Models for NoSQL databases: a contradiction?

Paolo Atzeni
Dipartimento di Ingegneria
Sezione di Informatica e Automazione

ROMA TRE
UNIVERSITÀ DEGLI STUDI

Based on work done with (or by!)
F. Bugiotti, L. Rossi, L. Cabibbo, R. Torlone and others

Stockholm, 21 October 2015

Content

- A personal (and questionable …) perspective on the role models have for NoSQL systems:
  - NoSQL systems aim at flexibility --- some of their advocates believe that traditional models are now over
  - Models cannot be used in the same way as for traditional applications, but some ideas are meaningful
  - I will discuss two research experiences in the area

- A disclaimer:
  - Personal means that I will refer to projects in my group (with some contradiction, as currently my involvement in the topic, and in research in general, is decreasing…)

What is modeling for?

- Description at the appropriate level of abstraction
- Comprehension
- Communication
- Support to design and development (including coding and maintenance!)
- Performance management

Conceptual modeling

- Born in the database world, with great success for the design of traditional databases
- Spread in many neighboring fields

A neighboring field, a talk from ER 1998

Introductiory DB Course: slide 1 ☺

- DataBase Management System (DBMS)
  - System that handles data sets that are
    - big
    - persistent
    - shared

    guaranteeing
    - privacy
    - relaiability and fault-tolerance
    - efficiency
    - effectiveness

Relational DBMSs

- Born in the Seventies (more than 35 years ago)
- Effective and efficient for "business" applications (accounting, reservations, …) with simple, specific (but common) requirements:
  - persistency, sharing, reliability
  - data with simple structure and simple types (numbers, strings)
  - many short transactions with ACID properties (OLTP)
  - Possibly complex queries, with declarative specifications and "associative" access

In more technological terms

(Stonebraker & Cattel, CACM June 2011)

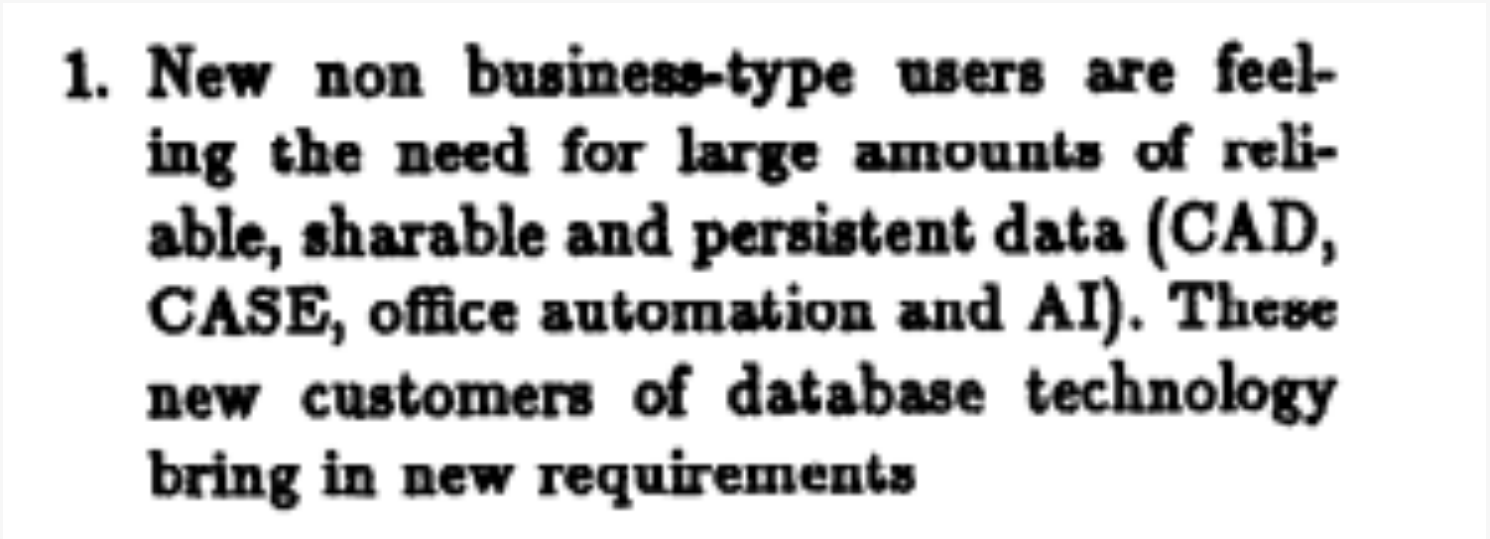
- "General-purpose traditional row stores"
  - Disk-oriented storage
  - Tables stored row-by-row on disk (hence, a row store)
  - B-trees as the indexing mechanism
  - Dynamic locking as the concurrency control mechanism
  - A write-ahead log (WAL) for crash recovery
  - SQL as the access language
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

