Beyond relational databases

Daniele Apiletti
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

- **no joins**
- **schema-less** (no tables, implicit schema)
  
<table>
<thead>
<tr>
<th>Student ID</th>
<th>Name</th>
<th>Surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>S123456</td>
<td>Mario</td>
<td>Rossi</td>
</tr>
</tbody>
</table>

**horizontal scalability**

## Comparison

<table>
<thead>
<tr>
<th>Relational databases</th>
<th>Non-Relational databases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Table</strong>-based, each record is a structured row</td>
<td><strong>Specialized storage solutions</strong>, e.g., document-based, key-value pairs, graph databases, columnar storage</td>
</tr>
<tr>
<td>Predefined <strong>schema</strong> for each table, changes allowed but usually blocking (expensive in distributed and live environments)</td>
<td><strong>Schema-less</strong>, schema-free, schema change is dynamic for each document, suitable for semi-structured or <strong>un-structured data</strong></td>
</tr>
<tr>
<td><strong>Vertically</strong> scalable, i.e., typically scaled by increasing the power of the hardware</td>
<td><strong>Horizontally</strong> scalable, NoSQL databases are scaled by increasing the databases servers in the pool of resources to reduce the load</td>
</tr>
<tr>
<td>Use <strong>SQL</strong> (Structured Query Language) for defining and manipulating the data, very powerful</td>
<td><strong>Custom query</strong> languages, focused on collection of documents, graphs, and other specialized data structures</td>
</tr>
<tr>
<td>Relational databases</td>
<td>Non-Relational databases</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Suitable for <strong>complex queries</strong>, based on data <strong>joins</strong></td>
<td><strong>No standard</strong> interfaces to perform complex queries, <strong>no joins</strong></td>
</tr>
<tr>
<td>Suitable for <strong>flat</strong> and structured data storage</td>
<td>Suitable for complex (e.g., <strong>hierarchical</strong>) data, similar to JSON and XML</td>
</tr>
<tr>
<td>Examples: MySql, <strong>Oracle</strong>, Sqlite, Postgres and Microsoft SQL Server</td>
<td>Examples: <strong>MongoDB</strong>, BigTable, Redis, Cassandra, Hbase and CouchDB</td>
</tr>
</tbody>
</table>
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
  • 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 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.
a notable NoSQL example

CouchDB
Cluster Of Unreliable Commodity Hardware
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 environment that allows HTTP requests.

CouchDB original home page
CouchDB original home page

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

• Published in 2004 by Google
  • 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 functional programming
    • In functional code, the output value of a function depends only on the arguments that are passed to the function, so calling a function \( f \) twice with the same value for an argument \( x \) produces the same result \( f(x) \) each time; this is in contrast to procedures depending on a local or global state, which may produce different results at different times when called with the same arguments but a different program state.
MapReduce: working principles

• Consists of two functions, a **Map** and a **Reduce**
  • The Reduce is optional

• **Map** function
  • Process each record (document) → INPUT
  • Return a list of **key-value** pairs → OUTPUT

• **Reduce** function
  • for each **key**, reduces the list of its **values**, returned by the map, to a “single” value
  • Returned value can be a complex piece of data, e.g., a list, tuple, etc.
Map

• Map functions are called once for each document:
  
  ```javascript
  function(doc) {
    emit(key1, value1); // key1 = f_k1(doc); value1 = f_v1(doc)
    emit(key2, value2); // key2 = f_k2(doc); value2 = f_v2(doc)
  }
  ```

• The map function can choose to skip the document altogether or emit one or more 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

<table>
<thead>
<tr>
<th>Id: 1</th>
<th>Exam: Database</th>
<th>Student: s123456</th>
<th>AYear: 2015-16</th>
<th>Date: 31-01-2016</th>
<th>Mark=29</th>
<th>CFU=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id: 2</td>
<td>Exam: Computer architectures</td>
<td>Student: s123456</td>
<td>AYear: 2015-16</td>
<td>Date: 03-07-2015</td>
<td>Mark=24</td>
<td>CFU=10</td>
</tr>
<tr>
<td>Id: 3</td>
<td>Exam: Computer architectures</td>
<td>Student: s654321</td>
<td>AYear: 2015-16</td>
<td>Date: 26-01-2016</td>
<td>Mark=27</td>
<td>CFU=10</td>
</tr>
<tr>
<td>Id: 4</td>
<td>Exam: Database</td>
<td>Student: s654321</td>
<td>AYear: 2014-15</td>
<td>Date: 26-07-2015</td>
<td>Mark=26</td>
<td>CFU=8</td>
</tr>
<tr>
<td>Id: 5</td>
<td>Exam: Software engineering</td>
<td>Student: s123456</td>
<td>AYear: 2014-15</td>
<td>Date: 14-02-2015</td>
<td>Mark=21</td>
<td>CFU=8</td>
</tr>
<tr>
<td>Id: 6</td>
<td>Exam: Bioinformatics</td>
<td>Student: s123456</td>
<td>AYear: 2015-16</td>
<td>Date: 18-09-2016</td>
<td>Mark=30</td>
<td>CFU=6</td>
</tr>
<tr>
<td>Id: 7</td>
<td>Exam: Software engineering</td>
<td>Student: s654321</td>
<td>AYear: 2015-16</td>
<td>Date: 28-06-2016</td>
<td>Mark=18</td>
<td>CFU=8</td>
</tr>
</tbody>
</table>
• List of exams and corresponding marks

