NoSQL Database Design for Next-Generation Web Applications

Joint work with Francesca Bugiotti, Paolo Atzeni, and Riccardo Torlone

- References

- F. Bugiotti, L. Cabibbo, P. Atzeni, R. Torlone
  A Logical Approach to NoSQL Databases
  Submitted for publication

- F. Bugiotti, L. Cabibbo
  An Object-Datastore Mapper Supporting NoSQL Database Design
  Submitted for publication
- Outline

- NoSQL databases systems
- NoSQL database design for next-generation web applications
- NoAM (NoSQL Abstract-Model)
  - overview
  - aggregates and aggregate design
  - an abstract data model for NoSQL databases
  - basic data representations
  - aggregate partitioning
  - a language for data representations
  - implementation
  - conclusion
- ONDM (Object-NoSQL Datastore Mapper)
  - architecture
  - conclusion
- A case study in NoSQL database design

- Introduction

- A promise of NoSQL database technologies
  - support the development of *next-generation web applications*
    - characterized by good horizontal scalability, high availability, and good response time – data have a flexible structure – simple read-write operations
  - in this context, we are interested in *NoSQL database design*
NoSQL database systems

- **NoSQL datastores** are a new generation of distributed database systems
  - they have been designed to support the needs of an increasing number of modern applications – for which traditional database technology is unsatisfactory
  - foremost requirements of next-generation web applications
    - horizontal scalability – the ability to manage large data sets distributed over many servers
    - high availability and good response time
    - data of interest have a flexible structure
    - data access is based on simple read-write operations
    - a certain degree of consistency is required – but consistency guarantees can be relaxed

NoSQL product landscape
The NoSQL landscape is characterized by a high heterogeneity:
- they have different data models and different APIs to access the data – as well as different consistency and durability guarantees

We focus here on three main categories of NoSQL databases:
- key-value stores
  - a database is a collection of key-value pairs
- document stores
  - a database is a collection of documents
- extensible record stores
  - data is organized as tables of extensible records
  - these categories include more than 70 systems

In a **key-value store**, a database is a schema-less collection of **key-value pairs**:
- **values** are usually binary strings, opaque to the datastore – even if some systems have interpreted values, such as counters, lists, or hashes
- programmer-defined **keys** are either binary strings or structured keys – in some systems, part of the key is used to control data distribution
- simple data access operations – put, get, and delete – over an individual key-value pair or a group of related key-value pairs
**Key-value stores: examples**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player:mary</td>
<td>username: mary, firstName: Mary, lastName: Wilson, games: [...]</td>
</tr>
<tr>
<td>Player:rick</td>
<td>username: rick, firstName: Ricky, lastName: Doe, score: 42, games: [...]</td>
</tr>
</tbody>
</table>

```
Player:mary
  /username mary
  /firstName Mary
  /lastName Wilson
  /games [...]

Player:rick
  /username rick
  /firstName Ricky
  /lastName Doe
  /score 42
  /games [...]
```

- In a **document store**, a database is a set of documents:
  - each **document** has a complex value and an identifier
  - documents are composed of **fields**, which are dynamically defined for each document at runtime – each field can be a scalar value, a list, or a document itself
  - documents are organized in **collections**
  - the structure of documents is not opaque to datastores – they create indexes on documents and support content-based querying
### Document stores: example

**collection Player**

<table>
<thead>
<tr>
<th>id</th>
<th>document</th>
</tr>
</thead>
</table>

---

### Extensible record stores

- An **extensible record** (or *column-family*) store organizes data around **tables, records/rows, and columns**

  - A relaxation of the relational model, in which databases are mostly schema-less – since each row can have its own set of columns
  
  - Each table designates a **primary key** – which comprises the only mandatory attributes of the table – in some systems, part of the primary key is used to control data distribution
### NoSQL Database Design

We consider here **NoSQL database design** – the problem of representing persistent data of an application in a target NoSQL database:

- NoSQL databases are claimed to be “schema-less”
- however, the data of interest do show some structure, and decisions on the organization of data are required
  - specifically, to map application data to the modeling elements (collections, tables, documents, key-value pairs) available in the target datastore

#### Extensible record stores: example

<table>
<thead>
<tr>
<th>username</th>
<th>firstName</th>
<th>lastName</th>
<th>score</th>
<th>games[0]</th>
<th>games[1]</th>
<th>games[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mary</td>
<td>Mary</td>
<td>Wilson</td>
<td></td>
<td>{...}</td>
<td>{...}</td>
<td></td>
</tr>
<tr>
<td>rick</td>
<td>Ricky</td>
<td>Doe</td>
<td>42</td>
<td>{...}</td>
<td>{...}</td>
<td>{...}</td>
</tr>
</tbody>
</table>
Consider a fictitious online, web 2.0 game – which should manage various application objects, including players, games, rounds, and moves.

