# Développement logiciel pour le Cloud (TLC)

## 5. NoSQL data models


Guillaume Pierre

Université de Rennes 1


Fall 2012

http://www.globule.org/~gpierre/

## Table of Contents


1. Introduction
2. NoSQL data models
3. Data modeling techniques
4. Complex queries thank to MapReduce




## Table of Contents


1. Introduction
2. NoSQL data models
3. Data modeling techniques
4. Complex queries thank to MapReduce




Stop following me, you fucking freaks!

Key-Value

Ordered Key-Value

Big Table

Document, Full-Text Search

Graph

SQL

## Data modeling for NoSQL datastores

- Data normalization techniques will not work for NoSQL
  - Forget UML and other related methodologies
- There is very little formal work on data schema design for NoSQL :-(
  - NoSQL is too young for that
  - Each NoSQL datastore has specific features
- But there exists **useful guidelines**
  - Keeping in mind that **each NoSQL datastore has specific functionality**
  - Exploit them to the fullest extent!



## Table of Contents

1 Introduction
2 NoSQL data models
3 Data modeling techniques
4 Complex queries thank to MapReduce



## Different types of NoSQL datastores

- **Key-value stores** do not attempt to interpret the content of values
  - PUT(key,value)
  - value=GET(key)
  - DELETE(key)
  - Examples: AppEngine's datastore, HBase, AWS Dynamo
- **Ordered key-value stores** let you iterate through keys
  - Examples: Scalarix
- **Document databases** do interpret the content of values
  - Impose a syntax for values (JSON, XML, etc.)
  - Support value-based operations (e.g., secondary-key queries)
    - With various performance behaviors depending on the database
  - Example: CouchDB, Apache Cassandra
- More exotic types of data stores: graph databases, object databases, etc.



## Common properties

Let's compare **Amazon's SimpleDB**, **Google's BigTable** and **Yahoo's PNUTS**

- Data are organized in **tables**
- A table contains a number of **data items** identified by a **primary key**
- Data items are organized as a collection of **key-value pairs**
  - Only data type: **string**
  - Data items from the same table **do not necessarily have the same list of attributes** (flexible data schema)
- Data items are accessed by PUT/GET using their primary key
- No supported operation across tables (such as joins)

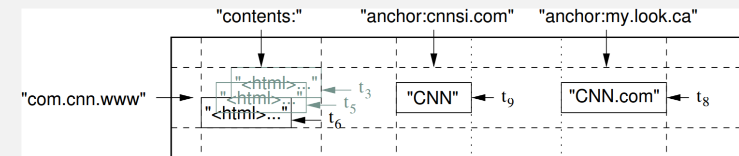
## Amazon's SimpleDB / Apache's Cassandra

- SimpleDB allows records to contain **multiple values with the same key** (e.g., a multiset)
- Data are organized into "domains"
  - Domains ∼ tables
  - No schema
- SimpleDB supports range queries
- Consistency: **eventual consistency**
  - Also some form of strong consistency is supported (with lower levels of performance)



## Google's BigTable / Apache's HBase

- Columns are organized in **column families**: `"family:column_name"`
  - Column families are the granularity for access control
- Tables have more dimensions than the standard model
  - Values are indexed by row, column **and timestamp**
  - (row:string, column:string, time:int64) → string




- Rows are sorted
  - BigTable allows users to **iterate** through records
  - . . . or through successive versions of the same record



## Yahoo's PNUTS

- PNUTS requires an **explicit list of attributes per record** (i.e., a schema)
  - But it is not necessary to use all attributes
  - And it is easy to change the list at runtime
- UPDATE, DELETE and INSERT queries must specify a primary key
- Tables can be **hashed** or **ordered**
  - Hashed: excellent load balancing, efficient primary-key queries
  - Ordered: less good load balancing, but support for **range queries**
  - In both cases: PNUTS supports "multiget" queries to retrieve several records in parallel (from one or more tables)
- Consistency: single-row transactions



## Comparison

| | **Amazon's SimpleDB** | **Google's Bigtable** | **Yahoo's PNUTS** |
|---|---|---|---|
| **Data Item** | Multi-value attribute | Multi-version with timestamp | Multi-version with timestamp |
| **Schema** | No schema | Column-families | Explicitly claimed attributes |
| **Operation** | Range queries on arbitrary attributes of a table | Single-table scan with various filtering conditions | Single-table scan with predicates |
| **Consistency** | Eventual consistency | Single-row transaction | Single-row transaction |

