NoSQL data stores and SOS:

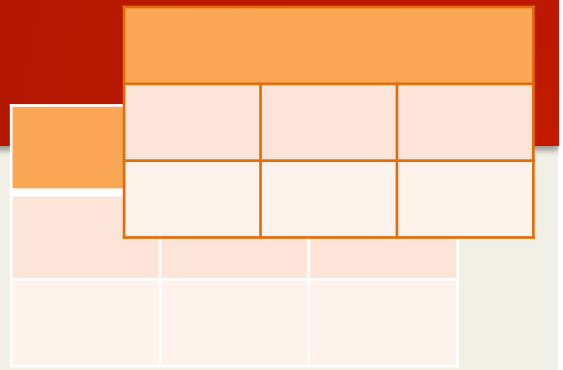
Uniform Access to Non-Relational Database Systems

Paolo Atzeni – Francesca Bugiotti – Luca Rossi

Outline

- **Context**
  - Relational DBMS
  - NoSQL Data Stores
  - NoSQL Timeline
- **NoSQL Data Stores**
  - Extensible Record Stores
  - Document Stores
  - Key-Value Stores
  - Rise of NoSQL: a survey
  - *"NoSQL is about choice"*
  - Heterogeneity
- **SOS - Save Our Systems**
  - Goal and requirements
  - Data Model
  - Interface
  - Architecture
  - Translation techniques
- **Future Work**

Relational DBMS



Relational Databases

- Provide efficient support for applications that:
  - Require perstistence, consistence, avaliability, usability, ...
  - Simple data structure (numeric, string, …)
  - Complex queries expressed by declarative languages

Relational DBMS

(Stonebraker & Cattel, CACM June 2011)

- "General-purpose traditional row stores”
  - Disk-oriented storage
  - Tables stored row-by-row on disk (hence, a row store)
  - B-trees as the indexing mechanism
  - Dynamic locking as the concurrency control mechanism
  - A write-ahead log (WAL) for crash recovery
  - SQL as the access language
  - A "row-oriented" query optimizer and executor

Relational DBMS

- Relational databases proposed as universal solution
  - It isn't completely true
    - OLAP, OLTP, XML , Stream processing, …

One size does fit all?

(Michael Stonebraker, Ugur Çetintemel: "One Size Fits All": An Idea Whose Time Has Come and Gone. ICDE 2005: 2-11)

Why NoSQL?

- **Web apps changed the game**
  - Many **users**
  - Many concurrent **interactions** (reads and writes)
  - Lots of **data**.

- **Need for:**
  - **Scalable, distributed** storage systems
  - **Flexible** "*web-proof*" data models
  - **Easy** interaction with programming languages

NoSQL Data Stores

- **Characteristics**
  - High scalability
  - Data replication according to a distributed architecture
  - Flexible data structure
  - New indexing patterns
- **Missing features**
  - Simple interface
  - New approach towards consistency

Consistency

- CAP Theorem: A distributed system cannot satisfy all the following properties:
  - Consistency
  - Availability
  - Partition-tolerance

- New Consistency Approaches
  - Strong consistency
  - Weak consistency
  - Eventual consistency

NoSQL Timeline

- **2006** – *Google BigTable*
- **2007** – *Amazon Dynamo*

- **2007** – HBase
- **2008** – Cassandra
- **2009** – Voldemort, Redis, Riak, MongoDB

  …

- **2011** – Oracle NoSQL

Extensible Record Stores

- **Extensible Record Stores (also called column-family stores)**
  - Relaxation of the Relational Model
  - Store tables of extensible records
  - Partitioned across multiple nodes

- **2006** – *Google BigTable*
- **2007** – *Amazon Dynamo*
- **2007** – HBase

HBase

- **HBase:**
  - **Database** made of **Tables**, containing **Rows** identified by a unique **Id**.
  - Columns (named **qualifiers**) within a table are grouped into **Column Families** (CFs).
  - Tables and CFs are **static**, defined in advance.
  - Qualifiers are **dynamic**, can be added/removed at runtime.
  - Within the same CF, different rows can have different qualifiers.

