

Introduction to NoSQL

Olivier Curé

Université Paris-Est Marne la Vallée , LIGM UMR CNRS 8049, France

February 9, 2016

NoSQL stands for **Not Only SQL**

- RDBMS have been successful for more than twenty years, providing standards (SQL), persistence, concurrency control, and an integration mechanism (several apps accessing a single database).
- App developers have been frustrated with the impedance mismatch between the relational model and the in-memory data structures.
- Exponential growth of data set size (161Eo (2006) to 988Eo (2010) created and replicated data) requires a cluster-based approach to DBMS.
- RDBMS are not designed to run efficiently on a cluster (e.g., Oracle RAC and SQL Server clustered version use a shared-disk approach → disk system as a SPOF)

- Connectivity of data (e.g., graph databases) and data locality.
- Structure of documents (structured, semi-structured, unstructured) → RDBMS performance
- Architecture of database oriented applications (one database for one app)

- Common characteristics of NoSQL databases:
 - Not based on the relational model.
 - Run well on cluster (except graph databases)
 - Most of them are open-source
 - Schemaless
 - No joins

4 categories

- Key-value
- Column family (aka Bigtable-like)
- Document
- Graph DB

Key-Value

- Origin: Dynamo @ Amazon ¹
- Data model: global key-value mapping. Distributed hash mapp
- Systems : Voldemort (LinkedIn), Tokyo (Cabinet, Tyrant), Riak (Basho), Oracle NoSQL, Redis

¹G. De Candia et al. Dynamo: Amazon's highly available key-value store.
SOSP 2007

Column family

- Origin: Bigtable @ Google ²
- Data model: a big table with column families
- Systems : HBase (Apache), Cassandra (Apache), HyperTable

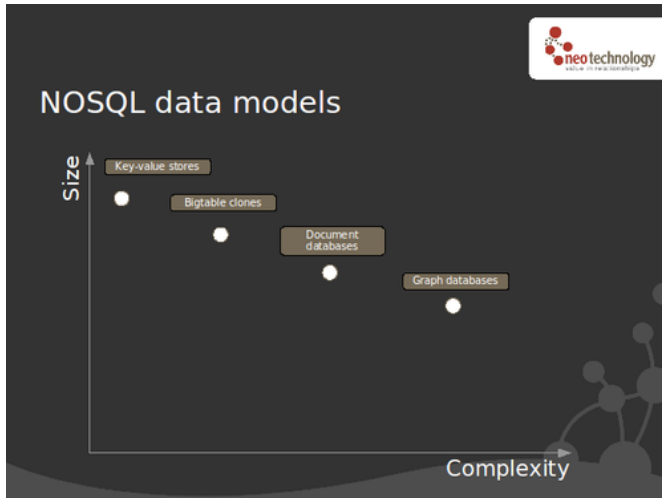
²F. Chang et al. Bigtable: a distributed storage system for structured data.
OSDI 2006

Document

- Origin: Lotus notes
- Data model: Collections of documents where a document is a key-value collection
- Systems: CouchDB (Apache), MongoDB (10gen), Terrastore

Graph

- Origin: Graph theory
- Data model (property graph): Nodes with properties. Typed relationships with properties
- Systems: Neo4J, InfiniteGraph, Sones GraphDB, Trinity (Microsoft), FlockDB (Apache)



- One can generally represent an instance in one model into another model.
- Implemented systems:
 - Common features: Schemaless, no joins
 - Main differences: consistency approach, conflict detection, concurrency control, integration of parallelization

- Key-Value, Document and Column family stores can be considered as Aggregate oriented.
- Enables to operate on data units that are more complex than a set of tuples.
- Example of complex structures: lists, nested structures.
- Aggregate: collection of object information that one wants to treat as a unit.

- Pros of aggregate vision:
 - updating with atomic operations,
 - accessing data storage (data locality = everything about an object is in one place) → eases the job of app programmers since they manipulate aggregate structures in their programs.
 - replicating,
 - sharding.

- Aggregate oriented stores and transactions
 - these stores usually do not have ACID transactions spanning multiple aggregates.
 - Only atomic manipulation of a single aggregate at a time.
 - If one needs to handle several aggregate at a time, she needs to handle that in the app code.
 - Not necessary if one models appropriately.

