Luca Cabibbo

NoSQL Database Design for Next-Generation Web Applications

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

- Introduction

- *NoSQL datastores* are a new generation of distributed database systems – they have been designed to manage large data sets distributed over many servers
  - a promise of NoSQL database technology is to support the development of *next-generation web applications*
  - in this context, we are interested in *NoSQL database design*
  - we also present an *abstract data model* for NoSQL databases

- Outline

- NoSQL database systems
- NoAM – an abstract data model for NoSQL databases
- NoSQL database design for next-generation web applications
- The NoAM approach to NoSQL database design
  - overview
  - aggregates and aggregate design
  - aggregate partitioning
  - a language for data representations
  - implementation
  - conclusion
- ONDM (Object-NoSQL Datastore Mapper)
  - architecture
  - conclusion
- A case study 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
  - a main requirement for these systems is the ability to manage large data sets distributed over many servers – whereas relational DBMSs are not designed to be run on clusters

NoSQL and heterogeneity

- The NoSQL landscape is characterized by a high heterogeneity
  - http://nosql-database.org/ lists 150 non-relational databases
  - 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

Key-value stores

- 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

redis

Tokyo Cabinet 8192PIB

Scalaris

riak

ORACLE NOSQL DATABASE

Key-value stores: examples

key	value
Player:mary	username : mary firstName : Mary lastName : Wilson games : { … }
Player:rick	username : rick firstName : Ricky lastName : Doe score : 42 games : { ... }

a hash value

redis

key	value
/Player/mary/-	{ “username” : “mary”, “firstName” : “Mary”, “lastName” : “Wilson”, “games” : […] }
/Player/rick/-	{ “username” : “rick”, “firstName” : “Ricky”, lastName : “Doe”, “score” : “42”, “games” : […] }

key	value
/Player/mary/-/username	mary
/Player/mary/-/firstName	Mary
/Player/mary/-/lastName	Wilson
/Player/mary/-/games	[…]

ORACLE NOSQL DATABASE

Document stores

- 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

terrastore Couchbase CouchDB mongoDB

Document stores: example

collection **Player**

id	document

mary	{
	“_id” : “mary”,
	“username” : “mary”,
	“firstName” : “Mary”,
	“lastName” : “Wilson”,
	“games” : [{ “game” : “Game:2345”, “opponent” : “Player:rick” },
	{ “game” : “Game:2611”, “opponent” : “Player:ann” }]
	}

rick	{
	“_id” : “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” }]
	}

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

Extensible record stores: example

table **Player**

<u>username</u>	firstName	lastName	score	games[0]	games[1]	games[2]	…
mary	Mary	Wilson		{…}	{…}		
rick	Ricky	Doe	42	{…}	{…}	{…}	

* NoAM – an abstract data model for NoSQL databases

- The NoSQL landscape is characterized by a high heterogeneity
  - 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"
- To this end, we propose **NoAM** (**NoSQL Abstract Model**) – an abstract and system independent data model for NoSQL databases
  - NoAM aims at exploiting the commonalities of their various data models – but it also introduces abstractions to balance their differences and variations

The NoAM abstract data model

- Main commonalities of their various NoSQL data models
  - NoSQL datastores share the common provision of having a modeling data element that is a *distribution, access and manipulation unit* (*DAM unit*)
    - a data access unit
      - more precisely, a maximal unit of consistency/atomic data access and manipulation
    - a unit of distribution
      - each DAM unit is located on a single node of the cluster – but in general different DAM units are distributed among the nodes of the cluster
  - in the various systems, a DAM unit can be
    - a record/row – a document – a group of key-value pairs sharing part of the key
  - in NoAM, a DAM unit is called a **block**

The NoAM abstract data model

- Main commonalities of their various NoSQL data models
  - NoSQL datastores also offer the ability to access just some parts of a DAM unit – they have a modeling data element that is a *"smaller" data access unit* (*SDA unit*)
  - in the various systems, an SDA unit can be
    - a column – a field – an individual key-value pair
  - in NoAM, a SDA unit is called an **entry**
  - moreover, many datastores provide a notion of *collection* of data access units
  - in the various systems, a collection can be
    - a table – a document collection
  - in NoAM, a collection of DAM units (blocks) is called a **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 value (either a scalar or a complex value), called the *entry value*