Are we satisfied with relational databases?

- In many cases we are
- But in some cases we have always complained

A stament we have made for 25+ years ☺

- With the progress in computing systems new application requirements have emerged, for which relational technology is probably not adequate



1. New non business-type users are feeling the need for large amounts of reliable, sharable and persistent data (CAD, CASE, office automation and AI). These new customers of database technology bring in new requirements

(F. Bancilhon: Object-Oriented Database Systems. PODS 1988: 152-162)

New application areas (in 1988 ☺)

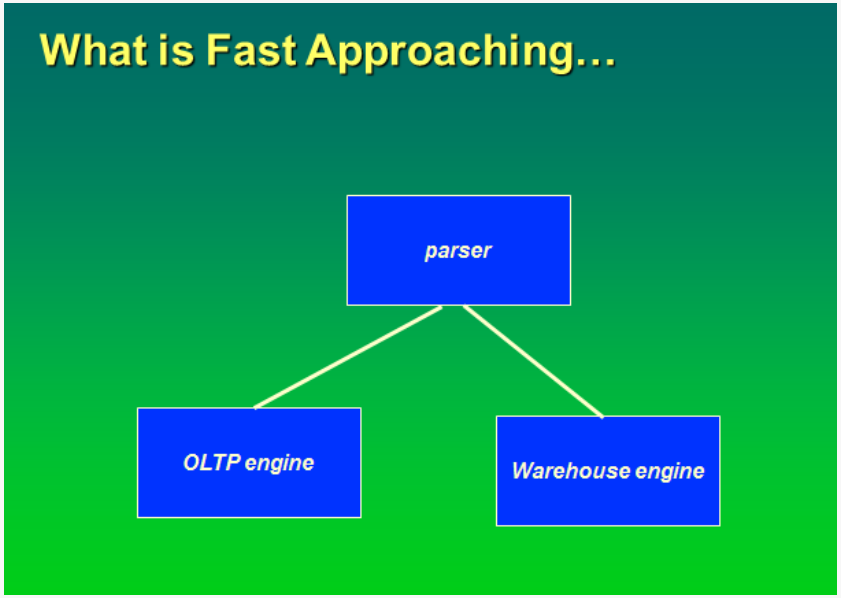
- Support to design and production
  - CASE (Computer-Aided Software Engineering)
  - CAD (Computer-Aided Design)
  - CAM (Computer-Aided Manufacturing)
- Document management
  - office automation and text management
  - Multimedia data
- More
  - science and medicine
  - AI systems

Another recurring claim

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

"One-size-fits-all"

- Indeed, relational systems currently offer one interface (SQL) but with various implementations, at least two, for the two major application families:
  - OLTP (On-Line Transaction Processing)
  - OLAP (On-Line Analytical Processing)



Probably mpre than two engines (in 2005 …)

Candidates for a Separate Engine

- OLTP
- Warehouses
- Stream processing
- Sensor networks (TinyDB, etc.)
- Text retrieval (Google, etc.)
- Scientific data bases (lineage, arrays, etc.)
- XML (argued by some)

… and later, after 2005

- Lack of satisfaction, in the Internet/Web world
  - rigidity of the relational model (emerged since the advent of the Web, only partially satisfied with XML)
  - need for scalability, for simple operations over huge quantities of data
- More generally:
  - "heaviness" of relational systems, in terms of performance and of administration