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

Result:

<table>
<thead>
<tr>
<th>id</th>
<th>Exam</th>
<th>Student</th>
<th>AYear</th>
<th>Date</th>
<th>Mark</th>
<th>CFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Database</td>
<td>s123456</td>
<td>2015-16</td>
<td>31-01-2016</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Computer architectures</td>
<td>s654321</td>
<td>2015-16</td>
<td>03-07-2015</td>
<td>24</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures</td>
<td>s654321</td>
<td>2015-16</td>
<td>26-01-2016</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Database</td>
<td>s654321</td>
<td>2014-15</td>
<td>26-07-2015</td>
<td>26</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Software engineering</td>
<td>s123456</td>
<td>2014-15</td>
<td>14-02-2015</td>
<td>21</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Bioinformatics</td>
<td>s123456</td>
<td>2015-16</td>
<td>18-09-2016</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Software engineering</td>
<td>s654321</td>
<td>2015-16</td>
<td>28-06-2015</td>
<td>18</td>
<td>8</td>
</tr>
</tbody>
</table>
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);
}
```

Result:

<table>
<thead>
<tr>
<th>doc.id</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>[Bioinformatics, 2015-16]</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>[Computer architectures, 2015-16]</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>[Computer architectures, 2015-16]</td>
<td>27</td>
</tr>
<tr>
<td>1</td>
<td>[Database, 2015-16]</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>[Software engineering, 2015-16]</td>
<td>18</td>
</tr>
</tbody>
</table>
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);
}

Result:

<table>
<thead>
<tr>
<th>Id</th>
<th>Exam</th>
<th>Student</th>
<th>AYear</th>
<th>Date</th>
<th>Mark</th>
<th>CFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Database</td>
<td>s123456</td>
<td>2015-16</td>
<td>31-01-2016</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Computer architectures</td>
<td>s123456</td>
<td>2015-16</td>
<td>03-07-2015</td>
<td>24</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures</td>
<td>s654321</td>
<td>2015-16</td>
<td>26-01-2016</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Database</td>
<td>s654321</td>
<td>2014-15</td>
<td>26-07-2015</td>
<td>26</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Software engineering</td>
<td>s123456</td>
<td>2014-15</td>
<td>14-02-2015</td>
<td>21</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Bioinformatics</td>
<td>s123456</td>
<td>2015-16</td>
<td>18-09-2016</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Software engineering</td>
<td>s654321</td>
<td>2014-15</td>
<td>28-06-2015</td>
<td>18</td>
<td>8</td>
</tr>
</tbody>
</table>
Reduce

• Documents (key-value pairs) 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 execution of the reduce function returns 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 in CouchDB
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;
  }

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

<table>
<thead>
<tr>
<th>doc.id</th>
<th>Key</th>
<th>Value</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Bioinformatics</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Computer architectures</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Database</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Database</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Database</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Software engineering</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Software engineering</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bioinformatics</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Computer architectures</td>
<td>25.5</td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td>26.67</td>
<td></td>
</tr>
<tr>
<td>Software engineering</td>
<td>19.5</td>
<td></td>
</tr>
</tbody>
</table>
MapReduce example (2)

- **Map** - List of exams and corresponding mark
  ```java
  Function(doc){
    emit(
      [doc.exam, doc.AYear],
      doc.mark
    );
  }
  ```

- **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;
  }
  ```

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

### Table: Map

<table>
<thead>
<tr>
<th>doc.id</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Bioinformatics, 2015-16</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>Computer architectures, 2015-16</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures, 2015-16</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Database, 2014-15</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Database, 2014-15</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>Database, 2015-16</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>Software engineering, 2014-15</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Software engineering, 2015-16</td>
<td>18</td>
</tr>
</tbody>
</table>