- Assume for example that the target database is an extensible record store.
- What records (and tables) should we use?
  - A distinct record for each different application object?
  - Or should we use each record to represent a group of related objects? What is the grouping criterion?
- What columns should we use?
  - A distinct column for each object field?
  - Or should we use each column to represent a group of related fields? What is the grouping criterion?
In NoSQL database design

- decisions on the organization of data are required, in any case
- these decision are significant – as the data representation affects major quality requirements – such as scalability, performance, and consistency
- a randomly chosen data representation may not satisfy the needed qualities
- how should we make design decisions to indeed support the qualities of next-generation web applications?

Next-generation web applications

- We focus here on database design for next-generation web applications – also called scalable web applications, or scalable simple OLTP-style applications
  - foremost requirements of these applications
    - data of interest are large and have a flexible structure
    - data access is based on simple read-write operations
    - horizontal scalability – data should be distributed over a cluster of many servers
    - high availability and good response time
    - relaxed consistency guarantees – general ACID transactions are typically unnecessary – however, a certain degree of consistency is required, to easy application development – e.g., eventual consistency or BASE
State-of-the-art in NoSQL database design

- a lot of best practices and guidelines
  - but usually related to a specific datastore or class of datastores
- neither a systematic methodology nor a high-level data model
  - as in the case of relational database design
- however
  - “the availability of a high-level representation of the data at hand, be it logical or conceptual, remains a fundamental tool for developers and users, since it makes understanding, managing, accessing, and integrating information sources much easier, independently of the technologies used”

---

NoAM (NoSQL Abstract Model)

- NoAM (NoSQL Abstract Model) is a logical approach to NoSQL database design
  - NoAM is based on an intermediate, abstract data model – which aims at exploiting the commonalities of the various NoSQL datastores
  - a logical approach – the initial design activities are independent of any specific target systems
    - a NoAM abstract database is first used to represent the application data
    - the intermediate representation is then implemented in a target NoSQL datastore, taking into account its specific features
The NoAM approach to NoSQL database design is based on the following main phases:

- **aggregate design** — to identify the various classes of aggregate objects needed in the application
  - this activity is driven by use cases (functional requirements) and scalability and consistency needs
- **aggregate partitioning** — aggregates are partitioned into smaller data elements
  - driven by use cases and performance requirements
- **high-level NoSQL database design** — aggregate are mapped to the NoAM intermediate data model
- **implementation** — to map the intermediate representation to the specific modeling elements of the target datastore

---

**Application data**

We start by considering application data objects...

![Diagram of application data objects and their relationships]
... we group them in aggregates ...

Player:mary:

  username : "mary",
  firstName : "Mary",
  lastName : "Wilson",
  games : {
    ( game : Game:2345, opponent : Player:rick ),
    ( game : Game:2611, opponent : Player:ann )
  }

Player:rick:

  username : "rick",
  firstName : "Ricky",
  lastName : "Doe",
  score : 42,
  games : {
    ( game : Game:2345, opponent : Player:mary ),
    ( game : Game:7425, opponent : Player:ann ),
    ( game : Game:1241, opponent : Player:johnny )
  }

Game:2345:

  id : "2345",
  firstPlayer : Player:mary,
  secondPlayer : Player:rick,
  rounds : {
    ( moves : ... , comments : ... ),
    ( moves : ... , actions : ... , spell : ... )
  }
… we partition these complex values…

```json
Player:mary: {
  username: "mary",
  firstName: "Mary",
  lastName: "Wilson",
  games: {
    game: Game:2345, opponent: Player:rick
    game: Game:2611, opponent: Player:ann
  }
}

Player:rick: {
  username: "rick",
  firstName: "Ricky",
  lastName: "Doe",
  score: 42,
  games: {
    game: Game:2345, opponent: Player:mary
    game: Game:7425, opponent: Player:ann
    game: Game:1241, opponent: Player:johnny
  }
}

Game:2345: {
  id: "2345",
  firstPlayer: Player:mary,
  secondPlayer: Player:rick,
  rounds: {
    moves: ... , comments: ... 
    moves: ... , actions: ... , spell: ...
  }
}
```