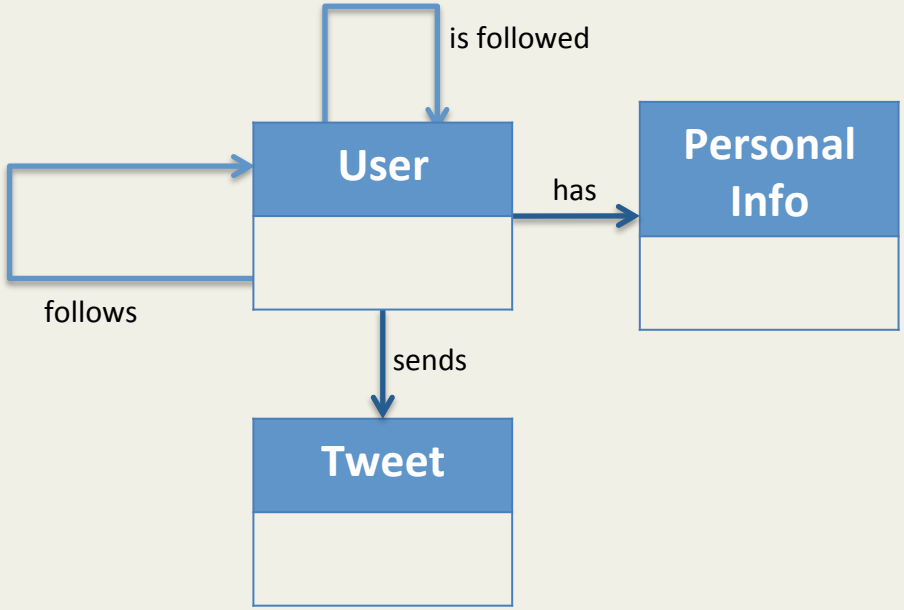
Row id	*Column families*			
	ColumnFamily1	**ColumnFamily2**	**...**	**ColumnFamilyN**
row_id_1	qualifier1 = "value1" qualifier2 = "value2"	qualifier1 = "value1"	...	...
row_id_2				
...				
row_id_m				

HBase

Row id	*Column families*			
	ColumnFamily1	**ColumnFamily2**	**...**	**ColumnFamilyN**
row_id_1	qualifier1 = "value1" qualifier2 = "value2"	qualifier1 = "value1"	...	...
row_id_2				
...				
row_id_m				

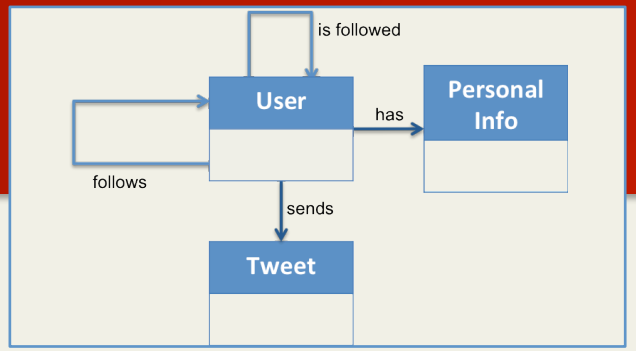
- **HBase:**
  - **Database** made of **Tables**, containing **Rows** identified by a unique **Id**.
  - Columns (named **qualifiers**) within a table are grouped into **Column Families** (CFs).
  - Tables and CFs are **static**, defined in advance.
  - Qualifiers are **dynamic**, can be added/removed at runtime.
  - Within the same CF, different rows can have different qualifiers.
- **Unique data type:**
  - Byte-array
- **Unstructured data stored in a semi-structured environment**
  - Some assumptions are needed

Example



- **Twitter Example**
  - Users, Tweets, Personal Info
  - Users follows other users and are followed themself
  - Users have personal info
  - Users send tweets

Example



Row id	Column families		
	User Data	Personal Info	Tweet
1001	Username = “Alice” Password = TheAlicePassword	FirstName = “Alice” LastName = “Smith” …	…
1002	Username = “Bob” Password = TheBobPassword	…	…
…	…		

HBase

- **Operations**
  - Work on sigle rows or on lists of rows
  - Provide direct access to rows given the row key
    - get(key)
    - put(key)
    - delete(key), deleteColumn(key, column), deleteFamily(…)
    - add(key, columnFamily), add(key, columnFamily, qualifier, value), …
    - scan(table)
  - Rows selection on the basis of filters defined on column families or qualifiers
  - …