The NoSQL answer …

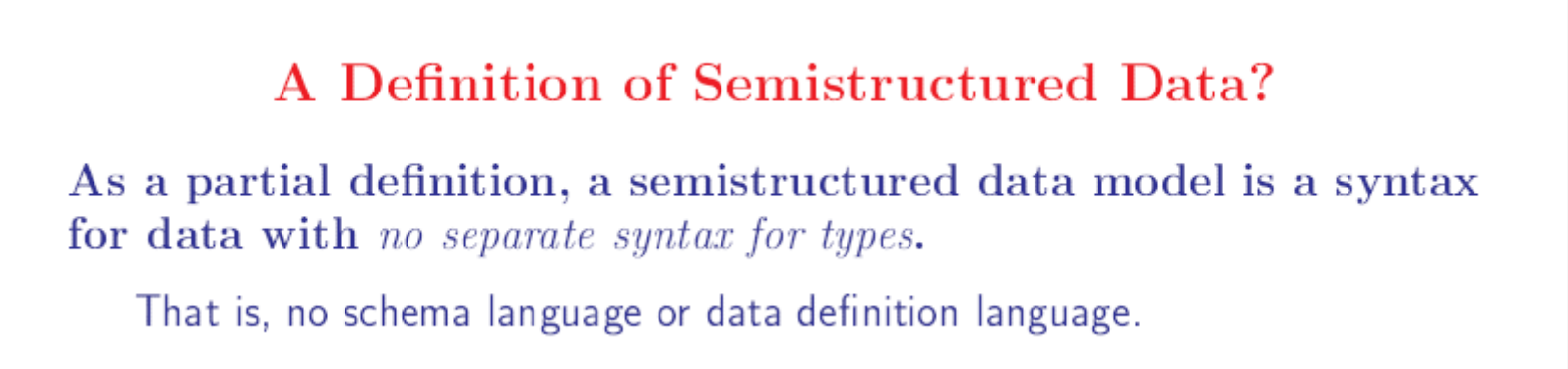
- New systems with
  - Scalability of simple operations over many nodes
  - Replication and distribution
  - Flexibility in data structure
  - New tecniques for indexing and main memory management
- Giving something away
  - A very simple application interface (much less powerful than SQL)
  - Less rigorous transaction management

- Let us see a few aspects

- New systems with
  - Scalability of simple operations over many nodes
  - Replication and distribution
  - **Flexibility in data structure**
  - New techniques for indexing and main memory management
- Giving something away
  - A very simple application interface (much less powerful than SQL)
  - Less rigorous transaction management

Flexibility in data structure

- Semistructured data, a concept studied in the Nineties, in the first Web era



A Definition of Semistructured Data?

As a partial definition, a semistructured data model is a syntax for data with *no separate syntax for types*.

That is, no schema language or data definition language.