### Table: Reduce

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Bioinformatics, 2015-16]</td>
<td>30</td>
</tr>
<tr>
<td>[Computer architectures, 2015-16]</td>
<td>25.5</td>
</tr>
<tr>
<td>[Database, 2015-16]</td>
<td>29</td>
</tr>
<tr>
<td>[Software engineering, 2015-16]</td>
<td>18</td>
</tr>
</tbody>
</table>

Reduce is the same as before
Rereduce in CouchDB

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

<table>
<thead>
<tr>
<th>Id</th>
<th>Exam</th>
<th>Student</th>
<th>AYear</th>
<th>Date</th>
<th>Mark</th>
<th>CFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Database</td>
<td>s123456</td>
<td>2015-16</td>
<td>31-01-2016</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Computer architectures, 2015-16</td>
<td>s123456</td>
<td>2015-16</td>
<td>3-07-2015</td>
<td>24</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures, 2015-16</td>
<td>s123456</td>
<td>2015-16</td>
<td>3-07-2015</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Database, 2014-1015</td>
<td>s123456</td>
<td>2014-15</td>
<td>14-02-2015</td>
<td>30</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Bioinformatics, 2015-16</td>
<td>s123456</td>
<td>2015-16</td>
<td>18-09-2016</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Software engineering, 2015-16</td>
<td>s654321</td>
<td>2015-16</td>
<td>28-06-2016</td>
<td>18</td>
<td>8</td>
</tr>
</tbody>
</table>

Map

<table>
<thead>
<tr>
<th>doc.id</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Bioinformatics, 2015-16</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>Computer architectures, 2015-16</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>Computer architectures, 2015-16</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Database, 2014-1015</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Database, 2014-15</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>Database, 2015-16</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>Software engineering, 2014-15</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Software engineering, 2015-16</td>
<td>18</td>
</tr>
</tbody>
</table>

Reduce

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Bioinformatics, 2015-16]</td>
<td>30</td>
</tr>
<tr>
<td>[Computer architectures, 2015-16]</td>
<td>25.5</td>
</tr>
<tr>
<td>[Database, 2015-16]</td>
<td>29</td>
</tr>
<tr>
<td>[Software engineering, 2015-16]</td>
<td>18</td>
</tr>
</tbody>
</table>
MapReduce example (3a)

Average CFU-weighted mark for each student

Map

The reduce function receives:
- \texttt{key} = \texttt{doc.student}
- \texttt{value} = \texttt{[doc.mark, doc.CFU]}
- \texttt{emit(key, value)}

Reduce

The reduce function results:
- \texttt{key} = \texttt{id}
- \texttt{values} = \texttt{...}
- \texttt{key} = \texttt{...}
- \texttt{values} = \texttt{...}

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

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

<table>
<thead>
<tr>
<th>doc.id</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S123456</td>
<td>[29, 8]</td>
</tr>
<tr>
<td>2</td>
<td>S123456</td>
<td>[24, 10]</td>
</tr>
<tr>
<td>5</td>
<td>S123456</td>
<td>[21, 8]</td>
</tr>
<tr>
<td>6</td>
<td>S123456</td>
<td>[30, 6]</td>
</tr>
<tr>
<td>3</td>
<td>S654321</td>
<td>[27, 10]</td>
</tr>
<tr>
<td>4</td>
<td>S654321</td>
<td>[26, 8]</td>
</tr>
<tr>
<td>7</td>
<td>S654321</td>
<td>[18, 8]</td>
</tr>
<tr>
<td>8</td>
<td>s987654</td>
<td>[25, 8]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>S123456</td>
<td>25.6</td>
</tr>
<tr>
<td>S654321</td>
<td>23.9</td>
</tr>
<tr>
<td>s987654</td>
<td>25</td>
</tr>
</tbody>
</table>
MapReduce example (3b)

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

<table>
<thead>
<tr>
<th>DB</th>
<th>Map</th>
<th>Reduce</th>
<th>Rereduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>doc.id</td>
<td>Key</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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**
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 **unreliable**

---

![Diagram of master and slaves with replication arrows]

Master

Can commit other transactions

Replicate

Slave  Slave  Slave  Slave  ……
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
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 2012, a follow-up by Eric Brewer, “CAP twelve years later: How the "rules" have changed”

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

Data Models
- Relational (Comparison)
- Key-value
- Column-oriented/Tabular
- Document oriented