HBase

- **Other characteristics**
  - Map-Reduce support (Hadoop)
  - Strong consistency
  - Max 10 column families
  - Using filters deteriorates performances
  - Bloom filters and column family compression for more efficient indexes

DynamoDB

- **DynamoDB**
  - **Database** made of **Tables**, containing **Items** identified by a unique **key**
  - Items group a set of **Attributes**
  - Attributes are characterized by a **Name** and a **Value**
  - Every attribute can have multiple values
  - Different Items belonging to the same table can have sets of disjoints attributes

Table

Key	Other attributes
key	($Name_1$, value), …, ($Name_1$, value) ($Name_3$, value)
key	($Name_2$, value)
…	…

DynamoDB

Other attributes
($Name_1$, value), …, ($Name_1$, value) ($Name_3$, value)
($Name_2$, value)
…

- **DynamoDB**
  - **Database** made of **Tables**, containing **Items** identified by a unique **key**
  - Items group a set of **Attributes**
  - Attributes are characterized by a **Name** and a **Value**
  - Every attribute can have multiple values
  - Different Items belonging to the same table can have sets of disjoints attributes
- **Provided on the cloud**
- **Data types**
  - Scalar data types: number, string, binary
  - Set data types: number set, string set, binary set

Example

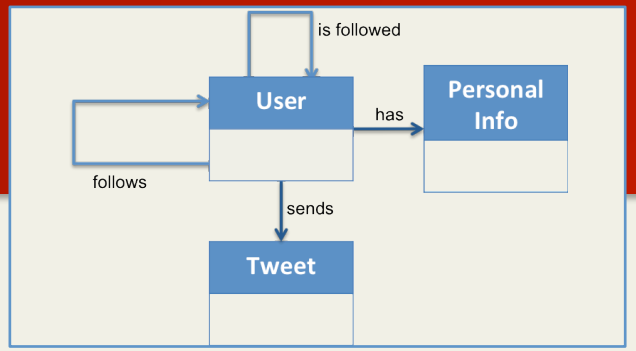


Table: Users

Key	Other attributes
1001	Username = “Alice”, Password = TheAlicePassword, FirstName = “Alice”, …
1002	Username = “Bob”, Password = TheBobPassword, …
…	…

DynamoDB

- **Operations**
  - work on sigle items
    - getItem(table, key)
    - putItem(table, key, av)
    - deleteItem(table, key)
  - row selection on the basis of filters that use attribute names
  - batch operations
    - batch putItem
    - batch deleteItem,
  - ...

DynamoDB

- **Other characteristics**
  - High avaliability
  - Keys are hashed: databases can be seen as a distributed hash table
  - Node are located in specific regions (specified when the datastore is created)
  - Data are replicated across nodes
  - DynamoDB access cost policy
  - Eventual consistency/strong consistenct

Document Stores

Document Stores

– Store collection of documents

– Documents are objects characterized by fields whose value can be a scalar, a list, a document itself.

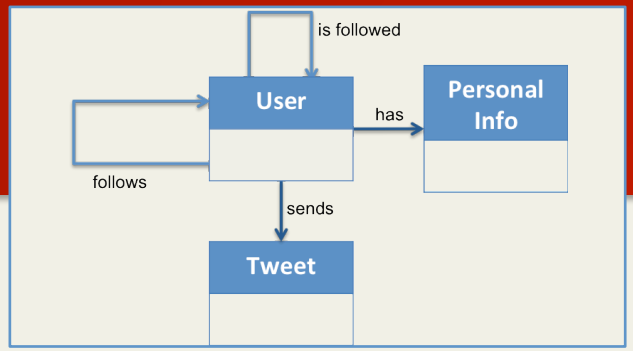
- **2009** – MongoDB

…

MongoDB

- **MongoDB**
  - A **Database** is made of **Collections**
  - A collection is a named group of **Documents**
  - Documents are made of **fields**
  - Fields value can be a scalar, a list, a document itself.