Example: a NoAM database

Player

mary	username	"mary"
	firstName	"Mary"
	lastName	"Wilson"
	games[0]	⟨ game : Game:2345, opponent : Player:rick ⟩
	games[1]	⟨ game : Game:2611, opponent : Player:ann ⟩

rick	username	"rick"
	firstName	"Ricky"
	lastName	"Doe"
	score	42
	games[0]	⟨ game : Game:2345, opponent : Player:mary ⟩
	games[1]	⟨ game : Game:7425, opponent : Player:ann ⟩
	games[2]	⟨ game : Game:1241, opponent : Player:johnny ⟩

Game

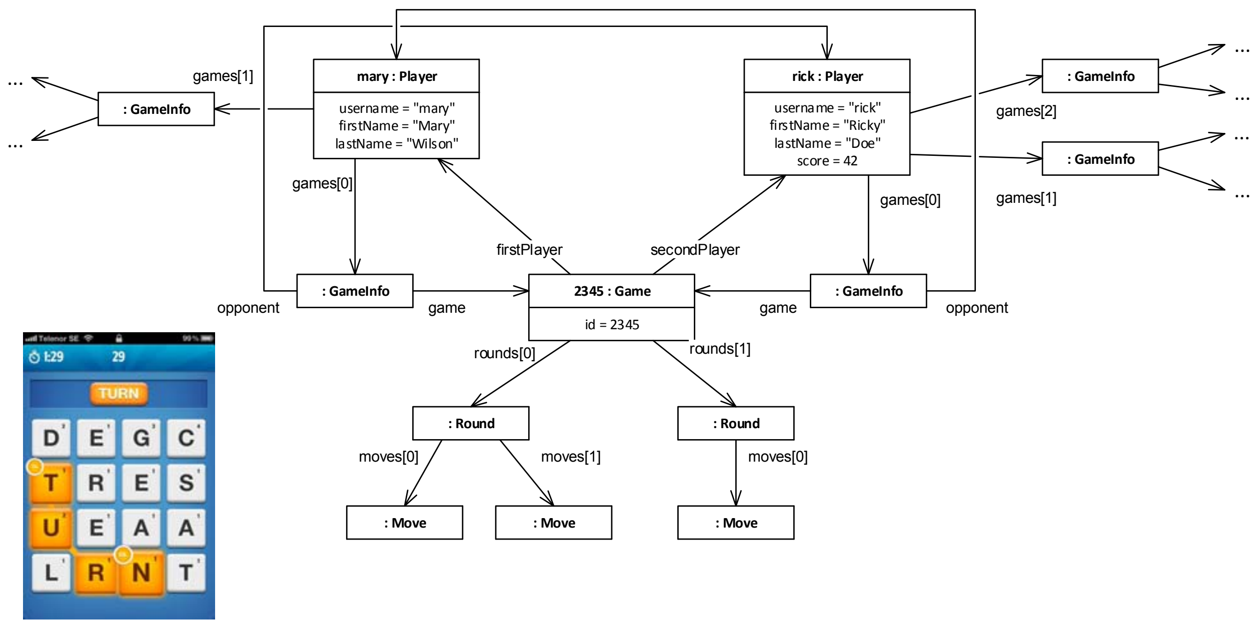
2345	id	2345
	firstPlayer	Player:mary
	secondPlayer	Player:rick
	rounds[0]	⟨ moves : … , comments : … ⟩
	rounds[1]	⟨ moves : … , actions : … , spell : … ⟩

* NoSQL database design for next-generation web applications

- 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

A running example

- Consider a fictious online, web 2.0 game – e.g., some variant of Ruzzle – which should manage various application objects, including players, games, rounds, and moves



A running example

- Consider a fictious online, web 2.0 game – e.g., some variant of Ruzzle – 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?

NoSQL database design

- In NoSQL database design
  - decisions on the organization of data are required, in any case
  - these decisions 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