… and represent them into an abstract data model for NoSQL databases…

```json
Player:mary: {
  username: "mary",
  firstName: "Mary",
  lastName: "Wilson",
  games: {
    game: Game:2345, opponent: Player:rick
    game: Game:2611, opponent: Player:ann
  }
}

Player:rick: {
  username: "rick",
  firstName: "Ricky",
  lastName: "Doe",
  score: 42,
  games: {
    game: Game:2345, opponent: Player:mary
    game: Game:7425, opponent: Player:ann
    game: Game:1241, opponent: Player:johnny
  }
}

Game:2345: {
  id: "2345",
  firstPlayer: Player:mary,
  secondPlayer: Player:rick,
  rounds: {
    moves: ... , comments: ... 
    moves: ... , actions: ... , spell: ...
  }
}
```
… and finally we map the intermediate representation to the data structures of the target datastore

**table Player**

<table>
<thead>
<tr>
<th>username</th>
<th>firstName</th>
<th>lastName</th>
<th>score</th>
<th>games[0]</th>
<th>games[1]</th>
<th>games[2]</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>mary</td>
<td>Mary</td>
<td>Wilson</td>
<td></td>
<td>{...}</td>
<td>{...}</td>
<td>{...}</td>
<td></td>
</tr>
<tr>
<td>rick</td>
<td>Ricky</td>
<td>Doe</td>
<td>42</td>
<td>{...}</td>
<td>{...}</td>
<td>{...}</td>
<td></td>
</tr>
</tbody>
</table>

**table Game**

<table>
<thead>
<tr>
<th>id</th>
<th>firstPlayer</th>
<th>secondPlayer</th>
<th>rounds[0]</th>
<th>rounds[1]</th>
<th>rounds[2]</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
<td>Player:mary</td>
<td>Player:rick</td>
<td>{...}</td>
<td>{...}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

key value

- `/Player/mary/-/username` mary
- `/Player/mary/-/firstName` Mary
- `/Player/mary/-/lastName` Wilson
- `/Player/mary/-/games[0]` { “game” : “Game:2345”, “opponent” : “Player:rick” }
- ...
In our approach, we consider application data arranged in *aggregates*

- the notion of aggregate comes from Domain-Driven Design (DDD) – a popular object-oriented design methodology – and from principles in the design of scalable applications

The design of scalable applications is discussed in a seminal paper by Pat Helland – *Life beyond distributed transactions: an apostate’s opinion*, CIDR 2007

- data should be organized as a set of complex-value objects with unique identifiers – called *entities* or *aggregates* – each aggregate is a “chunk” of related data, and is intended to be a *unit of data access and manipulation*

- aggregates should govern *data distribution* – aggregates are distributed among the nodes of the cluster, but each aggregate is located on a single node

- atomic transactions can not span multiple aggregates – to avoid the coordination overhead required by distributed transactions

- operations spanning multiple aggregates should be implemented as multiple operations, each over a single aggregate – using asynchronous messages and eventual consistency
Domain-Driven Design (DDD, Eric Evans, 2003) is also based on a similar notion of aggregate – DDD gives us other insights on aggregate design:

- Each **aggregate** is a group of application objects (entities and value objects) rooted in an entity.
  - An entity is a persistence object that has independent existence and a unique identifier.
  - A value object is a persistent object, without an own identifier.
- Aggregate boundaries govern distribution and transactions.
- Each aggregate should be large enough, to accommodate all the data involved by some integrity constraints or other business rules.
- Aggregates should be as small as possible, to reduce concurrency collisions – to support performance and scalability requirements.

Example

- Aggregates in our running example are players and games.
  - But rounds are not – to support an integrity constraint of the game.
Data organization at the application level

- an **application dataset** is a set of classes
- the extent of a **class** is a set of aggregate objects
- each **aggregate object** has a complex structure and value, and a unique identifier

To summarize, aggregates have the following characteristics

- an aggregate is a complex-value object
- each aggregate is a unit of data access and atomic manipulation
- aggregates govern data distribution

In NoSQL database design, we should map each aggregate to a data modeling element having analogous features

- i.e., a unit of data access, atomic manipulation, and distribution
  - therefore, to a record/row, a document, or a group of related key-value pairs
- the role for columns, document fields, or individual key-value pairs has to be decided
- we would like to abstract from the features of specific datastores
The NoAM abstract data model for NoSQL databases exploits the commonalities of their various data models – but also introduces abstractions to balance their differences and variations.