MongoDB



Users Collection

```
users: [
    {
        _id: "1001",
        username: "bob1987",
        password: ”TheBobPassword"
        personal: {
          firstName: "Bob",
          lastName: "Smith",
          ssn: "4hfe94"
        },
        followers: [
            {
                id: "2004",
                firstName: "Alice",
                lastName: "Smith",
                email: "alice@gmail.com"
            },
            {
                id: "1714",
...
```

MongoDB

- **Operations**
  - Operations defined on single fields:
    - insert(collection, doc)
    - find(selector, collection)
    - remove(selector, collection)
  - Advanced operations

- The simplest selector is the empty document {} that matches all the documents of a collection.

MongoDB

- **Other characteristics**
  - Full index support
  - Rich query API
  - Sharding
  - Strong consistency

Key-value datastores

Key-value data datastores

– Store values and an index for finding them based on programmer-defined key.

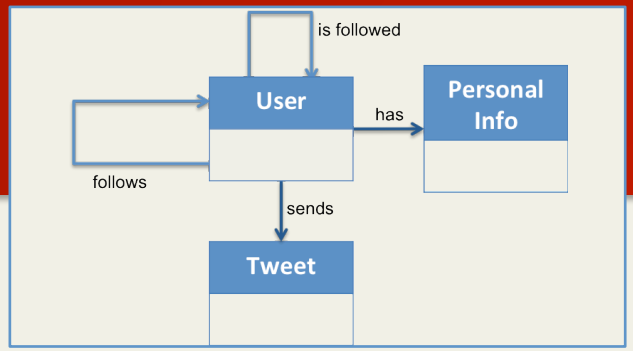
– A database is a collection of key-value pairs.

- **2009** –Redis
- **2011** – Oracle NoSQL
- …

Redis

- **Redis**
  - A **Database** is a schema-less collection of key-values pairs
  - Key-value index
- **Data types**
  - Binary Strings: any type of binary data (byte array, number, plain string, …).
  - Integer counters
  - Lists, Sets of binary strings
  - Hashes
- **Complex data types cannot be nested**
  - It is not possible to define Sets of Lists, …
- **Unique key-space**

Redis




```
users:1001:firstName = "Bob"
users:1001:lastName = "Smith”


users:1001 = {
   username = "bob1987"
   password = ” TheBobPassword"
   friends:2004.email = "alice@gmail.com"
   friends:2004.firstName = "Alice"
   ...
   }
```

Redis

- **Operations**
  - Simple operations:
    - set(key, value)
    - get(key)
    - delete(key)
  - Advanced operations
    - Insert an element into a list or a set
    - Increment a counter
    - hgetall(key) that retrieves all the field-value pairs of a hash associated with the key

Redis

- **Other characteristics**
  - Efficient access (in memory)
  - Map-Reduce support
  - Strong consistency

NoSQL Data Stores

- **Aspects to be considered**
  - Number of accesses to retrieve an object
  - Resilience to unstructuredness
  - Partition-friendliness
  - How data are supposed to be queried

- **Datastore best practices**

- **Performances influence data organization:**
  - Denormalize data or not? Always? Never? When?
  - How we want to support consistency?

Rise of NoSQL: a survey

- 50% of IT managers/developers funded NoSQL projects in 2011
- 70% plan to fund NoSQL projects in 2012
- Enterprises in U.S.:
  - 56% already use some NoSQL database
  - 63% has plans to use in the next 2 years

[1] http://www.infoq.com/news/2012/02/NoSQL-Adoption-Is-on-the-Rise
[2] http://www.prweb.com/releases/2011/6/prweb8609164.htm

“NoSQL is about choice”

- Jan Lenhardt (Couchbase Co-Founder)

- **Many data model families**
  - *Key-value* store
  - *Column-based* store
  - *Document* store
  - *Graph* store
- **Many query models**
  - CRUD operations
  - Map/Reduce queries
  - Custom query languages
  - Traversals
- **Many architectural choices**
  - Replicas (DHT?) vs sharding
  - In-RAM vs traditional storage
  - AP vs CP vs CA
  - Strong vs eventual consistency
  - …

“NoSQL is about choice” (2)

- Jan Lenhardt (Couchbase Co-Founder)

