Beyond relational databases

Daniele Apiletti

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

«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

NoSQL main features

schema-less (no tables, implicit schema)

horizontal scalability

no joins

Comparison

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

Comparison of NoSQL databases

Types of NoSQL databases

Key-Value

Column-Family

Graph

Document

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 Of Unreliable Commodity Hardware

CouchDB original home page

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.

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

Provides a RESTful JSON API that can be accessed from any environment that allows HTTP requests.

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

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

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 that can be accessed from any environment that allows HTTP requests.

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

MapReduce

A scalable distributed programming model to process Big Data.

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

MapReduce example database, a collection of docs describing university exam records.
Map example (1)

- List of exams and corresponding mark

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

Result:

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>18</td>
</tr>
<tr>
<td>Computer</td>
<td>24</td>
</tr>
<tr>
<td>Database</td>
<td>26</td>
</tr>
</tbody>
</table>
```

Map example (2)

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

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

Result:

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software, 2016</td>
<td>18</td>
</tr>
<tr>
<td>Computer, 2015</td>
<td>24</td>
</tr>
<tr>
<td>Database, 2016</td>
<td>26</td>
</tr>
</tbody>
</table>
```

Map example (3)

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

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

Result:

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>student1</td>
<td>[29, 8]</td>
</tr>
<tr>
<td>student2</td>
<td>[26, 8]</td>
</tr>
<tr>
<td>student3</td>
<td>[29, 8]</td>
</tr>
</tbody>
</table>
```

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-like 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

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

MapReduce example (2)

- Map - List of exams and corresponding mark

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

Reduce

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

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

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

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software, 2015</td>
<td>18</td>
</tr>
<tr>
<td>Computer, 2015</td>
<td>24</td>
</tr>
<tr>
<td>Database, 2016</td>
<td>26</td>
</tr>
<tr>
<td>Database, 2016</td>
<td>25</td>
</tr>
<tr>
<td>Software, 2015</td>
<td>23</td>
</tr>
<tr>
<td>Software, 2016</td>
<td>18</td>
</tr>
</tbody>
</table>
```

Daniele Apiletti
Data Base and Data Mining group
MapReduce example (3a)

- Map - Ordered list of students, with mark and CFU for each exam
  
  ```javascript
  Function(doc) {
    return new reduce({
      key: doc.student,
      value: [doc.mark, doc.CFU]
    });
  }
  ```

- Reduce - Average CFU-weighted mark for each student
  
  ```javascript
  AVG = \frac{\sum X \times Y}{\sum Y}; \quad \text{where} \quad X = \text{mark}, \ Y = \text{CFU}
  ```

```
<table>
<thead>
<tr>
<th>DocId</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S123456</td>
<td>[25, 8]</td>
</tr>
<tr>
<td>2</td>
<td>S123456</td>
<td>[21, 8]</td>
</tr>
<tr>
<td>3</td>
<td>S654321</td>
<td>[24, 10]</td>
</tr>
<tr>
<td>4</td>
<td>S654321</td>
<td>[29, 8]</td>
</tr>
<tr>
<td>5</td>
<td>S123456</td>
<td>[18, 8]</td>
</tr>
<tr>
<td>6</td>
<td>S123456</td>
<td>[27, 10]</td>
</tr>
<tr>
<td>7</td>
<td>S654321</td>
<td>[25, 8]</td>
</tr>
<tr>
<td>8</td>
<td>S654321</td>
<td>[23, 9]</td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>DocId</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S123456</td>
<td>[25, 8]</td>
</tr>
<tr>
<td>2</td>
<td>S123456</td>
<td>[21, 8]</td>
</tr>
<tr>
<td>3</td>
<td>S654321</td>
<td>[24, 10]</td>
</tr>
<tr>
<td>4</td>
<td>S654321</td>
<td>[29, 8]</td>
</tr>
<tr>
<td>5</td>
<td>S123456</td>
<td>[18, 8]</td>
</tr>
<tr>
<td>6</td>
<td>S123456</td>
<td>[27, 10]</td>
</tr>
<tr>
<td>7</td>
<td>S654321</td>
<td>[25, 8]</td>
</tr>
<tr>
<td>8</td>
<td>S654321</td>
<td>[23, 9]</td>
</tr>
</tbody>
</table>
```

