Beyond relational databases

Technologies for Data Management

- Distributed file systems (GFS, HDFS, etc.)
- MapReduce
    - and other models for distributed programming
- NoSQL databases
- Data Warehouses
- Grid computing, cloud computing
- Large-scale machine learning

Relational Database Management Systems

- RDBMS are predominant database technologies
  - first defined in 1970 by Edgar Codd of IBM's Research Lab
- Data modeled as relations (tables)
  - object = tuple of attribute values
    - each attribute has a certain domain
  - a table is a set of objects (tuples, rows) of the same type
    - relation is a subset of cartesian product of the attribute domains
  - each tuple identified by a primary key
    - field (or a set of fields) that uniquely identifies a row
  - tables and objects “interconnected” via foreign keys
- SQL query language

RDBMS Example

Students

ID#	Name	Phone	DOB
500	Matt	555-4141	06/03/70
501	Jenny	867-5309	3/15/81
502	Sean	876-9123	10/31/82

ID#	ClassID	Sem
500	1001	Fall02
501	1002	Fall02
501	1002	Spr03
502	1003	S203

Takes_Course

ClassID	Title	ClassNum
1001	Intro to Informatics	I101
1002	Data Mining	I400
1003	Internet and Society	I400

Courses

SELECT Name
FROM Students S, Takes_Course T
WHERE S.ID=T.ID AND ClassID = 1001

source: https://github.com/talhafazal/DataBase/wiki/Home-Work-%23-3-Relational-Data-vs-Non-Relational-Databases

Fundamentals of RDBMS

Relational Database Management Systems (RDMBS)

1. Data structures are broken into the smallest units
   - normalization of database schema
     - because the data structure is known in advance
     - and users/applications query the data in different ways
   - database schema is rigid
2. Queries merge the data from different tables
3. Write operations are simple, search can be slower
4. Strong guarantees for transactional processing

From RDBMS to NoSQL

Efficient implementations of table joins and of transactional processing require centralized system.

NoSQL Databases:

- Database schema tailored for specific application
  - keep together data pieces that are often accessed together
- Write operations might be slower but read is fast
- Weaker consistency guarantees

=> efficiency and horizontal scalability

Data Model

- The model by which the database organizes data
- Each NoSQL DB type has a different data model
  - Key-value, document, column-family, graph
  - The first three are oriented on aggregates

- Let us have a look at the classic relational model

The Value of Relational Databases

- A (mostly) standard data model
- Many well developed technologies
  - physical organization of the data, search indexes, query optimization, search operator implementations
- Good concurrency control (ACID)
  - transactions: atomicity, consistency, isolation, durability
- Many reliable integration mechanisms
  - “shared database integration” of applications
- Well-established: familiar, mature, support,...

RDBMS for Data Management

- relational schema
  - data in tuples
  - a priori known schema
- schema normalization
  - data split into tables
  - queries merge the data
- transaction support
  - trans. management with ACID
  - Atomicity, Consistency, Isolation, Durability
  - safety first

- however, real data are naturally flexible

- inefficient for large data
- slow in distributed environment

- full transactions very inefficient in distributed environments

«NoSQL» birth

- 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**

What is «NoSQL»?

- Term used in late 90s for a different type of technology
  - Carlo Strozzi: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/
- “Not Only SQL”?
  - but many RDBMS are also “not just SQL”

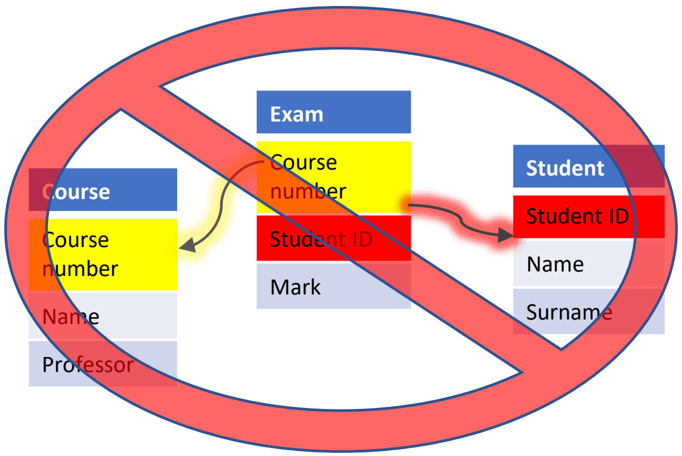
“NoSQL is an accidental term with no precise definition”