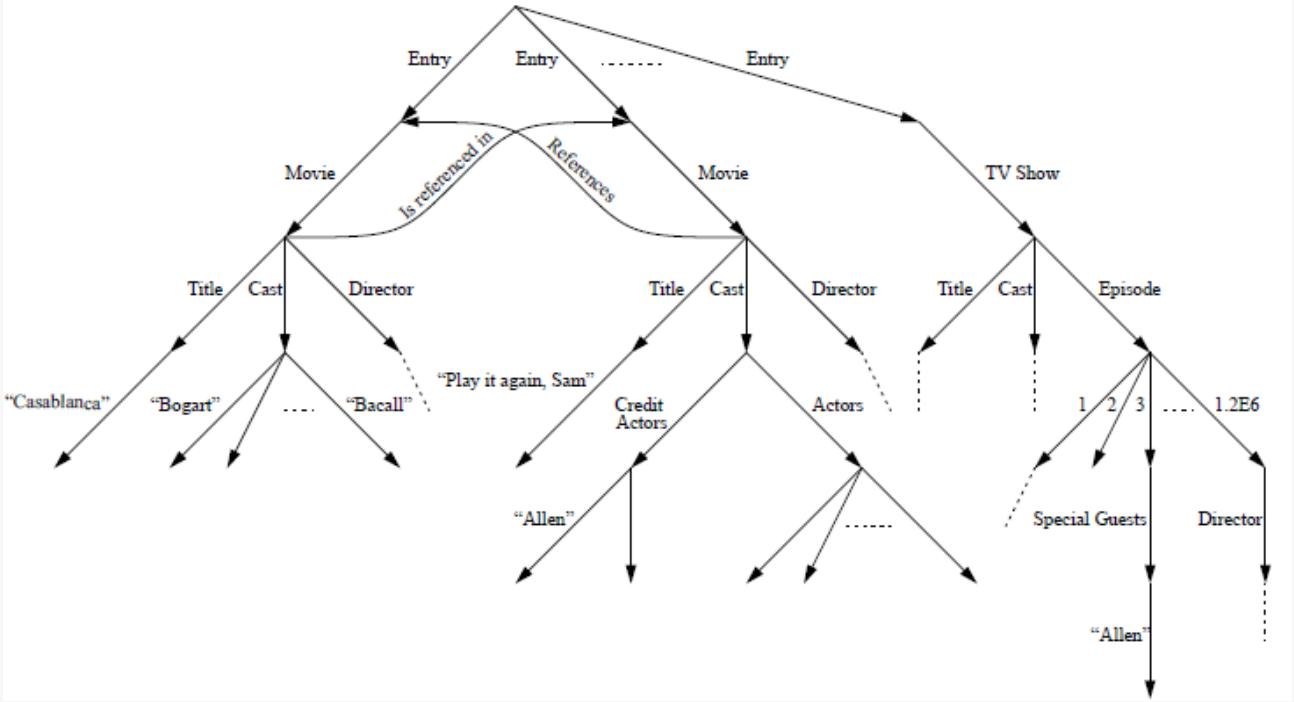
P. Buneman, tutorial, PODS 1997

http://db.cis.upenn.edu/DL/97/Tutorial-Peter/slides.ps.gz

Semistructured data, an example

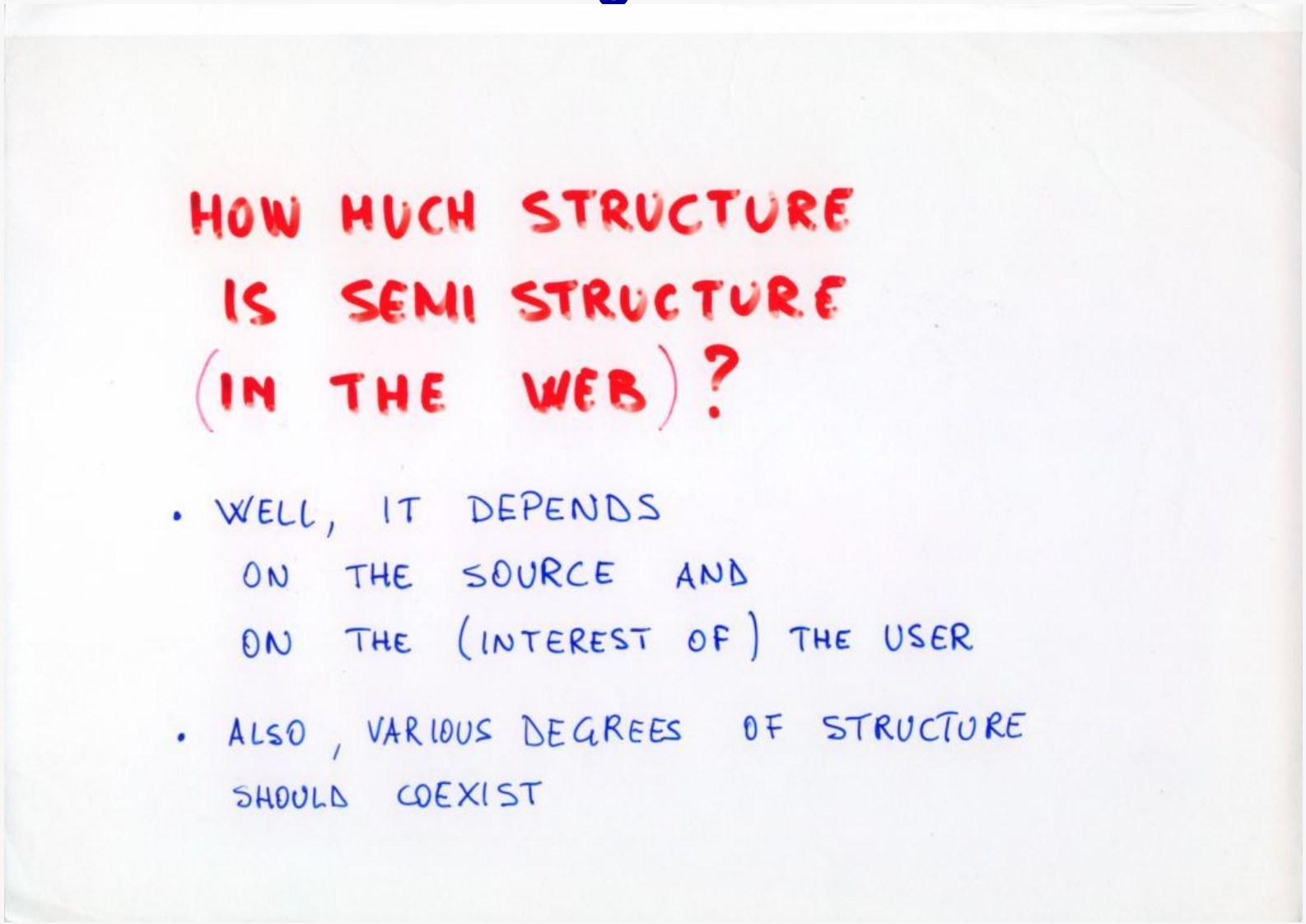
Semistructured data is usually "mostly structured". We are typically trying to capture data that has only minor deviations from relational / nested relational / object-oriented data. For example...