- NoSQL datastores share the common provision of having two distinct notions of data access “units”
  - a larger unit (block) – which is the maximal unit of consistency/atomic data access and manipulation
    - a record/row – a document – a group of key-value pairs sharing part of the key
  - a smaller unit (entry) – which is a unit of data access as well
    - a column – a field – an individual key-value pair
- moreover, many datastores provide a notion of collection of larger data access units
  - a table – a document collection

The NoAM abstract data model

- The NoAM abstract data model
  - a database is a set of collections – each collection has a distinct name
  - a collection is a set of blocks – each block is identified in its collection by a block key
  - a block is a non-empty set of entries
  - each entry is a pair (ek,ev)
    - ek is the entry key – unique within its block
    - ev is a complex value, called the entry value
An application dataset can be represented in NoAM as follows
- the application dataset is represented by a NoAM database
- each class of aggregates is represented by a collection
  - the class name is used as collection name
- each aggregate object is represented by a block
  - the aggregate identifier is used as block key
- each aggregate object is represented by one or more entries in the corresponding block
  - the complex value of the aggregate object is partitioned into one or more entry values

Example: partitioning of aggregates

Player: mary:
  username: "mary",
  firstName: "Mary",
  lastName: "Wilson",
  games: [
    game: Game:2345, opponent: Player: rick,
    game: Game:2611, opponent: Player: ann
  ]

Player: rick:
  username: "rick",
  firstName: "Ricky",
  lastName: "Doe",
  score: 42,
  games: [
    game: Game:2345, opponent: Player: mary,
    game: Game:7425, opponent: Player: ann,
    game: Game:1241, opponent: Player: johnny
  ]

Game: 2345:
  id: "2345",
  firstPlayer: Player: mary,
  secondPlayer: Player: rick,
  rounds: [
    moves: ..., comments: ..., spell: ...,
    moves: ..., actions: ..., spell: ...
  ]
- Basic data representations

- In representing an aggregate object in NoAM, we use one or more entries – to partition the complex value of the aggregate
  - this partitioning can be based on
    - basic (predefined) data representation strategies
    - custom data representations
Entry per Aggregate Object (EAO)

- **Entry per Aggregate Object (EAO)**
  - an aggregate object is represented by a single entry
  - the entry value is the whole complex value – the entry key is empty

```plaintext
<table>
<thead>
<tr>
<th>username</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;mary&quot;</td>
<td>&quot;Mary&quot;</td>
<td>&quot;Wilson&quot;</td>
</tr>
</tbody>
</table>
```

Entry per Top-level Field (ETF)

- **Entry per Top-level Field (ETF)**
  - an aggregate object is represented by multiple entries – a distinct entry for each top-level field of the complex value
  - the entry value is the field value – the entry key is the field name

```plaintext
<table>
<thead>
<tr>
<th>username</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;mary&quot;</td>
<td>&quot;Mary&quot;</td>
<td>&quot;Wilson&quot;</td>
</tr>
</tbody>
</table>
```

```plaintext
<table>
<thead>
<tr>
<th>games</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
</tr>
<tr>
<td>game : Game:2345, opponent : Player:rick },</td>
</tr>
<tr>
<td>game : Game:2611, opponent : Player:ann</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
```
Entry per Atomic Value (EAV)

- **Entry per Atomic Value (EAV)**
  - an aggregate object is represented by multiple entries – a distinct entry for each atomic value in the complex value
  - the entry value is the atomic value – the entry key is the “access path” to the atomic value

<table>
<thead>
<tr>
<th>entry</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>“mary”</td>
</tr>
<tr>
<td>firstName</td>
<td>“Mary”</td>
</tr>
<tr>
<td>lastName</td>
<td>“Wilson”</td>
</tr>
<tr>
<td>games[0].game</td>
<td>Game:2345</td>
</tr>
<tr>
<td>games[0].opponent</td>
<td>Player:rick</td>
</tr>
<tr>
<td>games[1].game</td>
<td>Game:2611</td>
</tr>
<tr>
<td>games[1].opponent</td>
<td>Player:ann</td>
</tr>
</tbody>
</table>