## Table of Contents


1. Introduction
2. NoSQL data models
3. Data modeling techniques
4. Complex queries thank to MapReduce




## Denormalization

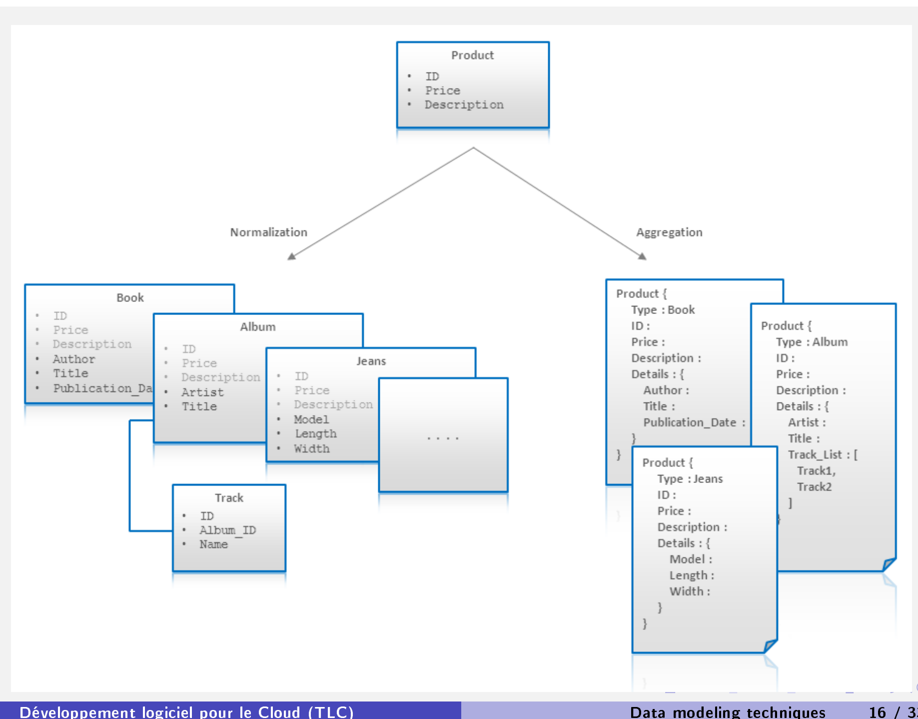
- Normalization defines **data stuctures regardless of the queries**
  - Hidden assumption: if the data are well-organized we can always query them easily
  - This is true for SQL databases but not for NoSQL datastores
- Denormalization does the opposite of normalization: **structure data according to future queries**
  - Group all data necessary for a query at the same place
  - We often end up copying the same data at multiple places in the datastore
  - 🙂 Excellent performance if we do things well
  - 🙁 Database consistency issues: all updates must be applied everywhere, it is easy to introduce mistakes



## Aggregates

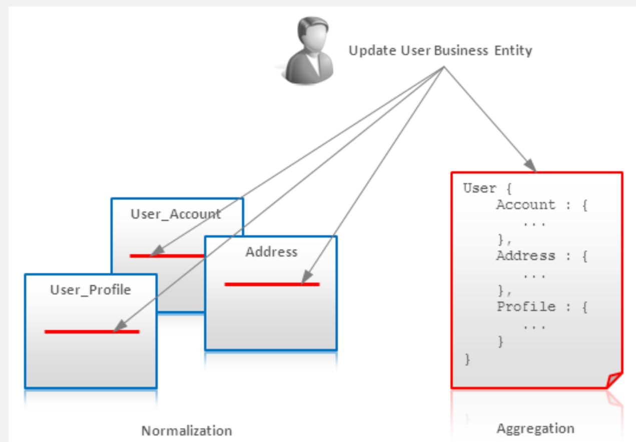
- NoSQL datastores allow **flexible data schemas**
  - Stored values may have complex nested structures
  - No need to pre-define these structures, we can simply create them at runtime
  - Each record may have a different structure
- Example 1: a `User` record links to the list of his `Messages`
  - Normalized version: two tables (`Users` and `Messages`) with **references** between the two
  - NoSQL version: insert the entire messages **inside** the User record
- Example 2: different types of products
  - Normalized verson: one table for each type of product (with its specific structure)
  - NoSQL version: store all products with their specificities next to eqch other





## Atomic aggregates

- Aggregates have one nice side-effect: **atomic updates**
  - NoSQL datastores often support atomic updates per data item
  - But they rarely support multi-item transactions
- If multiple updates are located in the same record they become atomic