- first used at an informal meetup in 2009 in San Francisco (presentations from Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, and MongoDB)

[Sadalage & Fowler: NoSQL Distilled, 2012]

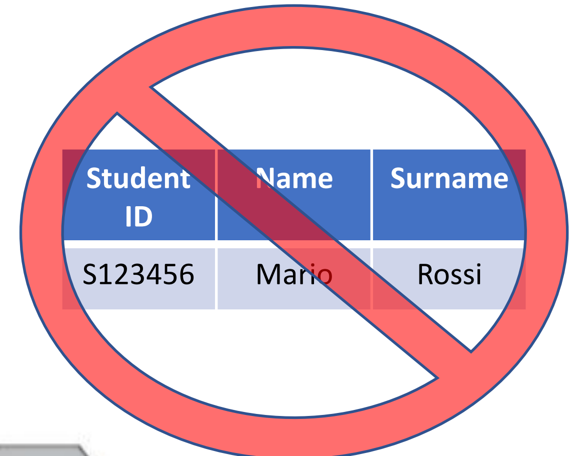
NoSQL main features

no joins

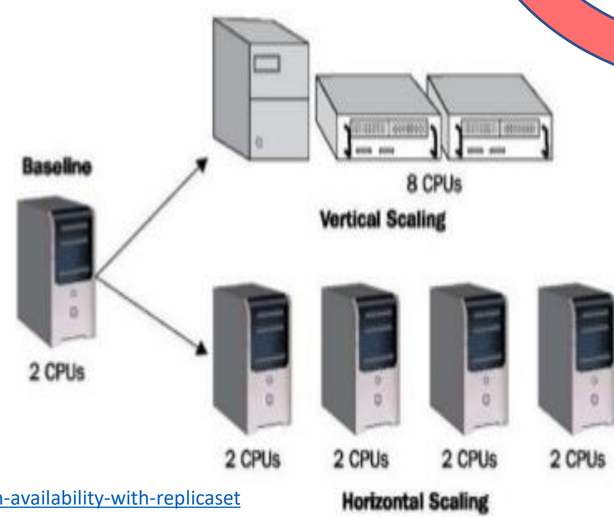


schema-less

(no tables, implicit schema)



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

Comparison

Relational databases	Non-Relational databases
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
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

Non-relational/NoSQL DBMSs

Pros

- Work with semi-structured data (JSON, XML)
- Scale out (horizontal scaling – parallel query performance, replication)
- High concurrency, high volume random reads and writes
- Massive data stores
- Schema-free, schema-on-read
- Support records/documents with different fields
- High availability
- Speed (join avoidance)

Non-relational/NoSQL DBMSs

Cons

- Do not support strict ACID transactional consistency
- Data is de-normalized
  - requiring mass updates (e.g., product name change)
- Missing built-in data integrity (do-it-yourself in your code)
- No relationship enforcement
- Weak SQL
- Slow mass updates
- Use more disk space (replicated denormalized records, 10-50x)
- Difficulty in tracking “schema” (set of attribute) changes over time

Just Another Temporary Trend?

- There have been other trends here before
  - object databases, XML databases, etc.

- But NoSQL databases:
  - are answer to real practical problems big companies have
  - are often developed by the biggest players
  - outside academia but based on solid theoretical results
    - e.g. old results on distributed processing
  - widely used

Challenges of NoSQL Databases

1. Maturity of the technology
   - it's getting better, but RDBMS had a lot of time
2. User support
   - rarely professional support as provided by, e.g. Oracle
3. Administration
   - massive distribution requires advanced administration
4. Standards for data access
   - RDBMS have SQL, but the NoSQL world is more wild
5. Lack of experts
   - not enough DB experts on NoSQL technologies

The End of Relational Databases?