- 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

* The NoAM approach to NoSQL database design

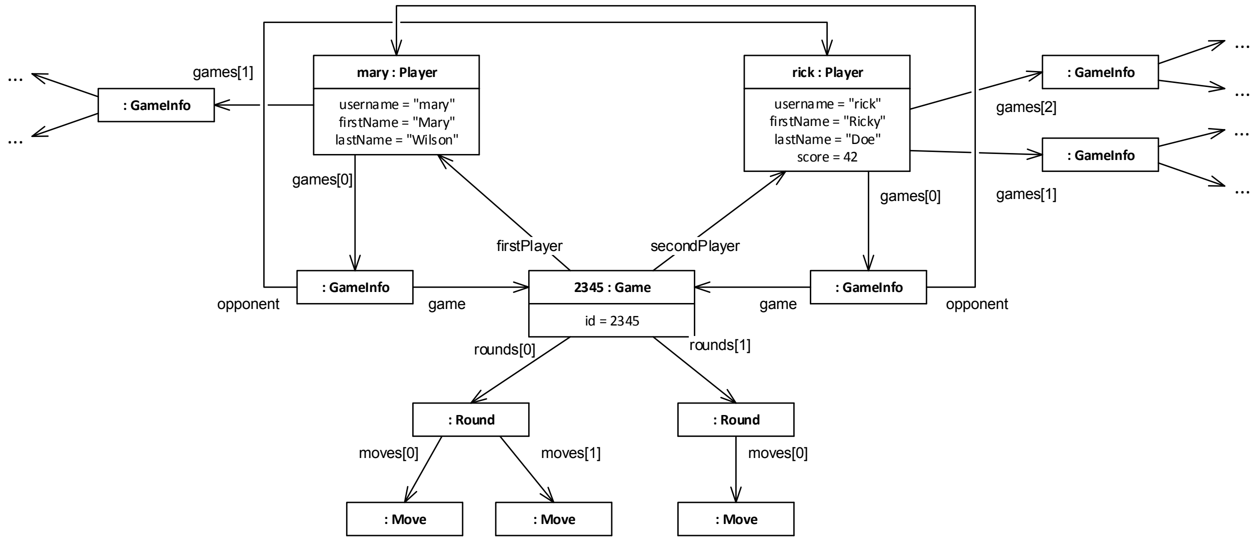
- We propose the **NoAM approach to NoSQL database design**
  - tailored to the requirements of next-generation web applications
  - based on the NoAM abstract data model for NoSQL databases
  - a high level/system independent approach – the initial design activities are independent of any specific target 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

- Overview

- 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

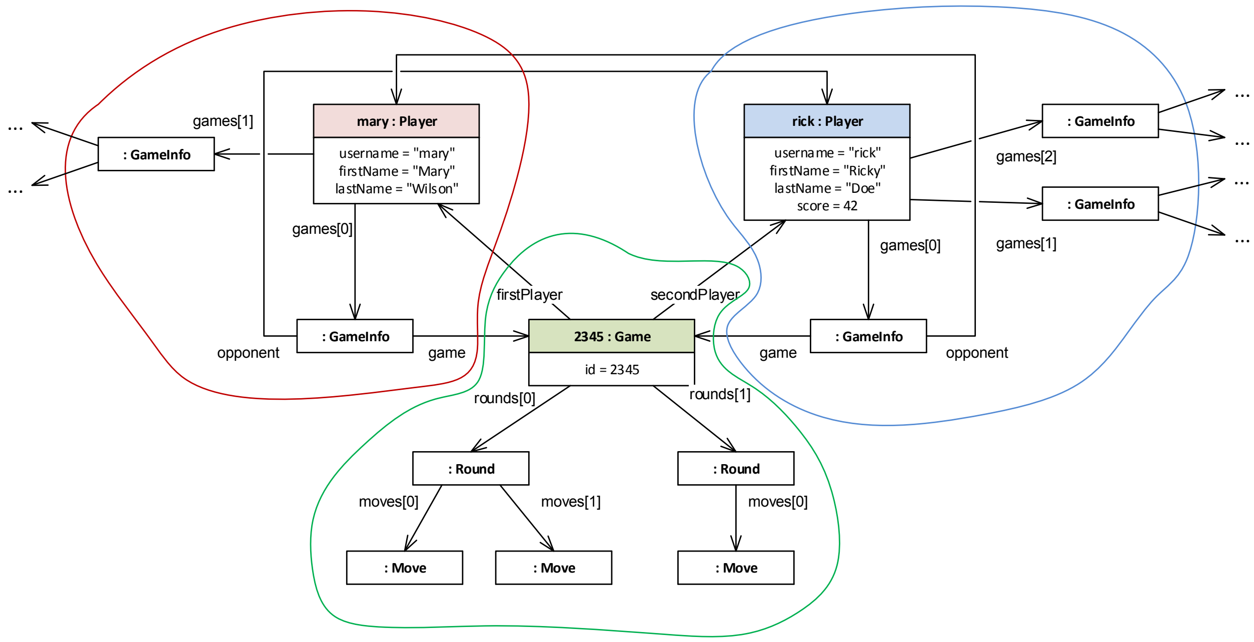
Application data

- We start by considering application data objects…



Aggregates

- … we group them in aggregates (*decisions needed!*) …



Aggregates as complex-value objects

- … we consider aggregates as complex-value objects…

```
Player:mary : ⟨
    username : "mary",
    firstName :
    lastName :
    games : {
        ⟨ gam
        ⟨ gam
    }
⟩
```

```
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 : … ⟩
    }
⟩
```