- **Choose the right tool for your needs**
  - One size does not fit all
- Do you need Map/Reduce?
  - Pick HBase or CouchDB
- Do you need great performances on simple operations?
  - Pick Redis
- **Chances are you may need both**

Heterogeneity still a problem

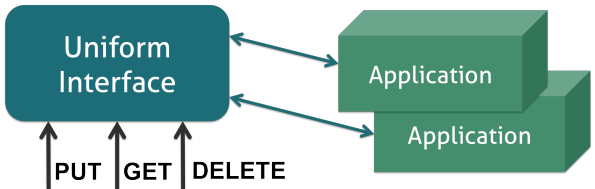
- **What if:**
  - I want to use many data stores at the same time
  - I want to migrate my data
  - I want to decouple my app from a specific technology
- **Reverse the canonical problem:**
  - One size (data store) does not fit all (apps)…
  - …but one size (your app) should fit all (the data stores)

SOS – Save our systems

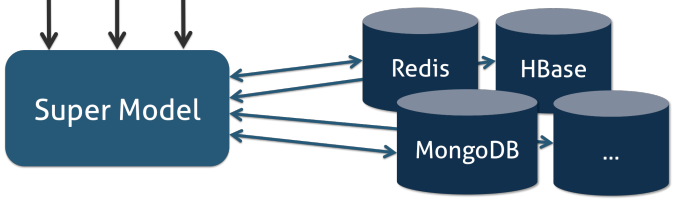
- **Goal:** seamless access to different NoSQL data stores.
  - Define *access*
  - Define *seamless*

- **Requirements:**
  - Lightweight: small footprint on performances
  - Coherent: with main NoSQL themes and features
    - Hint: do not reimplement SQL
      - Seriously, someone has done it
  - Scalable: easily extendable to different technologies and data stores

SOS – Save our systems



Common Interface
to access different NoSQL systems

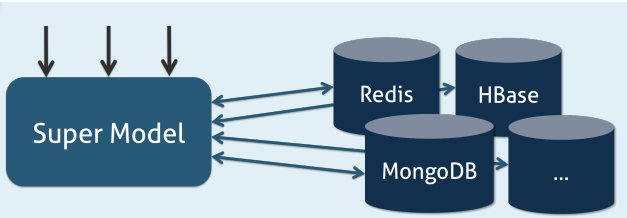


Common Data Model
instances are mapped to the data stores of choice

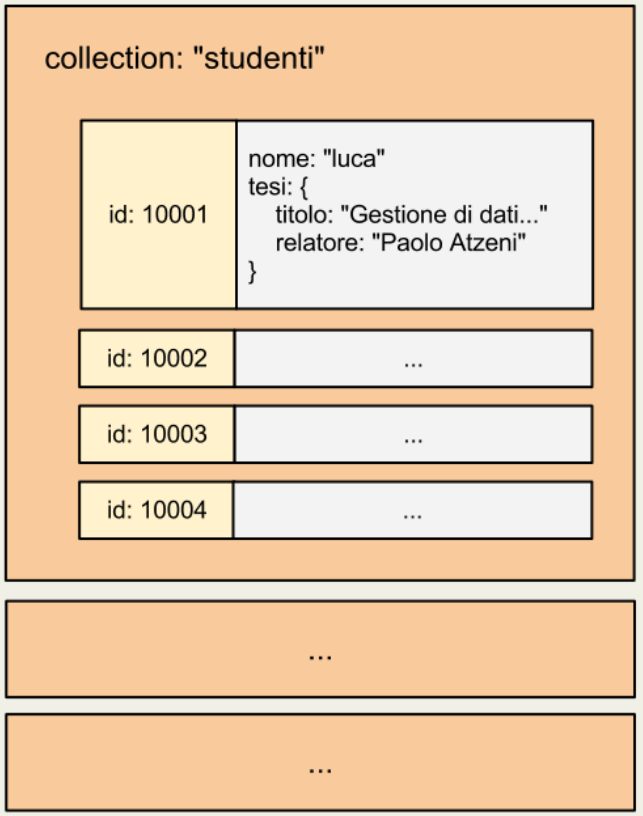
- **SOS is a *Database Access Layer* between the app and the data store**
  - It collects data from the interface and seamlessly manages its translation and deployment to specific data stores
- **Implementations provided for three data stores belonging to different families:**
  - HBase (column-based store)
  - Redis (key-value store)
  - MongoDB (document store)