- **Aggregate partitioning**

  - The basic data representation strategies can be suited in some cases – but we often need to partition aggregates in custom ways
  - indeed, aggregate partitioning can have impact on data access performance
Guidelines for aggregate partitioning – adapted from *Conceptual Database Design* (Batini, Ceri, Navathe, 1992)

- if an aggregate is small in size, or all or most of its data are accessed or modified together – then it should be represented by a single entry
- if an aggregate is large in size, and there are operations that frequently access or modify only specific portions of the aggregate – then it should be represented by multiple entries
- if two or more data elements are frequently accessed or modified together – then they should belong to the same entry
- if two or more data elements are usually accessed or modified separately – then they should belong to distinct entries

Operations for our online game

1. when a player connects to the application – the aggregate for the player should be retrieved
2. when a player selects a game to continue – the aggregate for the game should be retrieved
3. when a player completes a round for a game – the aggregate for the game should be updated, by adding the new round
4. when a player invites a friend for playing a new game – an aggregate for a new game should be created, and the aggregate for the opponent players should be updated, by adding the new game

For example, what does operation 3 suggest?
- each round should be represented using a distinct entry of the corresponding game aggregate
Aggregate partitioning: Example

<table>
<thead>
<tr>
<th>Player</th>
<th>username</th>
<th>&quot;mary&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>firstName</td>
<td>&quot;Mary&quot;</td>
</tr>
<tr>
<td></td>
<td>lastName</td>
<td>&quot;Wilson&quot;</td>
</tr>
<tr>
<td></td>
<td>games[0]</td>
<td>{ game : Game:2345, opponent : Player:rick }</td>
</tr>
<tr>
<td></td>
<td>games[1]</td>
<td>{ game : Game:2611, opponent : Player:ann }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Game</th>
<th>id</th>
<th>2345</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>firstPlayer</td>
<td>Player:mary</td>
</tr>
<tr>
<td></td>
<td>secondPlayer</td>
<td>Player:rick</td>
</tr>
<tr>
<td></td>
<td>rounds[0]</td>
<td>{ moves : ..., comments : ... }</td>
</tr>
<tr>
<td></td>
<td>rounds[1]</td>
<td>{ moves : ..., actions : ..., spell : ... }</td>
</tr>
</tbody>
</table>

- A language for data representations

- NoAM defines a language to specify aggregate partitioning – and therefore, data representations
  - the language can be used to describe or document a certain aggregate partitioning
  - more importantly, it can be used in a mapping system
    - the database designer uses the language to specify a data representation – in a system-independent way
    - the mapping framework interprets the specification – to represent aggregates in the specific target datastore and to handle operations over them

- The language has an XPath-like syntax – and we illustrate it by means of examples
Rule $/*$ specifies strategy Entry per Aggregate Object (EAO)
- the first * matches with aggregate classes
- the second * matches with aggregate identifiers
- the rule means “use an entry for each distinct aggregate class and distinct aggregate identifier”

```
```
| mary | { username : "mary", firstName : "Mary", lastName : "Wilson", games : { ( game : Game:2345, opponent : Player:rick ), ( game : Game:2611, opponent : Player:ann ) } } |
```

Rule $/*/*$ specifies strategy Entry per Top-level Field (ETF)
- the third * matches with top-level fields of aggregates
- the rule means “use an entry for each distinct aggregate class, aggregate identifier, and top-level field”

```
```
| mary | { username : "mary", firstName : "Mary", lastName : "Wilson", games : { ( game : Game:2345, opponent : Player:rick ), ( game : Game:2611, opponent : Player:ann ) } } |
```
A data representation is specified by a sequence of rules:

- `/Player/*/*` – “use ETF for players”
- `/Game/*` – “use EAO for games”

```
Player

<table>
<thead>
<tr>
<th>username</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>“mary”</td>
<td>“Mary”</td>
<td>“Wilson”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>games</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( game : Game:2345, opponent : Player:rick ),</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( game : Game:2611, opponent : Player:ann )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
Game

<table>
<thead>
<tr>
<th>id</th>
<th>firstPlayer</th>
<th>secondPlayer</th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
<td>mary</td>
<td>rick</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( moves : ... , comments : ... ),</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( moves : ... , actions : ... , spell : ... )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

It is possible to have more rules over a same aggregate class:

- `/Player/*/games[*]` – “use an entry for each game played by a player”
- `/Player/*/*` – “use ETF for the remaining data of each player”