Aggregate partitioning

- … we partition these complex values (*decisions needed!*) …

```
Player:mary : ⟨
    username : "mary",
    firstName :
    lastName :
    games : {
        ⟨ gam
        ⟨ gam
    }
⟩
```

```
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 : … ⟩
    }
⟩
```

Data representation in NoAM

- … and represent them into an abstract data model for NoSQL databases (*consequence of decisions*) …

Player

	username	"mary"
	firstName	"Mary"

rick	username	"rick"
	firstName	"Ricky"
	lastName	"Doe"
	score	42
	games[0]	〈 game : Game:2345, opponent : Player:mary 〉
	games[1]	〈 game : Game:7425, opponent : Player:ann 〉
	games[2]	〈 game : Game:1241, opponent : Player:johnny 〉

Game

2345	id	2345
	firstPlayer	Player:mary
	secondPlayer	Player:rick
	rounds[0]	〈 moves : … , comments : … 〉
	rounds[1]	〈 moves : … , actions : … , spell : … 〉

Implementation

- … and finally we map the intermediate representation to the data structures of the target datastore (*the approach specifies how*)

table **Player**

username	firstName	lastName	score	games[0]	games[1]	games[2]	…
mary	Mary	Wilson		{…}	{…}		
rick	Ricky	Doe	42	{…}	{…}	{…}	

table **Game**

id	firstPlayer	secondPlayer	rounds[0]	rounds[1]	rounds[2]	…
2345	Player:mary	Player:rick	{…}	{…}		

Implementation

- … and finally we map the intermediate representation to the data structures of the target datastore (*the approach specifies how*)

key	value
/Player/mary/-/username	mary
/Player/mary/-/firstName	Mary
/Player/mary/-/lastName	Wilson
/Player/mary/-/games[0]	{ “game” : “Game:2345”, “opponent” : “Player:rick” }
/Player/mary/-/games[1]	{ “game” : “Game:2611”, “opponent” : “Player:ann” }
…	…
/Games/2345/-/id	2345
/Games/2345/-/firstPlayer	Player:mary
/Games/2345/-/secondPlayer	Player:rick
/Games/2345/-/rounds[0]	{ … }
/Games/2345/-/rounds[1]	{ … }
…	…

ORACLE NOSQL DATABASE

- Aggregates and aggregate design

- 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
  - aggregate design affects scalability and the scope of atomic operations – and therefore, the ability to support relevant integrity constraints

