business
- They need serious databases to store serious transactions and serious account information
- They can't lose or create money
- A bank **must** be in balance **all the time**

Design recipe: banking example

Say you want to give $100 to your cousin Paul for Christmas.
You need to:

decrease your account balance by 100$

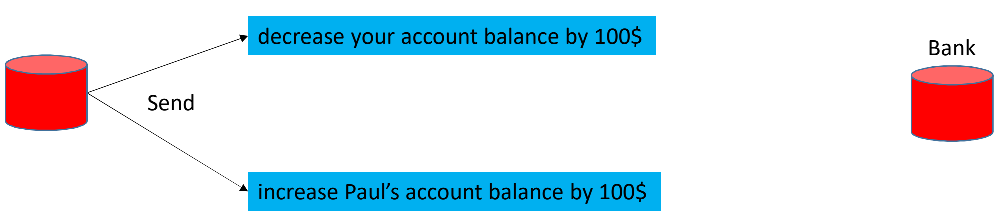
```
{
_id: "account_123456",
account:"bank_account_001",
balance: 900,
timestamp: 1290678353,45,
categories: ["bankTransfer"…],
…
}
```

increase Paul's account balance by 100$

```
{
_id: "account_654321",
account:"bank_account_002",
balance: 1100,
timestamp: 1290678353,46,
categories: ["bankTransfer"…],
…
}
```

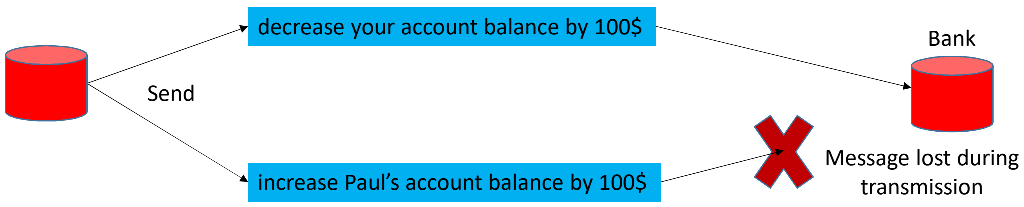
Design recipe: banking example

- What if some kind of failure occurs between the two separate updates to the two accounts?



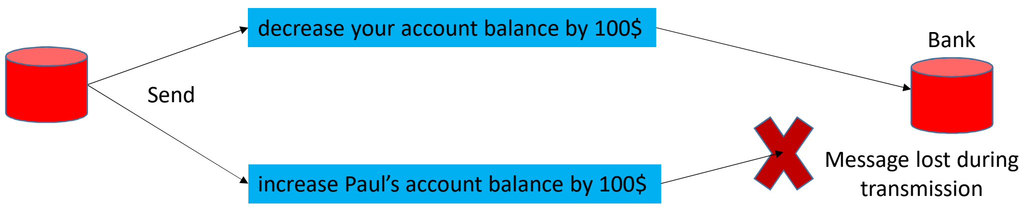
Design recipe: banking example

- What if some kind of failure occurs between the two separate updates to the two accounts?



Design recipe: banking example

- What if some kind of failure occurs between the two separate updates to the two accounts?



- CouchDB **cannot guarantee the bank balance**.
- A different strategy (design) must be adopted.

Banking recipe solution

- What if some kind of failure occurs between the two separate updates to the two accounts?
- CouchDB cannot guarantee the bank balance.
- A different strategy (design) must be adopted.

```
id:     transaction001
from:   "bank_account_001",
to:     "bank_account_002",
qty:    100,
when:1290678353.45,
...
```

Design recipe: banking example

- How do we read the current account balance?
- Map

```
function(transaction){
 emit(transaction.from, transaction.amount*-1);
 emit(transaction.to, transaction.amount);
}
```

- Reduce

```
function(key, values){
 return sum(values);
}
```

- Result

```
{rows: [ {key: "bank_account_001", value: 900} ]
{rows: [ {key: "bank_account_002", value: 1100} ]
```

The reduce function receives:
- **key**= bank_account_001, **values**=[1000, -100]
- ...
- **key**= bank_account_002, **values**=[1000, 100]
- ...

Beyond relational databases

Daniele Apiletti
Data Base and Data Mining group
Politecnico di Torino
http://dbdmg.polito.it

http://ict-ontic.eu/ http://ict-mplane.eu http://ooros.com