- Relational databases are not going away
  - are ideal for a lot of structured data, reliable, mature, etc.
- RDBMS became one option for data storage

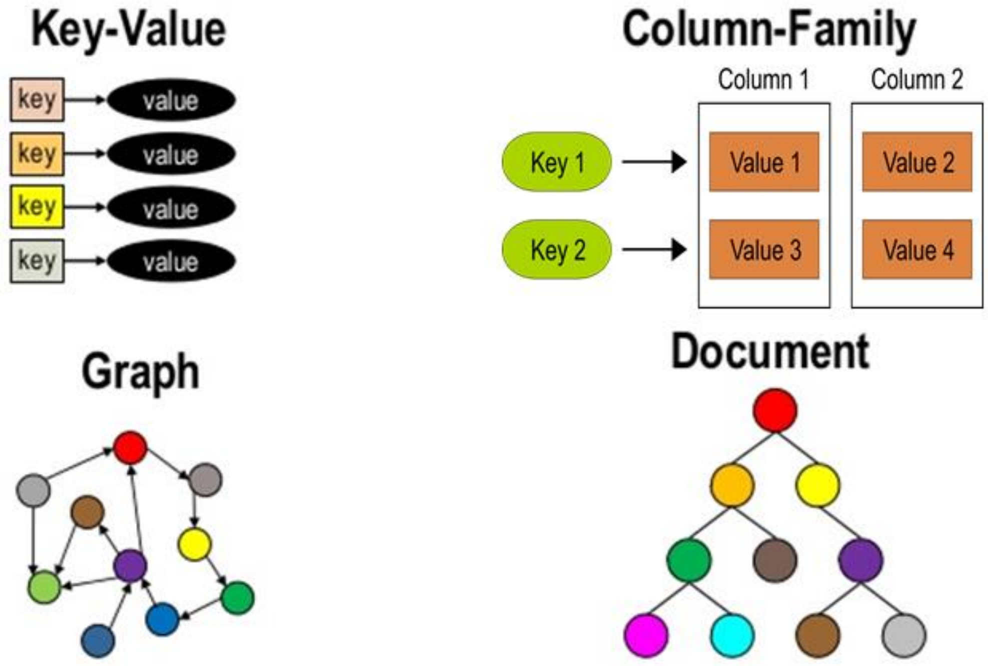
Polyglot persistence – using different data stores in different circumstances

[Sadalage & Fowler: NoSQL Distilled, 2012]

Two trends

1. NoSQL databases implement standard RDBMS features
2. RDBMS are adopting NoSQL principles

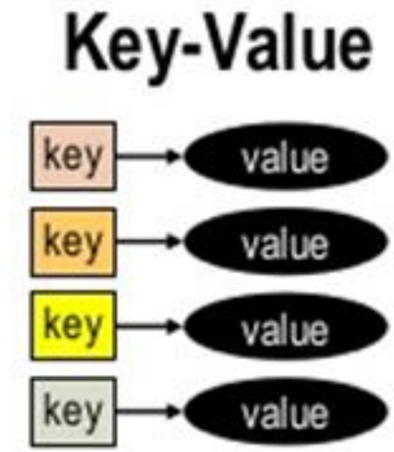
Types of NoSQL databases



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

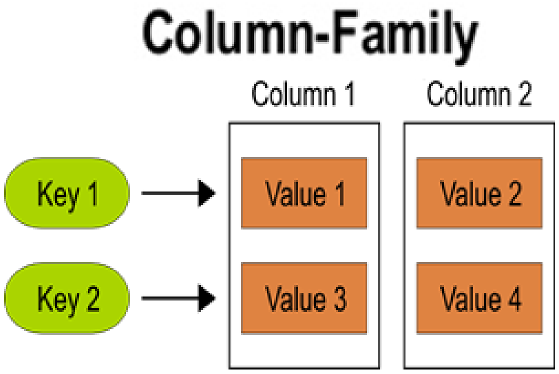
Key-values databases

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



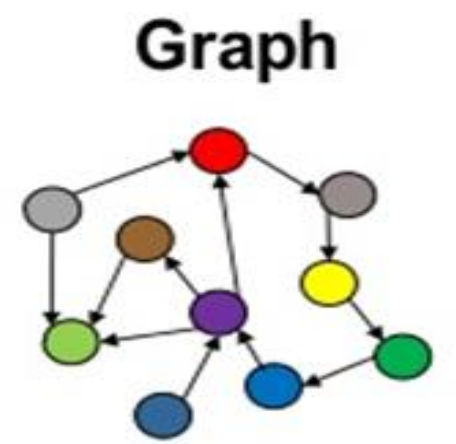
Column-oriented databases

- Store data in **columnar** format
  - Name = "*Daniele*":row1,row3; "*Marco*":row2,row4; …
  - Surname = "*Apiletti*":row1,row5; "*Rossi*":row2,row6,row7…
- 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