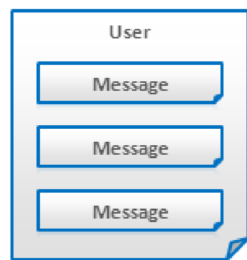



## Application-side joins

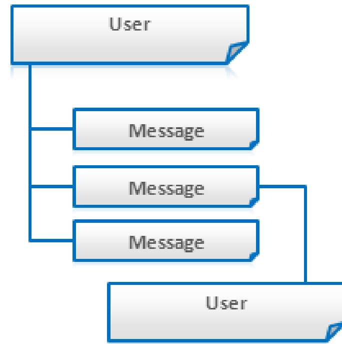
- Very few NoSQL data stores support joins
  - Denormalization and aggregates often allow us to avoid joins
- But sometimes we cannot avoid joins
  - Many-to-many relationships between records
  - Frequently updated data items
- Solution: application-side joins
  - Let the application fetch all necessary data items
  - Join them by hand



Aggregates




Joins



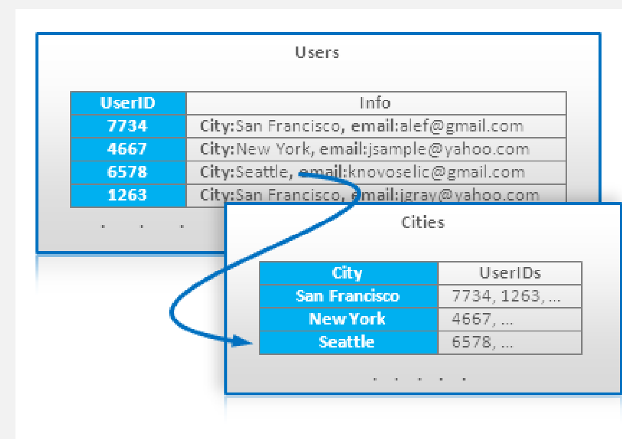

Static
One-To-Many

Dynamic
Many-To-Many



## Index tables

- We can implement foreign keys by simplyu building index tables
  - Replace one join query with 2 simple queries
  - Beware: you lose atomicity



# Enumerable keys

- DHTs normally hash keys before deciding where to store each data item
  - Excellent for load balancing
  - But contiguous keys end up being located in random nodes in the system
- Some NoSQL decided to drop hashing
  - Much less efficient for laod balancing
  - But it allows applications to **iterate through keys**
- You can embed information in the keys
  - Example: **key=*userID_messageID***
  - You can easily access all messages from a user: start at UserID_0 and iterate

# Composite key index

- We can combine index tables with fancy key structures
  - This often allows for efficient secondary-key queries
- Example: **select users by their location**
  - `SELECT * FROM users WHERE state="CA"`
  - `SELECT * FROM users WHERE city="San Francisco"`
  - NoSQL solution: design keys as **State:City:UserID**

| State:City:UserID | |
|---|---|
| AR:Little Rock:543211 | Values |
| CA:Los Angeles:211123 | Values |
| CA:Los Angeles:456546 | Values |
| CA:Oakland:666634 | Values |
| CA:San Francisco:756322 | Values |
| CA:San Francisco:972321 | Values |
| CA:San Francisco:972321 | Values |
| CO:Denver:972321 | Values |

# Aggregation with Composite Keys

- We can also use composite keys for data aggregation
- Example: search a log file for **all unique sites visited by a user**
  - `SELECT count(distinct(user_id)) FROM clicks GROUP BY site`
  - NoSQL solution: make sure to **keep contiguous log records per user**
    - And then eliminate redundancy in the application itself

| UserID : EventID | Site |
|---|---|
| 543211:324235 | t-mobile.co.uk |
| 623229:232773 | google.com |
| 623229:345444 | webehigh.com |
| 623229:562333 | sf-police.org |
| 623229:979949 | google.com |
| 883398:345436 | mongodb.org |



- This is much more efficient than keeping log entries from each user in a single record

# Inverted search

- If we want to search items along multiple criteria we cannot use composite keys
  - With composite keys we can support only one type of search
- Example: we want to search users by their gender, city, the sites they visit etc.
  - NoSQL solution: build inverted indexes explicitly
  - Key=property; Value=reference to the main table

Event: UserID, Category, Site, City

Category-Men: [ UserID1, UserID2, ... ]
Category-Women: [ UserID1, UserID2, ... ]
Category-Kids: [ UserID1, UserID2, ... ]
. . . .
City-NY: [ UserID1, UserID2, ... ]
City-SF: [ UserID1, UserID2, ... ]
. . . .
Site-google.com: [ UserID1, UserID2, ... ]
Site-facebook.com: [ UserID1, UserID2, ... ]
. . . .

Inverted Index

UserID: Category-Men, City-NY, Site-google.com
UserID: Category-Women, City-SF, Site-ya.ru
. . . .