- Key-value stores
 - Aggregate is opaque to the DBMS value position
 - Only lookup based on the key.
- Document
 - DBMS sees the aggregate
 - Limits the allowed structures and types.
 - Access flexibility (secondary indexes on elements of the aggregate)
- Fuzzy boundary between KV and document stores: Redis enables to define lists, sets, hashes, Riak supports metadata for indexing and interaggregate links.

■ Column stores

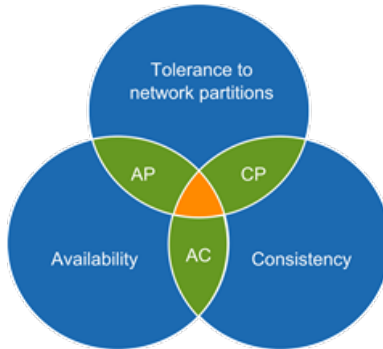
- a two level aggregate structure: first key as a column identifier which points to a second level (the column)
- operations enable to pick a particular column from a given row.
- Columns are organized as column families.
- Each column is part of single column-family and acts as a unit of access.

CAP conjecture from Brewer³ and theorem ⁴

- **Consistency:** A service that operates fully or not (in fact more like Atomic)
- **Availability:** The service is available
- **Partitioning tolerance:** no set of failures less than total network failure is allowed to cause the system to respond incorrectly.

³talk at ACM PODC 2000

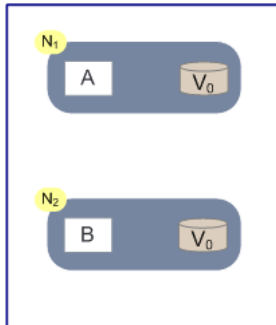
⁴S. Gilbert, N.Lynch: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2): 51-59 (2002)



Write-write consistency: 2 users want to write different values for the same data (e.g. phone numbers)

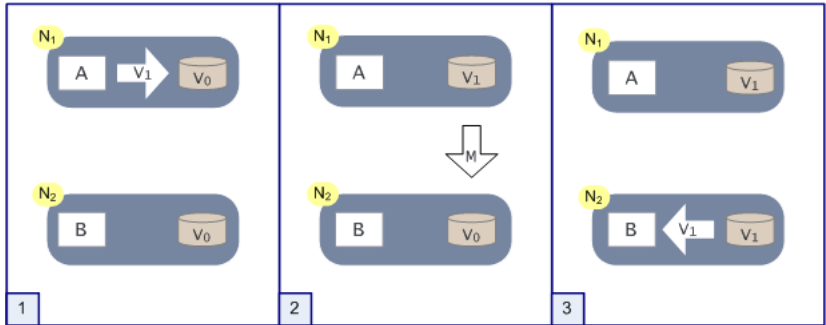
- In general 2 approaches: pessimistic (prevents conflict to occur) and optimistic (lets conflicts occur, but detects them and takes action to sort them out).
- Main pessimistic approach uses logs
- Main optimistic approaches: conditional updates (forces the user to check whether the to be modifier value has been recently updated) or save both updates, record that they are in conflict and asks someone to fix it (close to version control systems).

Example from ⁵

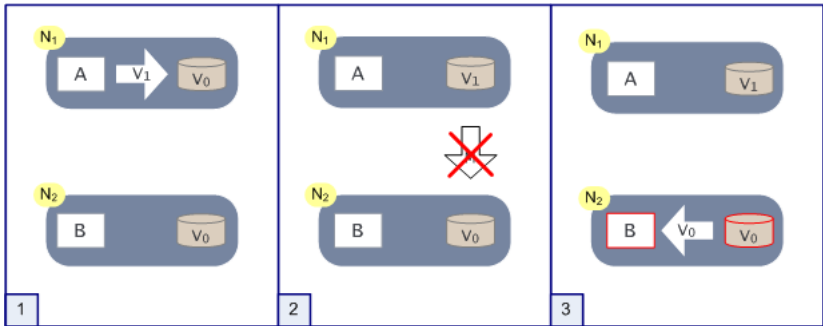


2 nodes (N_1 and N_2 sharing a piece of info V with value V_0 . A (writer) and B (reader) are reliable algos.