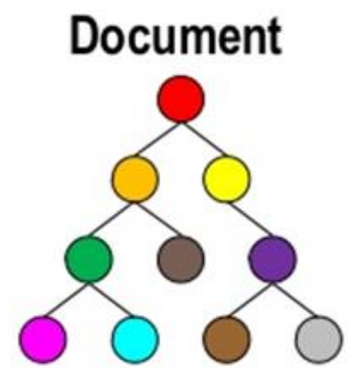
Graph databases

- Based on graph theory
- Made up by **Vertices** and unordered **Edges** or ordered **Arcs** between each Vertex pair
- 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.



Document-based model

- Strongly **aggregate**-oriented
  - Lots of aggregates
  - Each aggregate has a key
  - Each aggregate is a document
- Data model
  - A **set of <key,value> pairs**
  - Document: an aggregate instance of <key,value> pairs
- Access to an aggregate
  - Queries based on the fields in the aggregate

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}
```

```
# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Document basics

- Basic concept of data: *Document*
- Documents are self-describing pieces of data
  - Hierarchical tree data structures
  - Nested associative arrays (maps), collections, scalars
  - XML, JSON (JavaScript Object Notation), BSON, …
- Documents in a collection should be “similar”
  - Their schema can differ
- Documents stored in the value part of key-value
  - Key-value stores where the values are examinable
  - Building search indexes on various keys/fields

Document Example

```
key=3 ->  { "personID": 3,
            "firstname": "Martin",
            "likes": [ "Biking","Photography" ],
            "lastcity": "Boston",
            "visited": [ "NYC", "Paris" ] }

key=5 ->  { "personID": 5,
            "firstname": "Pramod",
            "citiesvisited": [ "Chicago", "London","NYC" ],
            "addresses": [
               { "state": "AK",
                 "city": "DILLINGHAM" },
               { "state": "MH",
                 "city": "PUNE" }  ],
            "lastcity": "Chicago" }
```

source: Sadalage & Fowler: NoSQL Distilled, 2012

Queries on Documents

Example in MongoDB syntax

- Query language expressed via JSON
- clauses: where, sort, count, sum, etc.

SQL: `SELECT * FROM users`
MongoDB: `db.users.find()`

```
SELECT *
FROM users
WHERE personID = 3
```

```
db.users.find( { "personID": 3 } )
```

```
SELECT firstname, lastcity
FROM users
WHERE personID = 5
```

```
db.users.find( { "personID": 5}, {firstname:1, lastcity:1} )
```