```
mary

<table>
<thead>
<tr>
<th>username</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>“mary”</td>
<td>“Mary”</td>
<td>“Wilson”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>games[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( game : Game:2345, opponent : Player:rick )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>games[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( game : Game:2611, opponent : Player:ann )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
It is possible to have more rules over a same aggregate class

- \(/Player/*\)/games[*] – “use an entry for each game played by a player”
- \(/Player/*\) – “use EAO for the remaining data of each player”

```json
mary
{
  username: "mary",
  firstName: "Mary",
  lastName: "Wilson"
}

games[0]
{
  game: Game:2345,
  opponent: Player:rick
}

games[1]
{
  game: Game:2611,
  opponent: Player:ann
}
```

In the *implementation* phase, we map the intermediate data representation to the specific modeling elements of the target NoSQL datastore

- given that the NoAM data model generalizes the features of the various systems, while keeping their major aspects, it is rather straightforward to perform this activity

Please note that the implementation takes also care of mapping operations – specifically, CRUD operations (create, read, update, delete) over aggregate objects to specific data access operations

- we do not discuss this issue here
- please find more details in the references
Oracle NoSQL: Implementation

Oracle NoSQL is a key-value store – a database is a collection of key-value pairs

- values are binary strings, opaque to the datastore
- a key is composed of two parts
  - the major key is a non-empty sequence of strings
  - the minor key is a (possibly-empty) sequence of strings
  - e.g., /Player/mary/-/username
- the major key controls data distributions – key-value pairs having the same major key are allocated in a same node
- atomic operations on individual key-value pairs – but also on groups of key-value pairs having the same major key

Mapping from NoAM to Oracle NoSQL

- a key-value pair for each entry
  - the major key is composed of
    - the collection name
    - the block key (i.e., the aggregate identifier)
  - the minor key represents the entry key (i.e., an access path)
  - the value represents the entry value
    - it can be either a simple value, or
    - the serialization of a complex value – e.g., in JSON
### Oracle NoSQL: Implementation

**Player**

<table>
<thead>
<tr>
<th>username</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>mary</td>
<td>Mary</td>
<td>Wilson</td>
</tr>
</tbody>
</table>

Games:

- **Player:mary**
  - game: Game:2345, opponent: Player:rick
  - game: Game:2611, opponent: Player:ann

**Key-Value pairs**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Player/mary/username</td>
<td>mary</td>
</tr>
<tr>
<td>/Player/mary/firstName</td>
<td>Mary</td>
</tr>
<tr>
<td>/Player/mary/lastName</td>
<td>Wilson</td>
</tr>
<tr>
<td>/Player/mary/games[0]</td>
<td>(game: Game:2345, opponent: Player:rick)</td>
</tr>
<tr>
<td>/Player/mary/games[1]</td>
<td>(game: Game:2611, opponent: Player:ann)</td>
</tr>
</tbody>
</table>

---

**Player**

<table>
<thead>
<tr>
<th>username</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>mary</td>
<td>Mary</td>
<td>Wilson</td>
</tr>
</tbody>
</table>

Games:

- **Player:mary**
  - game: Game:2345, opponent: Player:rick
  - game: Game:2611, opponent: Player:ann

**Key-Value pairs**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Player/mary/username</td>
<td>mary</td>
</tr>
<tr>
<td>/Player/mary/firstName</td>
<td>Mary</td>
</tr>
<tr>
<td>/Player/mary/lastName</td>
<td>Wilson</td>
</tr>
<tr>
<td>/Player/mary/games[0]</td>
<td>{game: Game:2345, opponent: Player:rick}</td>
</tr>
<tr>
<td>/Player/mary/games[1]</td>
<td>{game: Game:2611, opponent: Player:ann}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
MongoDB is a document store – a database is a set of documents
- each document has a complex value and an identifier, and documents are organized in collections

- Mapping from NoAM to MongoDB
  - a document collection for each NoAM collection (aggregate class)
  - a main document for each block (aggregate)
  - a top-level field for each entry
  - the special _id field for the block key (aggregate identifier)
  - atomic operations on individual documents – or on their fields