Common Data Model

SOS – Save our Systems

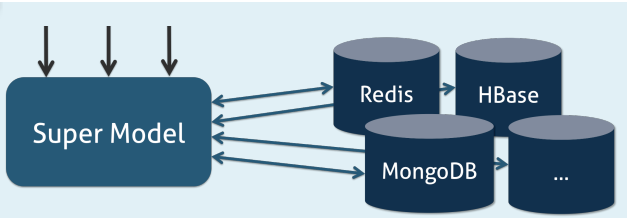


- **SOS** let users define collections of schemaless, tree-shaped objects
- **Each collection** is identified by a unique name
- **Each object** is identified by an ID, unique within the collection it belongs to

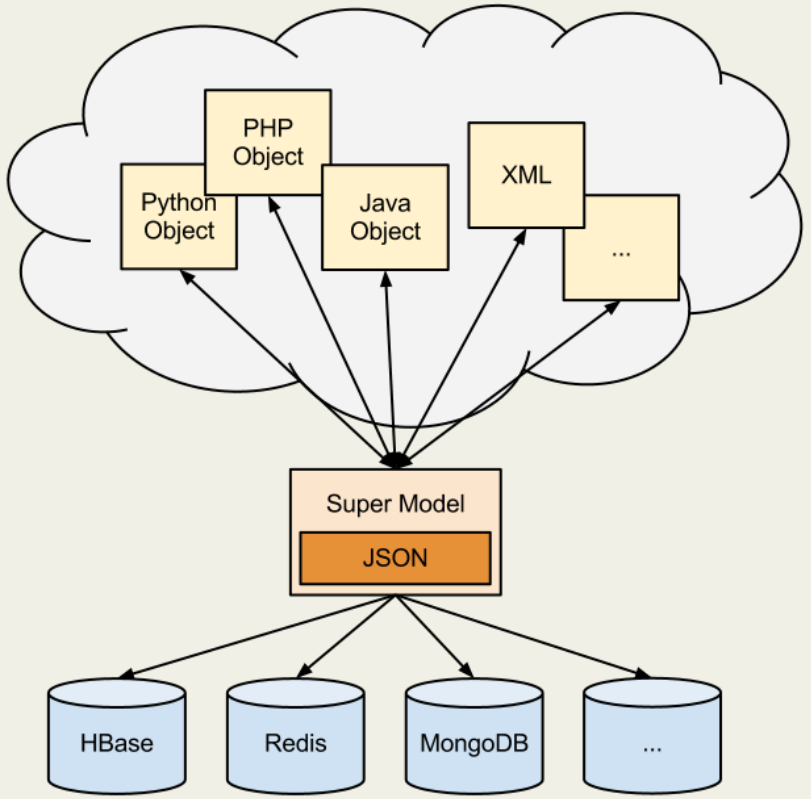


Common Data Model (2)

SOS – Save our Systems

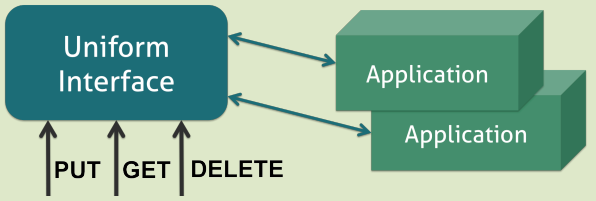


- Objects are materialized in **JSON** (**J**ava**S**cript **O**bject **N**otation)
  - Lightweight
  - Widely adopted
  - Platform-independent
- **Custom translations** are defined between JSON and each data store
- Translations are **optimized** to exploit efficiently the data store native structures