Design of scalable applications

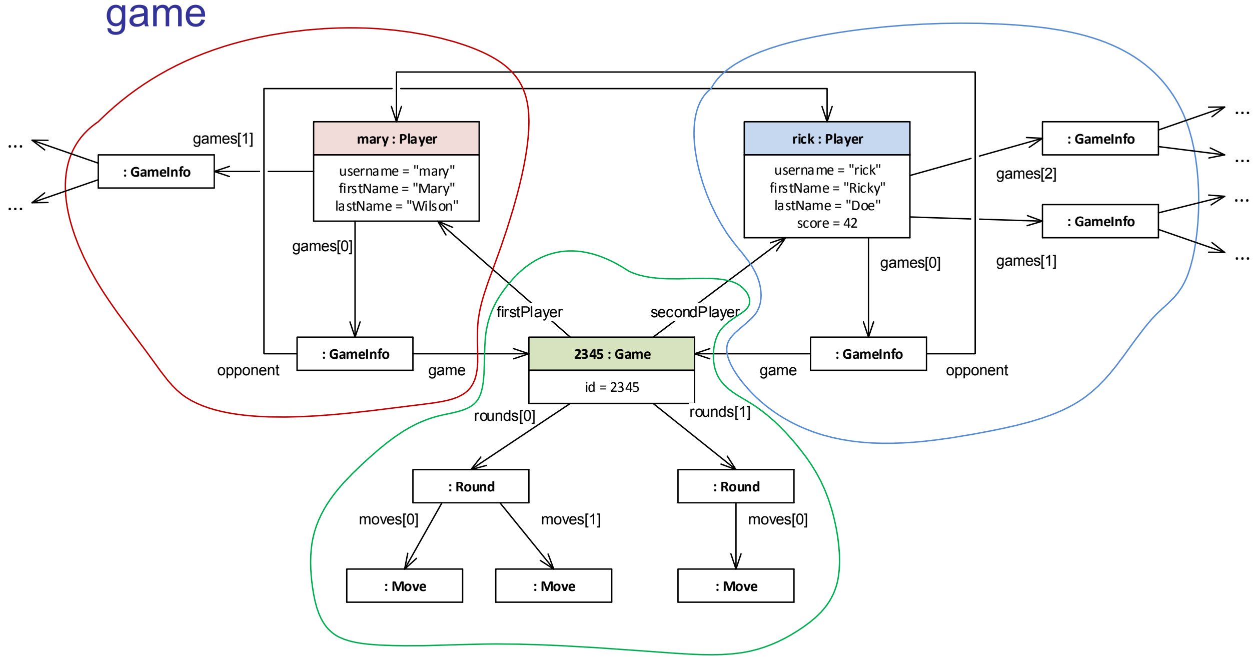
- The design of scalable applications is discussed in a seminal paper by Pat Helland – *Life beyond distributed transactions: an apostate's opinion*, CIDR 2007
  - data should be organized as a set of complex-value objects with unique identifiers – called *entities* or *aggregates* – each aggregate is 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

Aggregates in Domain-driven design

- *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
    - 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



Application data model

- At the application level, data is organized in aggregates
  - each **aggregate object is** can be considered a complex-value object, with a unique identifier
  - a set of aggregate objects is a **class**
  - a set of class form an **application dataset**

Player:mary : 〈
user
first
last
gam
}
〉

Player:rick : 〈
username : "rick",
firstName : "Ricky",
lastName : "Doe",
scor
gam
}
〉

Game:2345 : 〈
id : "2345",
firstPlayer : Player:mary,
secondPlayer : Player:rick,
rounds : {
〈 moves : … , comments : … 〉,
〈 moves : … , actions : … , spell : … 〉
}
〉

Aggregates in NoSQL db design

- 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

Aggregates in NoSQL db design

- In NoSQL database design, we should map each aggregate to a data modeling element having analogous features
  - each *aggregate* should be mapped to a unit of data access, atomic manipulation, and distribution
    - therefore, a record/row, a document, or a group of related key-value pairs – that is, a NoAM *block*
  - classes of aggregates can then be mapped to NoAM *collections*
  - the role for columns, document fields, or individual key-value pairs (i.e., NoAM *entries*) has to be discussed
  - we would like to abstract from the features of specific datastores – NoAM enables us to do so

Representing aggregates in NoAM

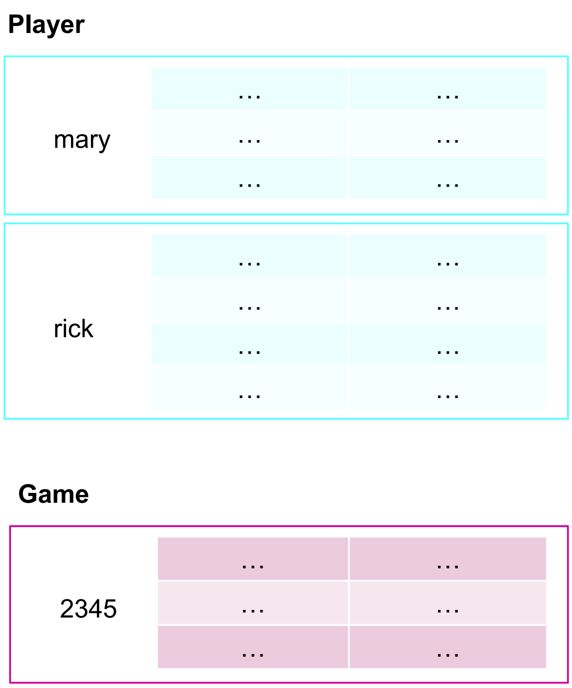
- An application dataset can be represented in NoAM as follows
  - the *application dataset* is represented by a NoAM *database*
  - each *class of aggregates* is represented by a *collection*
    - the class name is used as collection name
  - each *aggregate object* is represented by a *block*
    - 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