Replication

- **Same** data
  - in **different** places (content and schema)
  ```javascript
  function(doc) {
    this.emit(doc.student, doc);
  }
  ```

- **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
- Faster and unreliable

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
- In 2002 a formal proof was published, establishing it as a theorem
- In 2002 a formal proof was published, establishing it as a theorem
- In 2012, a follow-up by Eric Brewer, “CAP twelve years later: How the “rules” have changed”
The 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.

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 sync.
- 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 preserves 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 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.

## HTTP API

A «web» database, no ad-hoc client required.

<table>
<thead>
<tr>
<th>HTTP API</th>
<th>HTTP RESTful API</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to get a document? Use your browser and write its URL</td>
<td>How to get a document? Use your browser and write its URL</td>
</tr>
<tr>
<td>Any application and language can access web data</td>
<td>Any application and language can access web data</td>
</tr>
<tr>
<td>GET /somedatabase/some_doc_id</td>
<td>GET /somedatabase/some_doc_id</td>
</tr>
<tr>
<td>HEAD /somedatabase/some_doc_id</td>
<td>HEAD /somedatabase/some_doc_id</td>
</tr>
<tr>
<td>HTTP/1.0</td>
<td>HTTP/1.0</td>
</tr>
<tr>
<td>HTTP/1.1 200 OK</td>
<td>HTTP/1.1 200 OK</td>
</tr>
<tr>
<td>Write a document by means of PUT HTTP request (specify revision to avoid conflicts)</td>
<td>Write a document by means of PUT HTTP request (specify revision to avoid conflicts)</td>
</tr>
<tr>
<td>PUT /somedatabase/some_doc_id</td>
<td>PUT /somedatabase/some_doc_id</td>
</tr>
<tr>
<td>HTTP/1.1 201 Created</td>
<td>HTTP/1.1 201 Created</td>
</tr>
<tr>
<td>HTTP/1.1 409 Conflict</td>
<td>HTTP/1.1 409 Conflict</td>
</tr>
</tbody>
</table>

## HTTP RESTful API

- Delete a document
  - DELETE /somedatabase/some_doc_id?rev=1582603387
  - HTTP/1.0 200 OK
- Parametric multi-document fetch
  - GET /somedatabase/_all_docs?startkey=doc2&endkey=doc3
  - GET /somedatabase/_all_docs?startkey=doc2&endkey=doc3&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 - intro

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

![MongoDB Features](http://www.slideshare.net/mongodb/introduction-to-mongodb-56807822)

MongoDB - why

- Why MongoDB?
  - JSON
  - Easy to find
  - No Schemes: “No DBA”, Just Database
  - Schema: SAVVY database on virtual real-time
  - Fast: Similar to MRO's
  - Easiest to setup a scalable
  - Storage: Out of the Box
  - URI: Great for that
  - No Schema: None, no downside to create new columns
  - Base: Based on MongoDB

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

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
- Lists, sub-documents, etc.
- BUT
- Relations among documents / records are inefficient, and leads to de-normalization
- ObjectID 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”**

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

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$.
- increase Paul's account balance by 100$.

```json
{   "id": "account_123456",   "account": "bank_account_001",   "balance": 900,   "timestamp": 1290678353.45,   "categories": ["bankTransfer"]}
```

```json
{   "id": "account_654321",   "account": "bank_account_002",   "balance": 1100,   "timestamp": 1290678353.46,   "categories": ["bankTransfer"]}
```

- What if some kind of failure occurs between the two separate updates to the two accounts?
- How do we read the current account balance?

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.

```javascript
function(transaction001) {   emit(transaction001.from, transaction001.amount*-1);   emit(transaction001.to, transaction001.amount); }

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

Result

```json
{   "rows": [     {       "key": "bank_account_001",       "value": 900     },     {       "key": "bank_account_002",       "value": 1100     }   ]}
```
Beyond relational databases

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

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