Document Databases: Representatives

mongoDB

Apache CouchDB relax

OrientDB

RAVENDB

terrastore

PostgreSQL

amazon DynamoDB

MS Azure DocumentDB

Ranked list: http://db-engines.com/en/ranking/document+store

Distributed Data Management

Introduction to
data replication and
the CAP theorem

Replication

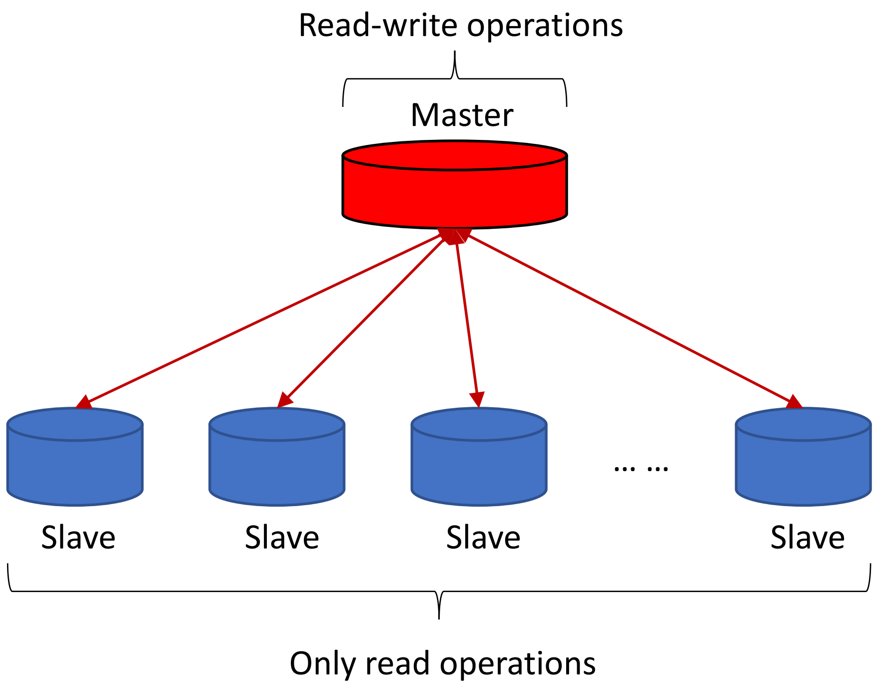
Same data
in **different** places

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

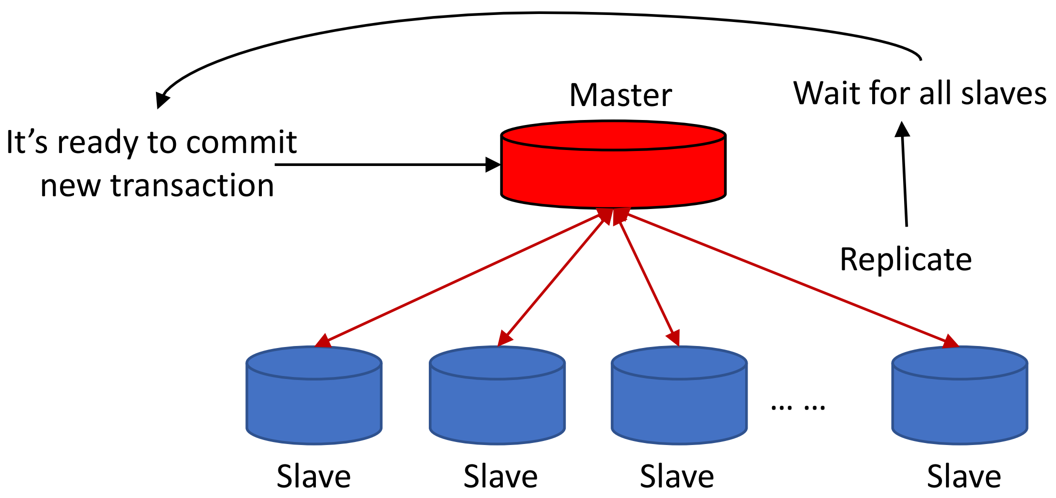
Master-Slave replication

- 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**
- Some NoSQLs (e.g., CouchDB) support Master-Master replica