Direct Index

Query: Describe google.com audience from NY or SF → (City-SF OR City-NY) AND (Site-google.com) → [ UserID1, UserID2, ... ] →

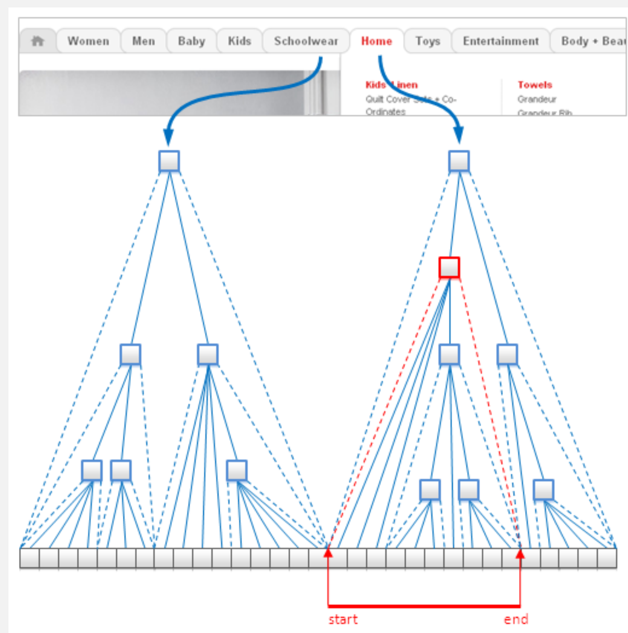
Report:
Category-Men : 20,010 unique users
Category-Women : 21,310 unique users
.....



# Nested sets

- How do we represent a hierarchical structure in NoSQL?
  - Bad solution #1: store the entire tree in one data item
  - Bad solution #2: store each node separately, maintain a list of children in all non-leaf nodes
- Solution: nested sets
  - Map each leaf to one data item in the NoSQL store
  - Make each non-leaf node maintain the beginning/end index
    - Very efficient for read/search
    - Not so efficient for updates








# Table of Contents

1 Introduction
2 NoSQL data models
3 Data modeling techniques
4 Complex queries thank to MapReduce

## Using MapReduce for complex queries

- Some queries can be **unfrequent** but **very complex**
  - E.g., data mining queries
- You cannot redesign your entire data schema for just one ad-hoc query
- Implementing the entire query in the application can be inefficient
  - In the worst case: fetch the entire data store on the client, let the client process the query locally
- Solution: **MapReduce**
  - Example: MongoDB is fully integrated with MapReduce
  - You can request a MapReduce job over the content of the datastore in just one command



## MapReduce queries in MongoDB

```
db.runCommand(
 { mapreduce : <collection>,
   map : <mapfunction>,
   reduce : <reducefunction>,
   out : <see output options below>
   [, query : <query filter object>]
   [, sort : <sorts the input objects using this key. Useful for optimization, like sorting by
the emit key for fewer reduces>]
   [, limit : <number of objects to return from collection, not supported with sharding>]
   [, keeptemp: <true|false>]
   [, finalize : <finalizefunction>]
   [, scope : <object where fields go into javascript global scope >]
   [, jsMode : true]
   [, verbose : true]
 }
);
```



## Example [1/2]

```
$ ./mongo
> db.things.insert( { _id : 1, tags : ['dog', 'cat'] } );
> db.things.insert( { _id : 2, tags : ['cat'] } );
> db.things.insert( { _id : 3, tags : ['mouse', 'cat', 'dog'] } );
> db.things.insert( { _id : 4, tags : []  } );

> // map function
> m = function(){
...    this.tags.forEach(
...        function(z){
...            emit( z , { count : 1 } );
...        }
...    );
...};

> // reduce function
> r = function( key , values ){
...    var total = 0;
...    for ( var i=0; i<values.length; i++ )
...        total += values[i].count;
...    return { count : total };
...};
```



## Example [2/2]

```
> res = db.things.mapReduce(m, r, { out : "myoutput" } );
> res
{
        "result" : "myoutput",
        "timeMillis" : 12,
        "counts" : {
                "input" : 4,
                "emit" : 6,
                "output" : 3
        },
        "ok" : 1,
}
> db.myoutput.find()
{"_id" : "cat" , "value" : {"count" : 3}}
{"_id" : "dog" , "value" : {"count" : 2}}
{"_id" : "mouse" , "value" : {"count" : 1}}

> db.myoutput.drop()
```

## Conclusion

- NoSQL datastores are designed for scalability
  - Even at the cost of reducing the set of offered functionalities
- Different NoSQL data stores can have very different properties
  - It is important to understand these specific functionalities to make the best use of each system
  - Also useful for choosing one datastore (when possible)
- Very little theoretical background on how to organize data
  - But there exists useful guidelines