Representing aggregates in NoAM

```
Player:mary : ⟨
        username : "mary",
        firstName : "Mary",
⟩
```

```
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 : … ⟩
        }
⟩
```



Example: partitioning of aggregates

Player:mary : ⟨
username : "mary",
firstName : "Mary",
lastName :
games : {
⟨ gam
⟨ gam
}
⟩

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 : … ⟩
}
⟩

Example: aggregates in NoAM

Player

mary	username	"mary"
	firstName	"Mary"
	lastName	"Wilson"
	games[0]	⟨ game : Game:2345, opponent : Player:rick ⟩
	games[1]	⟨ game : Game:2611, opponent : Player:ann ⟩
rick	username	"rick"
	firstName	"Ricky"
	lastName	"Doe"
	score	42
	games[0]	⟨ game : Game:2345, opponent : Player:mary ⟩
	games[1]	⟨ game : Game:7425, opponent : Player:ann ⟩
	games[2]	⟨ game : Game:1241, opponent : Player:johnny ⟩

Game

2345	id	2345
	firstPlayer	Player:mary
	secondPlayer	Player:rick
	rounds[0]	⟨ moves : … , comments : … ⟩
	rounds[1]	⟨ moves : … , actions : … , spell : … ⟩

- Aggregate partitioning

- In representing an aggregate object in NoAM, we use one or more entries – to partition the complex value of the aggregate
  - aggregate partitioning affects performance of data access and manipulation operations
  - 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

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

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

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

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

mary	username	"mary"
	firstName	"Mary"
	lastName	"Wilson"
	games[0].game	Game:2345
	games[0].opponent	Player:rick
	games[1].game	Game:2611
	games[1].opponent	Player:ann

Custom aggregate partitioning

- The basic data representation strategies can be suited in some cases – but we often need to partition aggregates in custom ways
  - aggregate partitioning can be driven by data access operations – since it affects the performance of database operations
  - each element of a partition (i.e., an entry) can represent either a scalar value or a complex value – the usage of "entries" with a complex value is a common practice in NoSQL datastores – e.g., Protocol Buffers, Avro schemas

Guidelines for aggregate partitioning

- *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*

Aggregate partitioning: Example

- 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

Player

mary	username	“mary”
	firstName	“Mary”
	lastName	“Wilson”
	games[0]	〈 game : Game:2345, opponent : Player:rick 〉
	games[1]	〈 game : Game:2611, opponent : Player:ann 〉

Game

2345	id	2345
	firstPlayer	Player:mary
	secondPlayer	Player:rick
	rounds[0]	〈 moves : … , comments : … 〉
	rounds[1]	〈 moves : … , actions : … , spell : … 〉

- 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

The language - by 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	ε	〈
		username : "mary",
		firstName : "Mary",
		lastName : "Wilson",
		games : {
		〈 game : Game:2345, opponent : Player:rick 〉,
		〈 game : Game:2611, opponent : Player:ann 〉
		}
		〉

The language - by examples

- 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 〉 }

The language - by examples

- A data representation is specified by a sequence of rules
  - **/Player/*/*** – “use ETF for players”
  - **/Game/*** – “use EAO for games”

Player

mary	username	“mary”
	firstName	“Mary”
	lastName	“Wilson”
	games	{ 〈 game : Game:2345, opponent : Player:rick 〉, 〈 game : Game:2611, opponent : Player:ann 〉 }

Game

2345	ε	〈 id : 2345, firstPlayer : Player:mary, secondPlayer : Player:rick, rounds : { 〈 moves : … , comments : … 〉, 〈 moves : … , actions : … , spell : … 〉 } 〉

The language – by examples

- 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	username	"mary"
	firstName	"Mary"
	lastName	"Wilson"
	games[0]	〈 game : Game:2345, opponent : Player:rick 〉
	games[1]	〈 game : Game:2611, opponent : Player:ann 〉

The language – by examples

- 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"

mary	ε	〈 username : "mary", firstName : "Mary", lastName : "Wilson" 〉
	games[0]	〈 game : Game:2345, opponent : Player:rick 〉
	games[1]	〈 game : Game:2611, opponent : Player:ann 〉