**MongoDB: Implementation**

| Player | username | “mary” |
|        | firstName | “Mary” |
|        | lastName  | “Wilson” |
|        | games[0]  | { game : “Game:2345”, opponent : “Player:rick” } |

collection **Player**

<table>
<thead>
<tr>
<th>id</th>
<th>document</th>
</tr>
</thead>
</table>
MongoDB: Alternative implementation

- A different implementation
  - reconstruct structure of complex values

<table>
<thead>
<tr>
<th>Player</th>
<th>username</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>mary</td>
<td>“mary”</td>
<td>“Mary”</td>
<td>“Wilson”</td>
</tr>
</tbody>
</table>

```

collection **Player**

<table>
<thead>
<tr>
<th>id</th>
<th>document</th>
</tr>
</thead>
</table>
| mary | {
   “_id”: “mary”,
   “username”: “mary”,
   “firstName”: “Mary”,
   “lastName”: “Wilson”,
   “games”: [{ “game”: “Game:2345”, “opponent”: “Player:rick” },
               { “game”: “Game:2611”, “opponent”: “Player:ann” }]
   } |

DynamoDB: Implementation

- Amazon DynamoDB is an extensible record store
  - a database is a set of tables
  - each table is a set of items
  - each item contains a set of attributes, each with a name and a value
  - each table has a primary key – composed of a hash partition attribute and an optional range attribute
  - the partition attribute controls distribution of items
  - atomic operations on individual items – or on their columns

- Mapping from NoAM to DynamoDB
  - a table for each collection (aggregate class)
  - an item for each block (aggregate) – whose primary key is the block key (aggregate identifier)
  - an attribute for each entry
- Conclusion (NoAM)

- NoAM (NoSQL Abstract Model) is a logical approach to NoSQL database design
  - a logical approach
    - initial design activities are independent of any specific target systems
  - it is based on NoAM
    - NoAM is an intermediate, abstract data model for NoSQL databases – which exploits the commonalities of their various data models – but also introduces abstractions to balance their differences and variations
Open issues

- NoAM approach
  - the application of the approach might result in a number of candidate data representations – rather than to a single one
  - tools can help the designer to assess a preferred solution
- NoAM data model
  - further abstractions are needed to represent aspects related to concurrency control and consistency – e.g., optimistic, pessimistic, and snapshot transactions, versions and locks
  - derived data and materialized views
  - so far, we assumed that data is represented in a non-redundant way – some redundancy can improve performance – but view maintenance could affect consistency negatively
- support to multi-aggregate transactions

*ONDM (Object-NoSQL Datastore Mapper)*

- **ONDM** (Object-NoSQL Datastore Mapper) is a framework that provides application developers with
  - a uniform access towards a variety of NoSQL datastores
  - the ability to map application data to different data representations, in a flexible way

- Main features of ONDM
  - object-oriented API, based on Java Persistence API (JPA)
  - transparent access to various NoSQL datastores – such as Oracle NoSQL, Redis, MongoDB, CouchBase, and Cassandra
  - internal representation based on NoAM
  - flexible data representations – based on the NoAM language for data representations
- Architecture of ONDM

- A layered architecture
  - **API** – based on JPA, offers CRUD operations to manipulate aggregates
  - **internal aggregate manager** – conversion between aggregate objects and an internal representation (JSON) – cache mgmt
  - **data representation manager** – in NoAM, wrt the specified data representation
  - **datastore adapters** – conversion between NoAM and specific data structures and operations

---

**Application programming interface**

- The API is based on JPA – with some **specific extensions**

```java
@Entity
@NoSQL(datastore="OracleNoSQL")
@DataRepresentation(strategy="EAO")
class Player {
    @Id String userName;
    String firstName;
    String lastName;
    List<GameInfo> games;
}

@Embeddable
class GameInfo {
    String opponent;
    @ManyToOne Game game;
}
```
The API is based on JPA – with some specific extensions

```java
@Entity
@NoSQL(datastore="OracleNoSQL")
@DataRepresentation(rules="*/rounds[*];*")
class Game {
    @Id long id;
    int roundCount;
    @ManyToOne Player firstPlayer;
    @ManyToOne Player secondPlayer;
    List<Round> rounds;
}

@Embeddable
class Round {
    List<Move> moves;
    List<String> comments;
}
```

The API offers CRUD operations on entities and aggregates

- with automatic change tracking