⁵Julian Browne's blog: <http://tinyurl.com/cxvk7z>



- (1) A writes a new value V1,
- (2) Message is passed from N₁ to N₂,
- (3) B reads the new value V1



In case of a partitioned network (2), at (3) B reads an inconsistent value.

- if M is synchronous: latency issues
- if M is asynchronous, N_1 as no way to know whether N_2 has received the message.
- if we want high availability of A and B, N_1 and N_2 to be tolerant to partition partitioning then we must accept that B reads inconsistent data.

- CA, i.e. drop partition tolerance : everything on one machine. No scaling out.
- CP, i.e. drop availability: latency issues. Complex recovery issues
- AP, i.e. drop consistency. In fact there is a spectrum of consistency. This approach is quite popular in NoSQL stores.

Visual Guide to NoSQL Systems



- Different solutions to relax consistency are presented in ⁶
- (Internet) systems must always be available. With CAP theorem, you have the choice of either CA or AP.
- The developer then has to deal with the adopted solution:
 - CA: what to do in the case of a network failure?
 - AP: does the client needs th absolute latest update all the time? Many applications can handle stale data.

⁶Werner Vogels: Eventually consistent. Commun. ACM 52(1): 40-44 (2009)

- A transaction is a sequence of database operations (read/write)
- **Atomicity:** all or none updates are executed
- **Consistency:** DB instance must go from one consistent state to another
- **Isolation:** Results of a transaction are visible to other users after a commit
- **Durability:** Committed transactions are persisted
- Is the responsibility of the developer but is assisted by the RDBMS.

ACID in distributed RDBMS

- 2 Phase Commit (2PC):
 - Phase 1: transaction coordinator asks each involved DB to precommit the operation and tell if commit is possible
 - Phase 2: transaction coordinator asks each involved DB to commit the data.
- If any involved DB votes commit then they all roll back.

- Example: S is a storage system, A, B and C are processes.
- A updates a given value in S
- **Strong consistency**: any subsequent access (by A, B or C) will return the updated value.
- **Weak consistency**: no guarantee that subsequent accesses will return the updated value.
- Period between update and the time at which any process is guaranteed to access it is the **inconsistency window**.

- An interesting form of weak consistency is **eventual consistency** where the storage system guarantees that if no other updates are made to the object then eventually all accesses will return the last updated value (e.g. DNS).
- Different variations of eventual consistency:
 - **Causal consistency**: A communicates to B that a new update is available. Subsequent accesses by B will return the last value. Uncertain for C.
 - **Read-your-writes consistency**: A will always access the last update and will never see an older value.
 - **Session consistency**: same as previous but in the context of a session.

- More variations of eventual consistency:
 - **Monotonic read consistency:** Never read an older value than the one you accessed
 - **Monotonic write consistency:** System guarantees to serialize the writes by the same process.
- These variations can be combined. For instance, monotonic reads + monotonic writes consistency is desirable.

- N: the number of nodes that store replicas of the data.
- W: the number of replicas that need to acknowledge the receipt of the update before the update is completed
- R: the number of replicas that are contacted when a data object is accessed through a read operation.
- **$W+R > N$** : guarantees strong consistency. Ex: $N=2$, $W=2$, $R=1$.
 - such a quorum protocol fails if the system can't write the W nodes.

- In distributed storage systems, generally $N \geq 2$
- Typical scenarios:
 - Focus on fault tolerance: $N=3$, $W=2$ and $R=2$
 - Focus on very high read loads: N can be 10 or 100 and $R=1$
 - Focus on consistency: $W=N$
 - Focus on fault tolerance and not consistency: $W=1$ with associated replica mechanisms.