Availability
- Each client can always read and write

Consistency
- All clients always have the same view of the data

Partition Tolerance
- The system works well despite physical network partitions

Pick 2

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.
• A system is allowed to \textit{not} answer requests at all (turn off “A”).
• We claim to tolerate \textit{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 \textit{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 \textit{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 \textit{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
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
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
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
**Why MongoDB?**

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

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

[Link](https://blog.cloudbust.io/why-you-should-not-use-only-mongodb/)
Hadoop

The de facto standard
Big Data platform
Hadoop, a Big-Data-everything platform

- **2003**: Google File System
- **2004**: MapReduce by Google (Jeff Dean)
- **2005**: Hadoop, funded by Yahoo, to power a search engine project
- **2006**: Hadoop migrated to Apache Software Foundation
- **2006**: Google BigTable
- **2008**: Hadoop wins the Terabyte Sort Benchmark, sorted 1 Terabyte of data in 209 seconds, previous record was 297 seconds
- **2009**: additional components and sub-projects started to be added to the Hadoop platform
Hadoop, platform overview
Hadoop, platform overview
Hadoop, platform overview
Hadoop, platform overview
Apache Hadoop, core components

• **Hadoop Common**: The common utilities that support the other Hadoop modules.

• **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.

• **Hadoop YARN**: A framework for job scheduling and cluster resource management.

• **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.
Hadoop-related projects at Apache

- **Ambari™**: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters which includes support for Hadoop HDFS, Hadoop MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig and Sqoop. Ambari also provides a dashboard for viewing cluster health such as heatmaps and ability to view MapReduce, Pig and Hive applications visually along with features to diagnose their performance characteristics in a user-friendly manner.

- **Avro™**: A data serialization system.

- **Cassandra™**: A scalable multi-master database with no single points of failure.

- **Chukwa™**: A data collection system for managing large distributed systems.

- **HBase™**: A scalable, distributed database that supports structured data storage for large tables.

- **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying.

- **Mahout™**: A Scalable machine learning and data mining library.

- **Pig™**: A high-level data-flow language and execution framework for parallel computation.

- **Spark™**: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.

- **Tez™**: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute an arbitrary DAG of tasks to process data for both batch and interactive use-cases. Tez is being adopted by Hive™, Pig™ and other frameworks in the Hadoop ecosystem, and also by other commercial software (e.g. ETL tools), to replace Hadoop™ MapReduce as the underlying execution engine.

- **ZooKeeper™**: A high-performance coordination service for distributed applications.
Apache Spark

• A fast and general engine for large-scale data processing

• Speed
  • Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
  • Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.

• Ease of Use
  • Write applications quickly in Java, Scala, Python, R.
  • Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.

• Generality
  • Combine SQL, streaming, and complex analytics.
  • Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.

• Runs Everywhere
  • Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.
Hadoop - why

• **Storage**
  • distributed,
  • fault-tolerant,
  • heterogenous,
  • Huge-data storage engine.

• **Processing**
  • Flexible (multi-purpose),
  • parallel and scalable,
  • high-level programming (Java, Python, Scala, R),
  • batch and real-time, historical and streaming data processing,
  • complex modeling and basic KPI analytics.

• **High availability**
  • Handle failures of nodes by design.

• **High scalability**
  • Grow by adding low-cost nodes, not by replacement with higher-powered computers.

• **Low cost.**
  • Lots of commodity-hardware nodes instead of expensive super-power computers.
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$
  ```json
  {
    _id: "account_123456",
    account:"bank_account_001",
    balance: 900,
    timestamp: 1290678353.45,
    categories: ["bankTransfer"...],
    ...
  }
  ```

- increase Paul's account balance by 100$
  ```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?

- Decrease your account balance by 100$.
- Increase Paul’s account balance by 100$.
Design recipe: banking example

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

- Decrease your account balance by 100$
- Increase Paul’s account balance by 100$

Bank

Message lost during transmission
Design recipe: banking example

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

  decrease your account balance by 100$
  increase Paul’s account balance by 100$
  Message lost during transmission

• 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
  ```javascript
  function(transaction){
    emit(transaction.from, transaction.amount*-1);
    emit(transaction.to, transaction.amount);
  }
  ```
• Reduce
  ```javascript
  function(key, values){
    return sum(values);
  }
  ```
• Result
  ```javascript
  {rows: [ {key: "bank_account_001", value: 900} ]
  {rows: [ {key: "bank_account_002", value: 1100} ]
  ```
Beyond relational databases

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