A Semistructured Movie Database

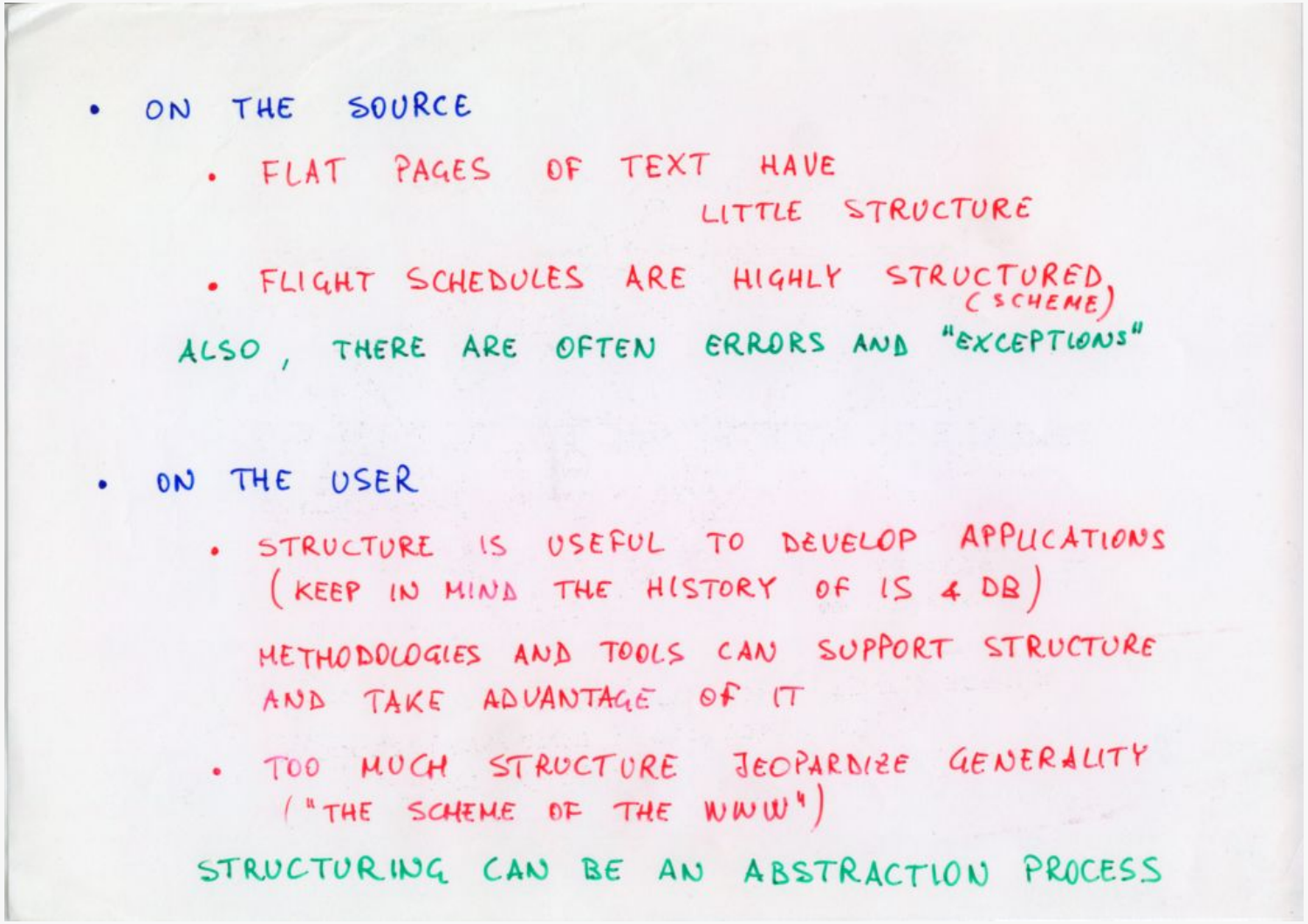


P. Buneman, tutorial, PODS 1997