```java
public void addNewRound(long gameId, Round round) {
    /* em is the entity manager */
    em.getTransaction().begin();
    Game game = em.find(Game.class, gameId);
    game.getRounds().add(round);
    game.incrementRoundCount();
    em.getTransaction().commit();
}
```
Scenario – *creation of a new aggregate*

- requested through the API
- the internal aggregate manager converts in-memory application objects to the internal representation (JSON) – and put it in the cache
- the data representation manager converts the JSON representation in NoAM – according to the specified data representation – e.g.,
  \@DataRepresentation(rules="*/rounds[*]:*")
- the selected datastore adapter – e.g.,
  \@NoSQL(datastore="OracleNoSQL") – stores the NoAM data according to specific data structures and operations

Scenario – *reading of an aggregate*

- requested through the API
- the internal aggregate manager checks if the aggregate is already in the cache
- otherwise, the request is passed to the proper datastore adapter
- the datastore adapter performs the reading – and converts data to NoAM
- the data representation manager converts data to the internal representation in JSON – according to the specified data representation
- the internal representation manager puts the aggregate in the cache – and converts JSON to in-memory application objects
Scenario – update of an aggregate – with automatic change tracking

- commit requested through the API
- the internal aggregate manager compares the updated aggregate object with the corresponding snapshot in the cache – to find possible changes that would require effective updates in the underlying datastore
- the difference is computed by the data representation manager – according to the specified data representation – to track down differences at the level of individual entries
- updates implied by actual differences are performed by a datastore adapter

ONDM (Object-NoSQL Datastore Mapper) is a framework for the uniform access towards a variety of NoSQL datastores

- object-oriented, JPA-based API
- transparent access to various NoSQL datastores
  - currently, Oracle NoSQL, Redis, MongoDB, CouchBase, and Cassandra
  - internal representation based on NoAM
- flexible data representations – based on the NoAM language for data representations
Conclusion (ONDM)

- Status of ONDM (as of December 2013)
  - current prototype has a simpler architecture – and implements less features – than the ones described here

- Future work
  - update prototype to new architecture
  - investigate issues related to derived data, materialized views, and asynchronous view maintenance
  - investigate issues related to concurrency control and consistency

A case study in NoSQL db design

- The database design activity can result in a number of candidate data representations – rather than to a single one
  - consider again the operations for our online game
    2. when a player selects a game to continue – the aggregate for the game should be retrieved
    3. when a player completes a round for a game – the aggregate for the game should be updated, by adding the new round
  - operations 2 and 3 suggest different choices for the representation of rounds – (i) all together in a single entry or (ii) using a distinct entry for each round

- In this case, experiments are needed to assess the most suitable design solution – and ONDM can help in performing them
  - an important feature is the ability to select a desired data representation in a declarative way – using the NoAM language for data representations
To decide between the various candidate representations, a few experiments can help:

- the target datastore is Oracle NoSQL (single node)
- three candidate representations
  - an entry for a whole game – EAO
  - `/Game/*/rounds[*]` + `/Game/*/*` – Rounds+ETF
  - `/Game/*/rounds[*]` + `/Game/*` – Rounds+EAO
- various workloads
  - game retrieval
  - round addition
    - mixed – 80% game retrievals + 20% round additions
- each game is 8kb, each round is 0.5kb
- database size is in GB, timings are ms per operation
- Note/commenti

- Posizionare il disclaimer (o eliminarlo?)
  - ad esempio, alla fine della panoramica sui sistemi NoSQL
- Forse il modello astratto di NoAM subito dopo la presentazione dei diversi sistemi NoSQL
- Forse parlare di più delle operazioni – che comunque sono parte del modello
- Forse l’overview dopo la parte di introduzione ai sistemi NoSQL

- Troppo lungo!!!
  - 20 minuti per la parte iniziale di overview e sui sistemi nosql
  - poi altri 30-35 minuti per noam
  - 5 minuti per gli esperimenti con ondm per dire che certe scelte hanno impatto sulle prestazioni

---

**Disclaimer**

- We do not consider here
  - other application areas for NoSQL datastores – such as the complex processing of large datasets, e.g., using MapReduce
  - other NoSQL system categories – such as graph datastores
  - some important NoSQL features – such as replication – as well as some important quality requirements – such as availability