- Configuring N, W and R:
 - $R=1, W=N$ for fast reads
 - $W=1, N=R$ for fast writes
 - Weak/Eventual consistency when $W + R \leq N$: read/write may not overlap

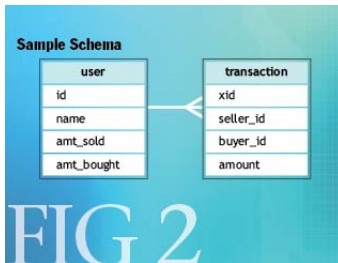
BASE⁷

- stands for Basic Availability, Soft state, Eventually Consistent.
- requires an in-depth analysis of the operations within a logical transaction.
- based on consistency patterns

⁷Dan Pritchett: Base An ACID alternative. ACM Queue may 2008.

Comparison

	ACID	BASE
Consistency	Strong	Weak
Approach	Pessimistic	Optimistic
Focus	on commit	on availability
Schema evolution	Isolation difficult	flexible
		faster
		Simpler
		Best effort



Standard (consistent) transaction

Begin transaction

```
Insert into transaction(xid, seller_id, buyer_id, amount);
```

```
Update user set amt_sold=amt_sold+$amount where id=$seller_id;
```

```
Update user set amt_bought=amount_bought+$amount where id=$buyer_id;
```

End transaction

FIG 3

The user table can be considered a cache of the transaction table.
It is used for efficient.

By decoupling the updates to the user and transaction tables, we
can relax the transaction.

Relaxed consistency transaction

Begin transaction

```
Insert into transaction(id, seller_id, buyer_id, amount);
```

End transaction

Begin transaction

```
Update user set amt_sold=amt_sold+$amount where id=$seller_id;
```

```
Update user set amt_bought=amount_bought+$amount  
where id=$buyer_id;
```

End transaction

FIG 4

In case of failure between the 2 transactions, user table may be permanently inconsistent.

If it is considered that the amt_sold and amt_bought are estimates, this is fine. Otherwise we have problem.

with persistent message queue

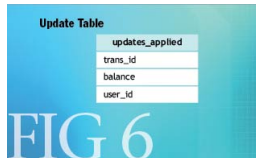
```
Begin transaction
  Insert into transaction(id, seller_id, buyer_id, amount);
  Queue message "update user("seller", seller_id, amount)";
  Queue message "update user("buyer", buyer_id, amount)";
End transaction
For each message in queue
  Begin transaction
    Dequeue message
    If message.balance == "seller"
      Update user set amt_sold=amt_sold + message.amount
        where id=message.id;
    Else
      Update user set amt_bought=amt_bought + message.amount
        where id=message.id;
    End if
  End transaction
End for
```

FIG 5

with queue stored on the same machine as the database (to avoid 2PC when queueing) but we have a 2PC when dequeuing.

Idempotence at the rescue

- An idempotent operation can be applied one or several times with the same result.
- Idempotent operations permit partial failures, as applying them repeatedly does not change the final state of the system.
- Update operations are generally not idempotent.
- In the case of balance updates, you need a way to track which updates have been applied successfully and which are still outstanding. One technique is to use a table that records the transaction identifiers that have been applied.



updates_applied
trans_id
balance
user_id

FIG 6

Solution handling partial failure with no 2PC transaction

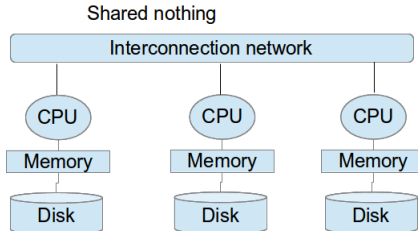
```
Begin transaction
Insert into transaction(id, seller_id, buyer_id, amount);
Queue message "update user("seller", seller_id, amount)";
Queue message "update user("buyer", buyer_id, amount)";
End transaction
For each message in queue
  Peek message
  Begin transaction
  Select count(*) as processed where trans_id=message.trans_id
    and balance=message.balance and user_id=message.user_id
  If processed == 0
    If message.balance == "seller"
      Update user set amt_sold=amt_sold + message.amount
        where id=message.id;
    Else
      Update user set amt_bought=amt_bought + message.amount
        where id=message.id;
    End if
  Insert into updates_applied
    (message.trans_id, message.balance, message.user_id);
  End if
End transaction
If transaction successful
  Remove message from queue
End if
End for
```

FIG 7

- Database Management systems (DBMS) are candidates for deployment in the cloud.
- ⁸ studies which DBMS are most likely to succeed on the cloud
- Two main approaches to study: OLTP and OLAP
- Decision is motivated by the architecture found on the cloud.