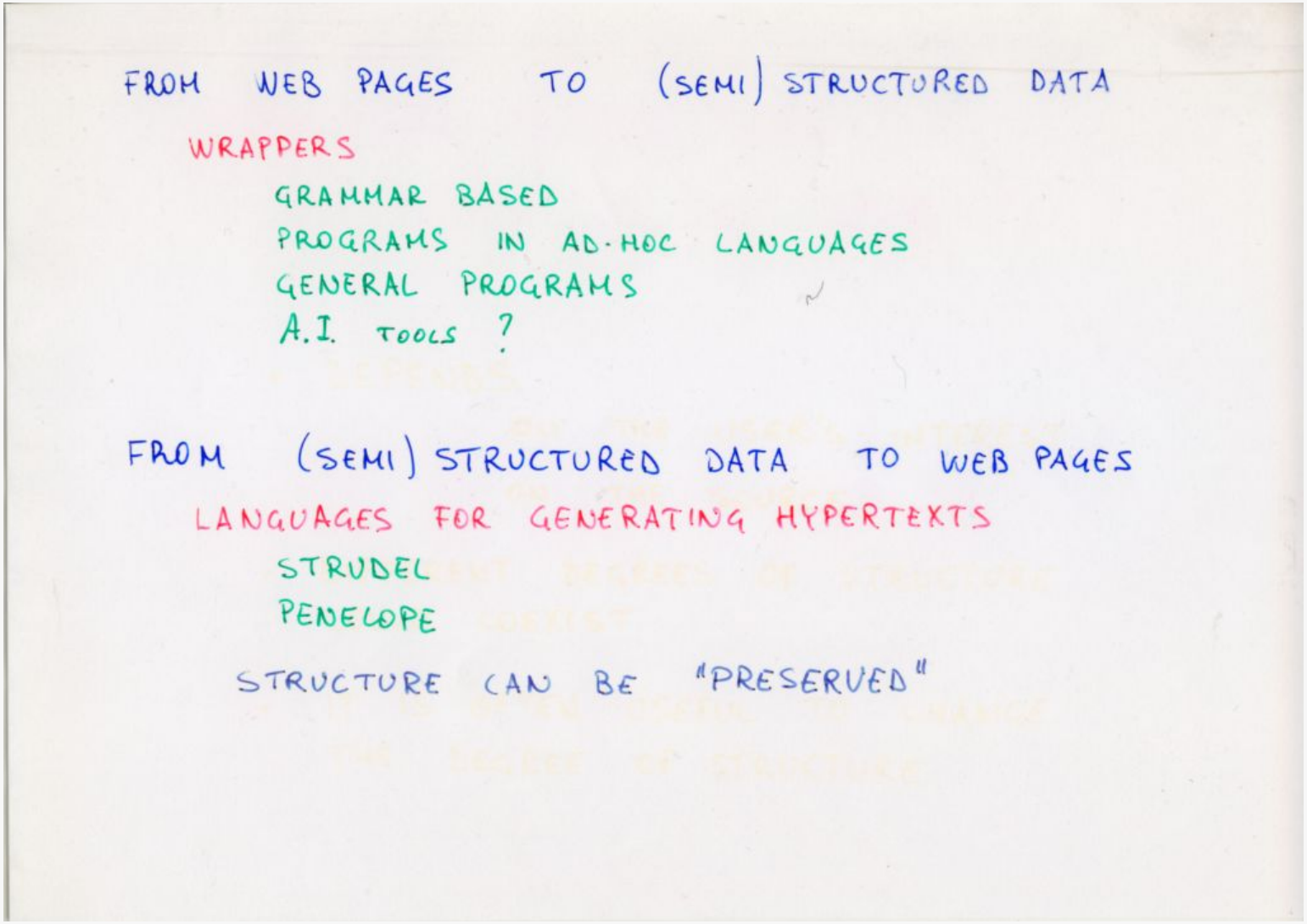
Workshop on Semistructured Data at Sigmod 1997