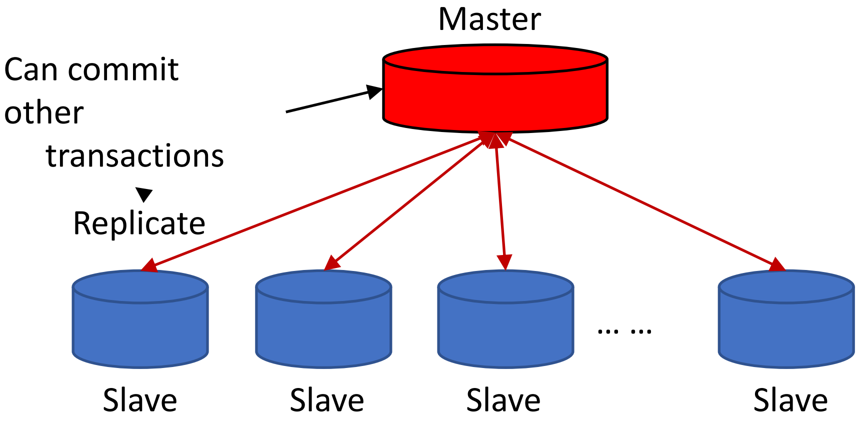
Synchronous replication

- 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



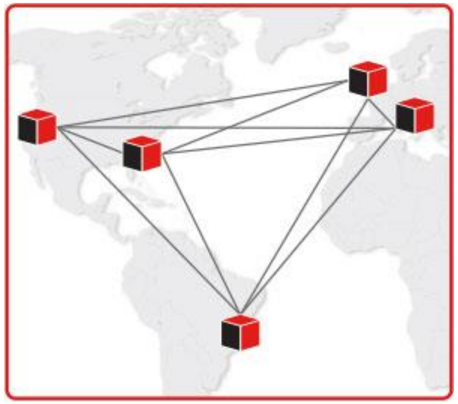
Asynchronous replication

- 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**

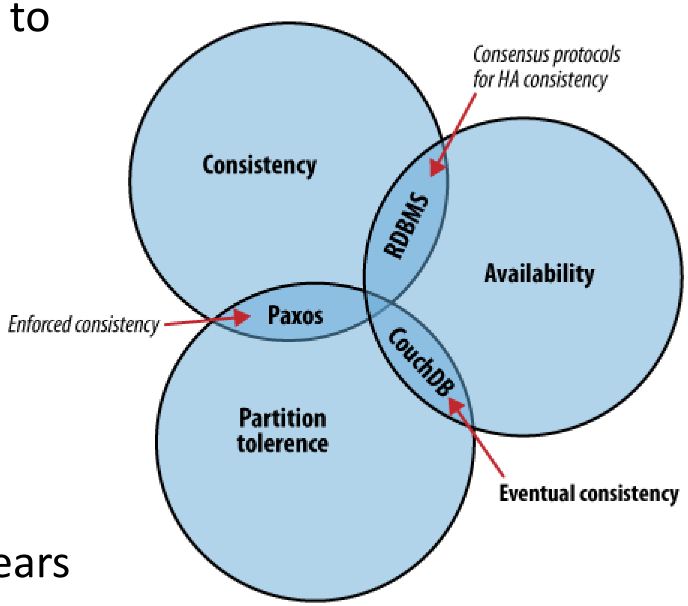


Key features of distributed databases

- 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

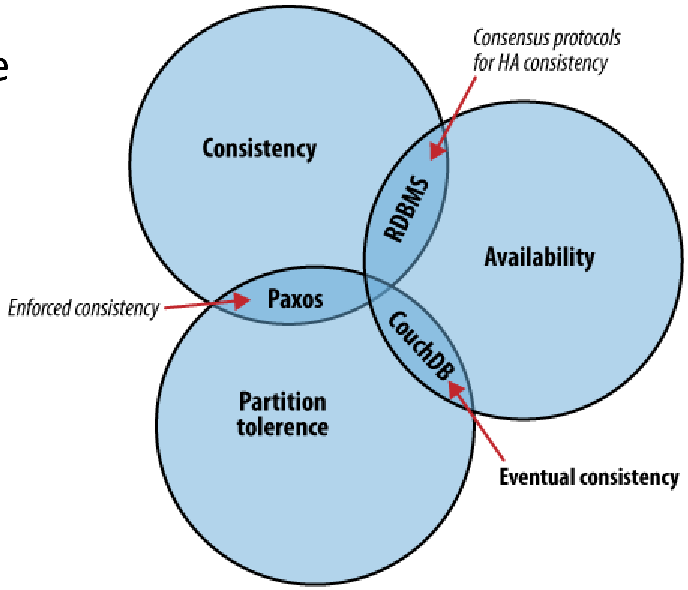
- The CAP theorem, also known as Brewer's theorem, states that it is **impossible** for a distributed system to **simultaneously** provide **all three** of the previous guarantees
- The theorem began as a **conjecture** made by University of California in 1999-2000
  - Armando Fox and Eric Brewer, “Harvest, 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