Common Interface

SOS – Save our Systems

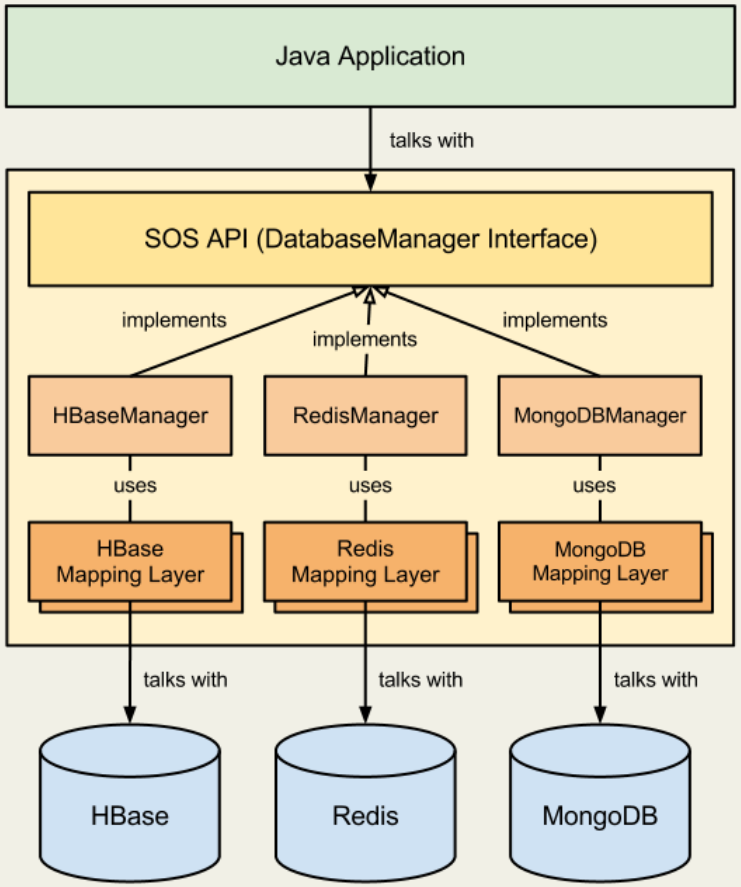


- **Operations on single objects:**
  - **put** (collection-name, id, object) : void
  - **get** (collection-name, id, type<T>) : <T>
  - **delete**(collection-name, id) : void
- **Operations on single fields:**
  - **put** (collection-name, id, path, object) : void
  - **get** (collection-name, id, path, type<T>) : <T>
  - **delete** (collection-name, id, path) : void
- **Operations on collections:**
  - **get** (collection-name, type<T>) : Collection<T>
  - **delete** (collection-name) : void

Architecture

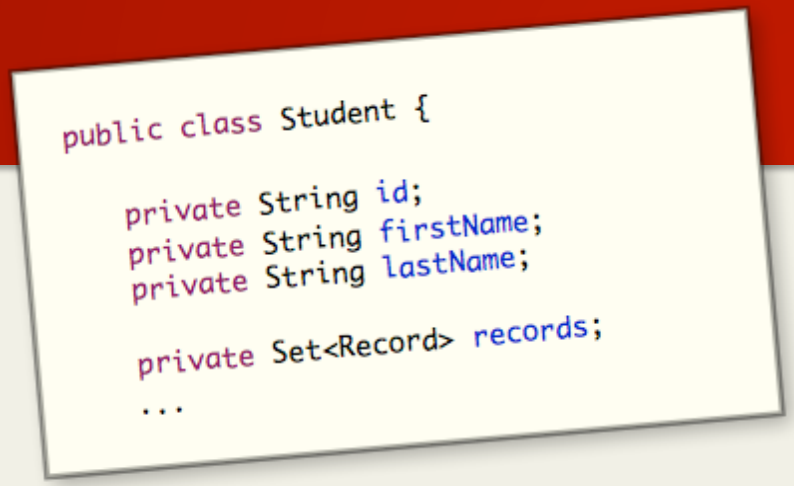
SOS – Save our Systems

- SOS is currently implemented as a **Java library**.
- It defines a **streamlined API** implemented by specific data store modules.