Workshop on Semistructured Data at Sigmod 1997 - 2



Workshop on Semistructured Data at Sigmod 1997 - 3



- New systems with
  - **Scalability of simple operations over many nodes**
  - **Replication and distribution**
  - Flexibility in data structure
  - New techniques for indexing and main memory management
- Giving something away
  - A very simple application interface (much less powerful than SQL)
  - Less rigorous transaction management

Scalability is easier if operations are local

- Scalability of reads can be obtained with replication
- Scalability of writes is usually obtained with sharding (and paying attention to replication)

"NoSQL" systems

- Work well for applications with operations that are simple and local
- Categories
  - Extensible record stores
    - BigTable, HBase
  - Key-value stores
    - Amazon Dynamo, Redis
  - Document stores
    - MongoDB, CouchDB
  - Graph DB
    - Neo4J
- There also SQL systems for operations that are simple and local (sometimes referred to as NewSQL)
  - MySQL Cluster, VoltDB, Clustrix, ScaleDB

NoSQL systems are different from one another

- **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
  - …

Heterogeneity is 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 application from a specific technology
- Reverse the canonical problem:
  - One size (DBMS) does not fit all (applications)…
  - …but one size (your application) should fit all (the DBMSs)

A problem we know

- In the same way as with traditional databases, we have heterogeneity
  - even more than we were used to (even more than with XML)