- Implementation

- In the *implementation* phase, we map the intermediate *data representation* to the specific *data 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

ORACLE
NOSQL DATABASE

Oracle NoSQL: Implementation

- 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 DATABASE

Oracle NoSQL: Implementation

Player

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

key	value
/Player/mary/-	{ “username” : “mary”, “firstName” : “Mary”, “lastName” : “Wilson”, “games” : […] }
/Player/rick/-	{ “username” : “rick”, “firstName” : “Ricky”, lastName : “Doe”, “score” : “42”, “games” : […] }
/Game/2345/-	{ “id” : “2345”, “firstPlayer” : “Player:mary”, “secondPlayer” : “Player:rick”, “rounds” : […] }

ORACLE NOSQL DATABASE

Oracle NoSQL: Implementation

Player

mary	username	“mary”
	firstName	“Mary”
	lastName	“Wilson”
	games[0]	〈 game : Game:2345, opponent : Player:rick 〉
	games[1]	〈 game : Game:2611, opponent : Player:ann 〉

key	value
/Player/mary/-/username	mary
/Player/mary/-/firstName	Mary
/Player/mary/-/lastName	Wilson
/Player/mary/-/games[0]	{ “game” : “Game:2345”, “opponent” : “Player:rick” }
/Player/mary/-/games[1]	{ “game” : “Game:2611”, “opponent” : “Player:ann” }
…	…

ORACLE NOSQL DATABASE

MongoDB: Implementation

- 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

MongoDB: Implementation

Player

mary	username	"mary"
	firstName	"Mary"
	lastName	"Wilson"
	games[0]	⟨ game : Game:2345, opponent : Player:rick ⟩
	games[1]	⟨ game : Game:2611, opponent : Player:ann ⟩

collection **Player**

id	document
mary	{ "_id" : "mary", "username" : "mary", "firstName" : "Mary", "lastName" : "Wilson", "games[0]" : { "game" : "Game:2345", "opponent" : "Player:rick" }, "games[1]" : { "game" : "Game:2611", "opponent" : "Player:ann" } }

MongoDB: Alternative implementation

- A different implementation
  - reconstruct structure of complex values

Player

mary	username	"mary"
	firstName	"Mary"
	lastName	"Wilson"
	games[0]	⟨ game : Game:2345, opponent : Player:rick ⟩
	games[1]	⟨ game : Game:2611, opponent : Player:ann ⟩

collection **Player**

id	document
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

DynamoDB: Implementation

Player

mary	username	"mary"
	firstName	"Mary"
	lastName	"Wilson"
	games[0]	〈 game : Game:2345, opponent : Player:rick 〉
	games[1]	〈 game : Game:2611, opponent : Player:ann 〉

table **Player**

username	firstName	lastName	score	games[0]	games[1]	games[2]	…
mary	Mary	Wilson		{…}	{…}		
rick	Ricky	Doe	42	{…}	{…}	{…}	

table **Game**

id	firstPlayer	secondPlayer	rounds[0]	rounds[1]	rounds[2]	…
2345	Player:mary	Player:rick	{…}	{…}		

- Conclusion (NoAM)

- NoAM (NoSQL Abstract Model) is a high-level approach to NoSQL database design for next-generation web applications
  - a high-level 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

Conclusion (NoAM data model)

- Open issues
  - NoAM data model
    - other abstractions are needed to represent further data modeling elements available in NoSQL datastores
    - further abstractions related to relevant metadata – e.g., versions and timestamps, to support concurrency control and consistency management
  - derived data and materialized views
    - so far, we have assumed that data is represented in a non-redundant way – some redundancy is usually suggested in NoSQL databases, to improve performance – but note that view maintenance could affect consistency negatively
    - support to multi-aggregate transactions is required

Conclusion (NoAM approach)