⁸D. Abadi. Data management in the cloud: limitations and opportunities.
Data Engineering. Vol 32 no1. 2009

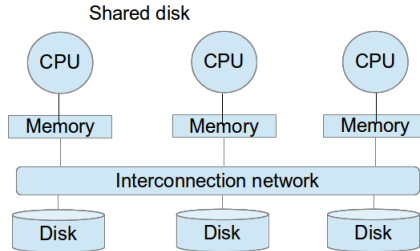
- Architecture generally adopted for cloud computing
- Efficient for high scalability but with a high cost of data partitioning



Shared nothing

- How to effectively distribute the data across the nodes is crucial for performance and scalability.
- Important for leveraging data parallelism and to reduce the amount of data that needs to be transferred over the network during query processing.

Shared disk

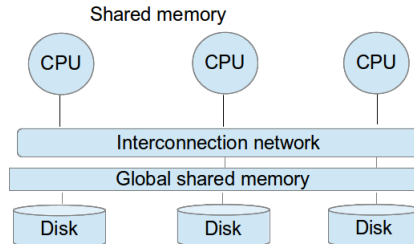


- no need to a priori partition data across nodes since all nodes access data via shared storage.
- no need to move data across nodes.
- load balancing is easy since all nodes can service any request

Shared disk

- the nodes need to communicate in order to ensure data consistency (via distributed lock manager).
- Network must support the combined IO bandwidth of all processors, and can become a bottleneck.
- Shared disk are relatively cost effective for small to medium sized data warehouse.

Shared memory



- Database Management systems (DBMS) are generally used to **On-Line Transactional Processing (OLTP)**
- On operational DB of average sizes (few TB), write intensive and requiring complete ACID transactional properties, strong data properties and response time guarantees.
- Typical use cases: item reservations (airline, concerts, etc.), on-line e-commerce, supply chain management, financial activities.

OLTP operations

- are structured and repetitive
- require detailed and up-to-date database
- are short, atomic and isolated transactions

- **On-Line Analytical Processing** deals with historical DBs of very large sizes (up to PB), read-intensive and hence relax ACID properties.
- Typical use case: business planning, problem solving and decision making/support.
- OLAP was a 4 billion \$ market of the 14.6 billion \$ of DB market with an annual growth of 10.3%.
- OLAP data are typically extracted from operational OLTP DBs → sensitive data can be anonymized. So called **ETL** (Extract Transform Load).

OLAP

- is supported by data warehouses (typically RDBMS with extended operations (cube, roll-up, drill-down, etc.).
- has some historical (temporal), summarized, integrated, consolidated and multidimensional data.
- used for business intelligence. Read *Information platform and the rise of the data scientist*. J. Hammerbacher in *Beautiful data* (O'reilly 2009) fo Facebook's evolution on this subject.

Currently OLAP is more suitable than OLTP for cloud computing for the following reasons:

- Elasticity requires a shared nothing cluster architecture
 - OLAP : effective data partitioning and parallel query processing. ACID not needed.
 - OLTP : complex concurrency control. Shared disk is more efficient. Hard to maintain data replication across geographically distributed data centers.
- Security
 - OLAP : anonymization of sensitive data coming from ETL process.
 - OLTP : no anonymization is possible. Resistance of customers.

- More ACIDity⁹
 - MongoDB adding durable logging storage in 1.7
 - Cassandra adding stronger consistency in 1.0

⁹Emil Eifrem at NoSQL eXchange 2011

- More Query languages
 - MongoDB had one right from the start
 - Cassandra : CQL
 - Couchbase : UnQL
 - Neo4J : Cypher
 - Gremlin in many graph and RDF Stores.

- Multi-model data stores
 - Sqrrl: key-value, document, graph and column
 - ArangoDB, OrientDB: document, graph and key-value
 - Mark Logic: XML, document and RDF store (graph)