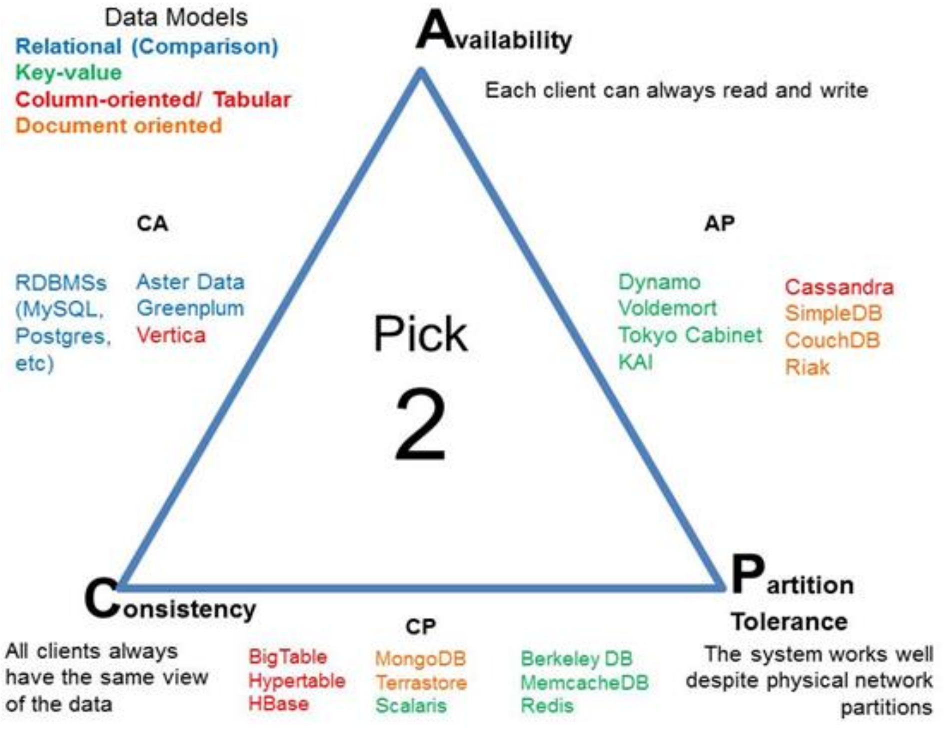
CAP Theorem

- 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
  - There are also many levels of consistency
  - Even partitions have nuances, including disagreement within the system about whether a partition exists

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**

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

ACID versus BASE

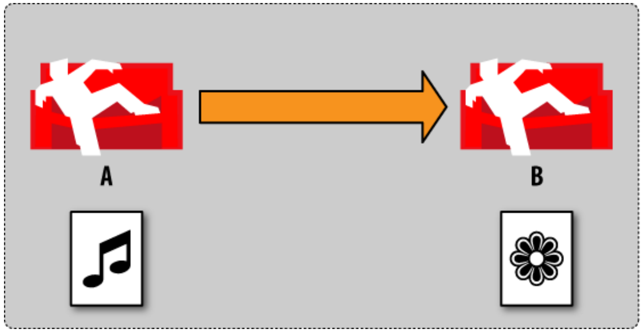
- ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum
- ACID properties focus on **consistency** and are the traditional approach of databases
- BASE properties focus on high **availability** and to make explicit both the choice and the spectrum
-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

Conflict detection and resolution

An example from a notable NoSQL database

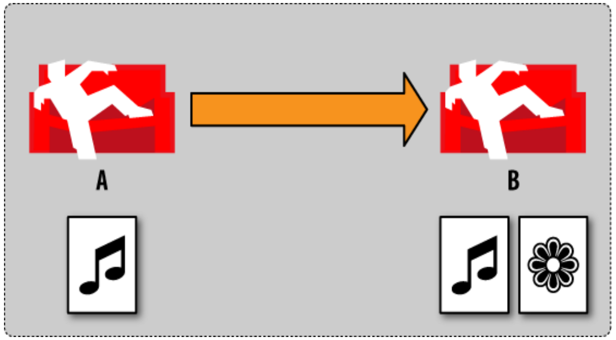
Conflict resolution problem

- There are two customers, **A** and **B**
- **A** books a hotel room, the last available 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.

A design recipe

A notable example of NoSQL design for «distributed transactions»

Design recipe: banking account

- Banks are serious businesses
- 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$