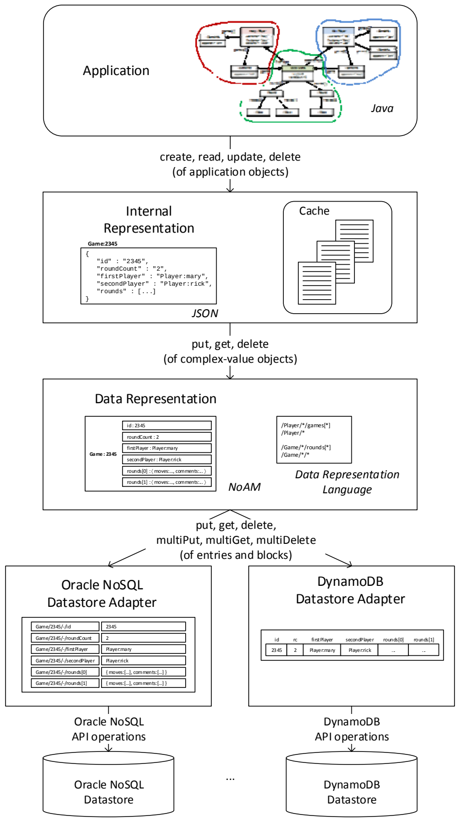
- Open issues
  - NoAM approach
    - the proposed guidelines can propose conflicting suggestions – therefore, the application of the approach might result in a number of candidate data representations, rather than to a single one
    - tools can help the designer to assess a preferred solution
  - NoSQL database design for different settings
    - for example, to support query-intensive applications and analytical queries

* 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



* 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

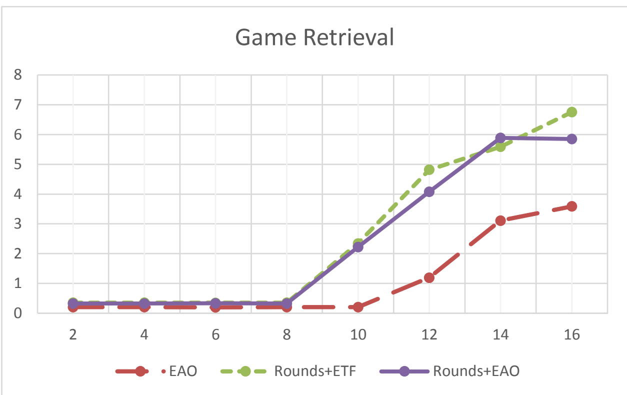
  2. when a player selects a game to continue – the aggregate for the game should be retrieved
  3. when a player completes a round for a game – the aggregate for the game should be updated, by adding the new round

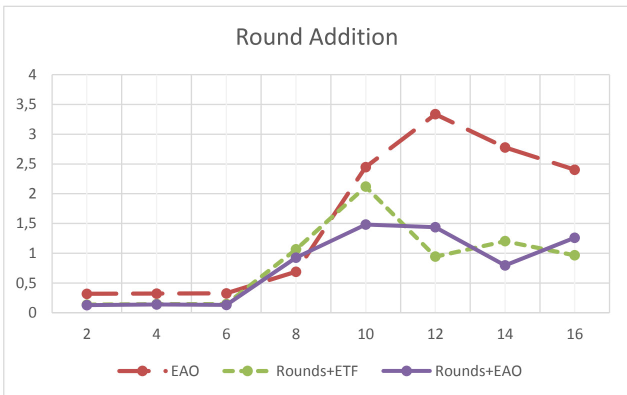
  - operations 2 and 3 suggest different choices for the representation of rounds – (i) all together in a single entry or (ii) using a distinct entry for each round
- In this case, experiments are needed to assess the most suitable design solution – and ONDM can help in performing them
  - an important feature is the ability to select a desired data representation in a declarative way – using the NoAM language for data representations

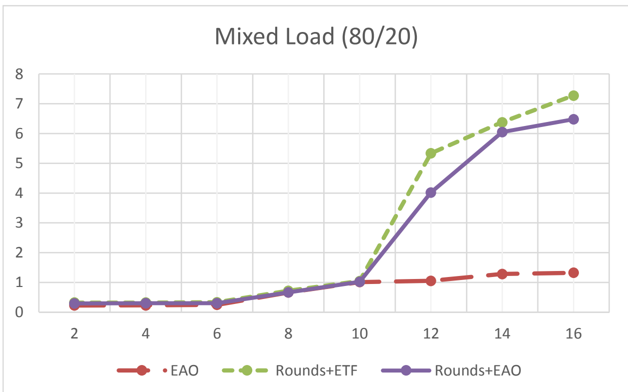
A case study in NoSQL db design

- 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

A case study in NoSQL db design







- Conclusion (case study)

- The experiments show that aggregate partitioning has indeed impact on the performance of the various operations
  - in general, when using a NoSQL database, decisions on the organization of data are required
  - these decisions are significant – as the data representation affects major quality requirements – such as scalability, performance, and consistency