Usage example

SOS – Save our Systems



```
public class Student {

    private String id;
    private String firstName;
    private String lastName;

    private Set<Record> records;
    ...
```

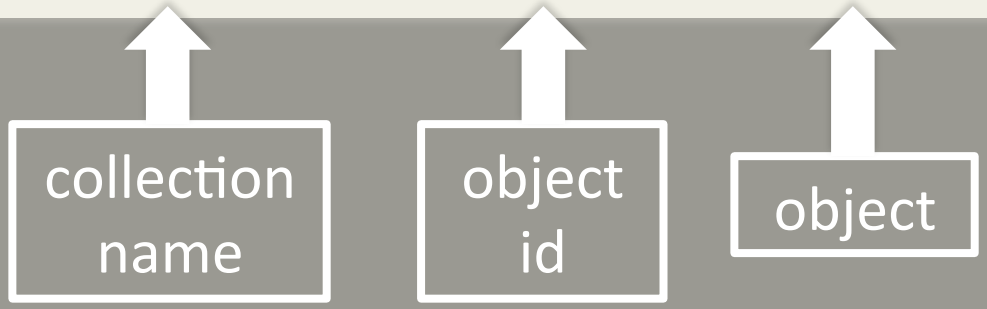
```
Student luca = new Student(...);

DatabaseHandler db = new HBaseHandler();

db.put("students", luca.getId(), luca);
```

Usage example

SOS – Save our Systems

```
public class Student {

    private String id;
    private String firstName;
    private String lastName;

    private Set<Record> records;
    ...
```

```
Student luca = new Student(...);

DatabaseHandler db = new HBaseHandler();

db.put("students", luca.getId(), luca);
```

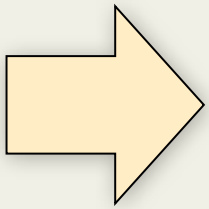


Translation example

SOS – Save our Systems

```java
public class Student {

    private String id;
    private String firstName;
    private String lastName;

    private Set<Record> records;
    ...
```

Java



```
{
    id = ”281283”,
    firstName = “Luca”,
    lastName = ”Rossi”,
    records = [
        {
            id = ”10001”,
            course = {
                id = ”20001”,
                name = ”Databases 101”
            };
            date = ”2011/06/12”,
            grade = “A”
        },
        {
            id = ”10002”,
            course = {
                id = ”20004”,
                name = ”Computer Vision”,
            };
            date = ”2011/05/21”,
            grade = “B”
        }
    ]
}
```

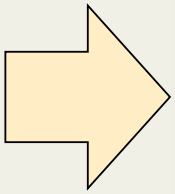
JSON

Translation example

SOS – Save our Systems

```
{
    id = ”281283”,
    firstName = “Luca”,
    lastName = ”Rossi”,
    records = [
        {
            id = ”10001”,
            course = {
                id = ”20001”,
                name = ”Databases 101”
            };
            date = ”2011/06/12”,
            grade = “A”
        },
        {
            id = ”10002”,
            course = {
                id = ”20004”,
                name = ”Computer Vision”,
            };
            date = ”2011/05/21”,
            grade = “B”
        }
    ]
}
```

JSON



students (table)

_top	records[]
id = “281283” firstName = “Luca” lastName = “Rossi”	[0].id = “10001” [0].course.id = “20001” [0].course.name = “Databases 101” [0].date = “2011/06/12” [0].grade = “A” [1].id = “10001” [1].course.id = “20004” [1].course.name = “Computer Vision” [1].date = “2011/05/21” [1].grade = “B”

HBase

Future work

SOS – Save our Systems

- **Architecture:**
  - Deploy SOS as a web application, exposing a REST interface that deals with JSON objects.
- **Translations:**
  - Enable custom translations, providing ways for the users to map structures into others, and so forth.
- **Interface:**
  - Provide support for the creation (and manteinance) of indexes
  - Provide support for "update" operations that involve multiple nodes
- **Support for other DBMSes**
  - Amazon DynamoDB and Oracle NoSQL are underway!

Outline

- **Context**
  - Relational DBMS
  - NoSQL Data Stores
  - NoSQL Timeline
- **NoSQL Data Stores**
  - Extensible Record Stores
  - Document Stores
  - Key-Value Stores
  - Rise of NoSQL: a survey
  - *"NoSQL is about choice"*
  - Heterogeneity
- **SOS - Save Our Systems**
  - Goal and requirements
  - Data Model
  - Interface
  - Architecture
  - Translation techniques
- **Future Work**

Questions?