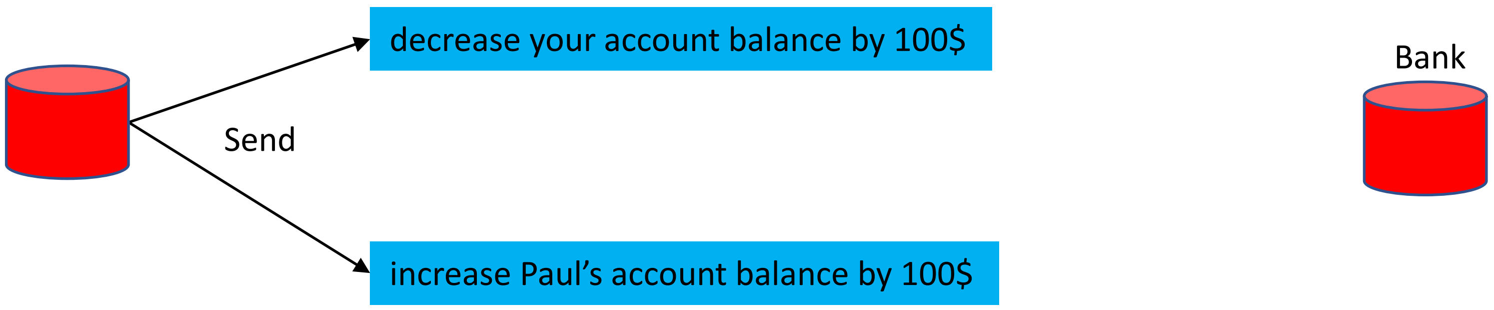
```
{
_id: "account_123456",
account:"bank_account_001",
balance: 900,
timestamp: 1290678353,45,
categories: ["bankTransfer"…],
…
}
```

increase Paul's account balance by 100$

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

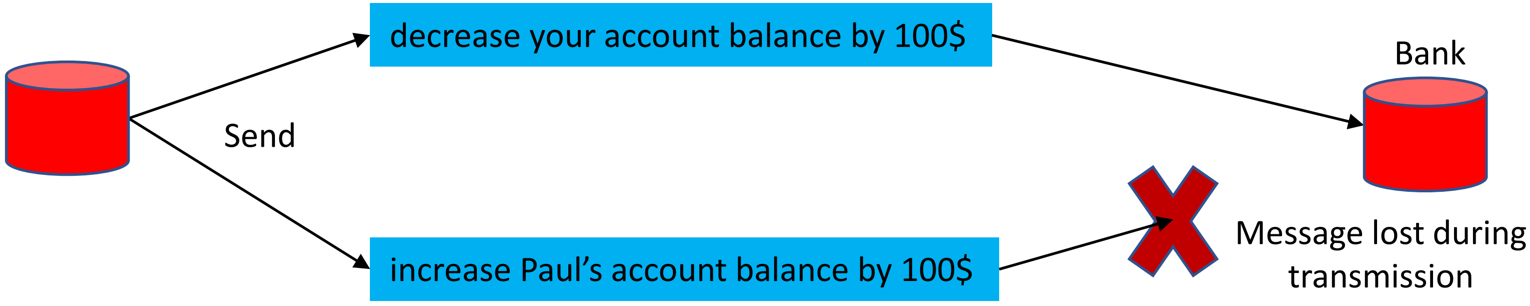
Design recipe: banking example

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



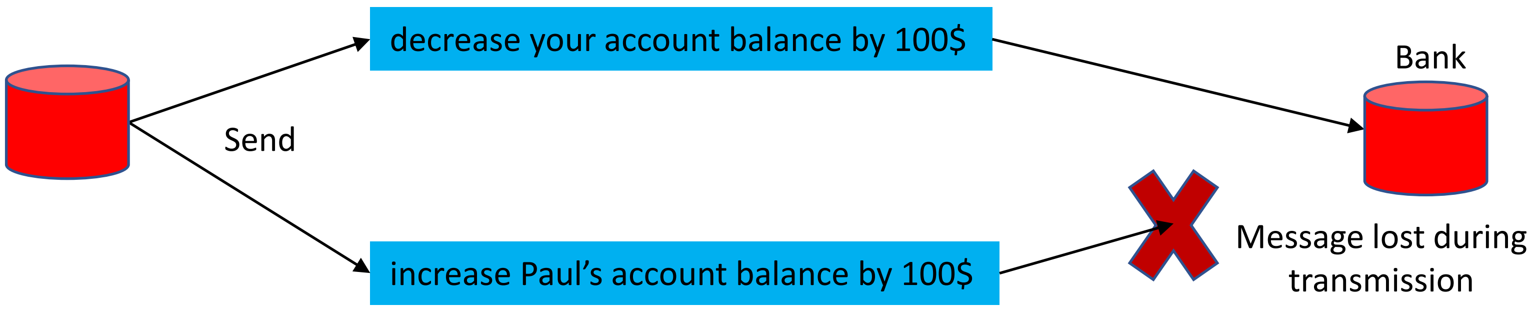
Design recipe: banking example

- 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?



- The NoSQL DB **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?
- A NoSQL database without 2-Phase Commit 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]
- …