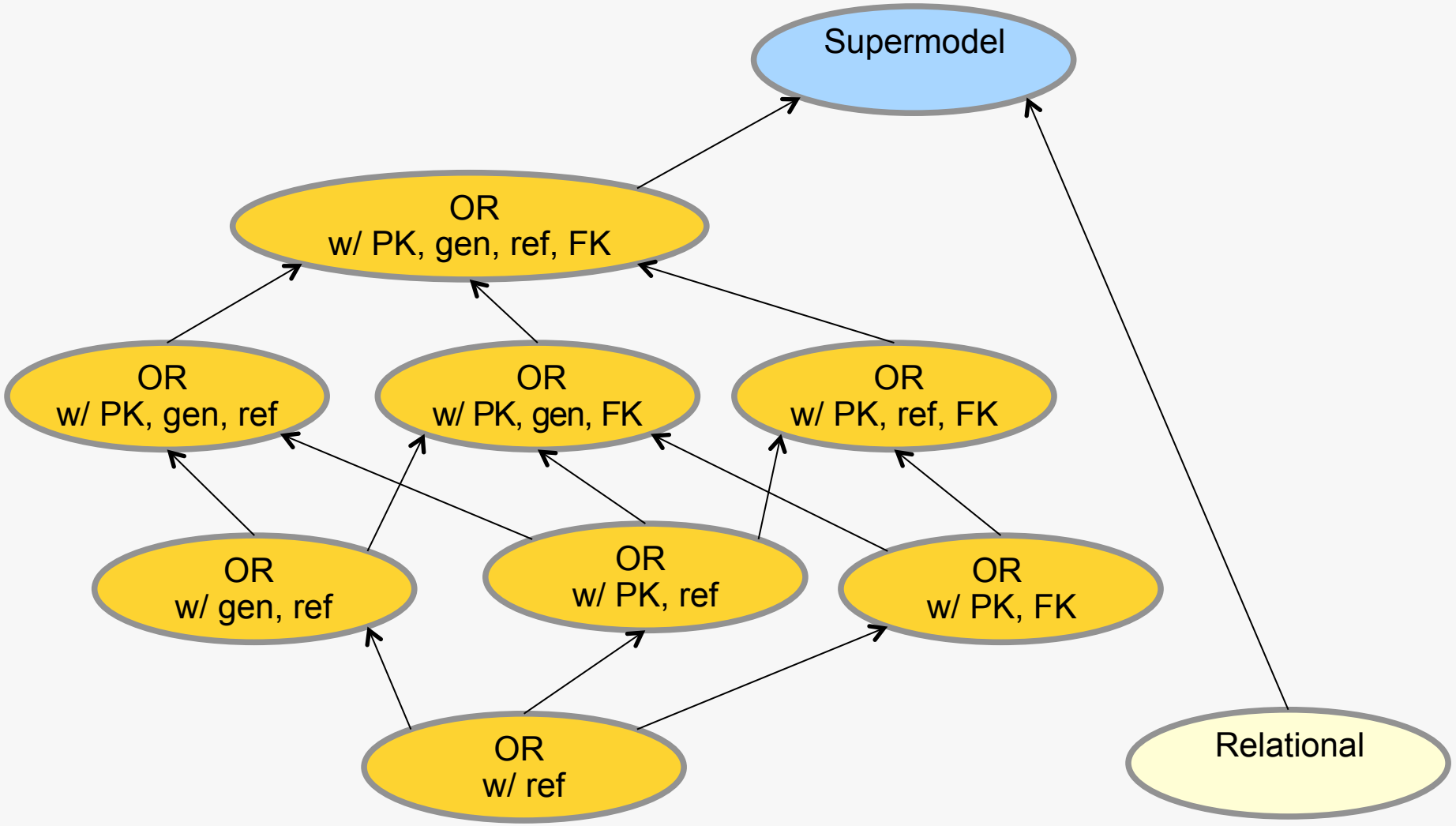
- Can modeling and abstraction help?
- We have experience in the area

MIDST
(Model Independent Schema and Data Translation)

(P. Atzeni et al. VLDB Journal 2008)

- Schema and data translation
  - initially with an off-line approach
  - later also with a run-time one
- Model-generic:
  - works for many models, in an extensible way
  - We have experimented with
    - Relational
    - OR, many variants
    - XSD
    - UML and ER (in many variants and extensions)
- Based on a lattice of models, with a most general one, the "supermodel"

A lattice of models



Issues in the NoSQL world

- Settings are not that much similar to those for traditional databases
  - Interfaces
    - are usually much simpler
    - have different "expressive power"
  - The structure of data is represented only to a certain extent (there is no notion of schema, and structure is usually very flexible)
  - Similarly, there is no notion of query language, nor a general pattern for queries

A supermodel based approach?

- In traditional settings, our idea was to have the supermodel as the most general model, at the top of a lattice
- Here, simplicity is a goal, even if objects could have some structure
- Also, while in databases data are "exposed" in full (and so there are powerful query languages that can exploit the structure), here operations are more focussed

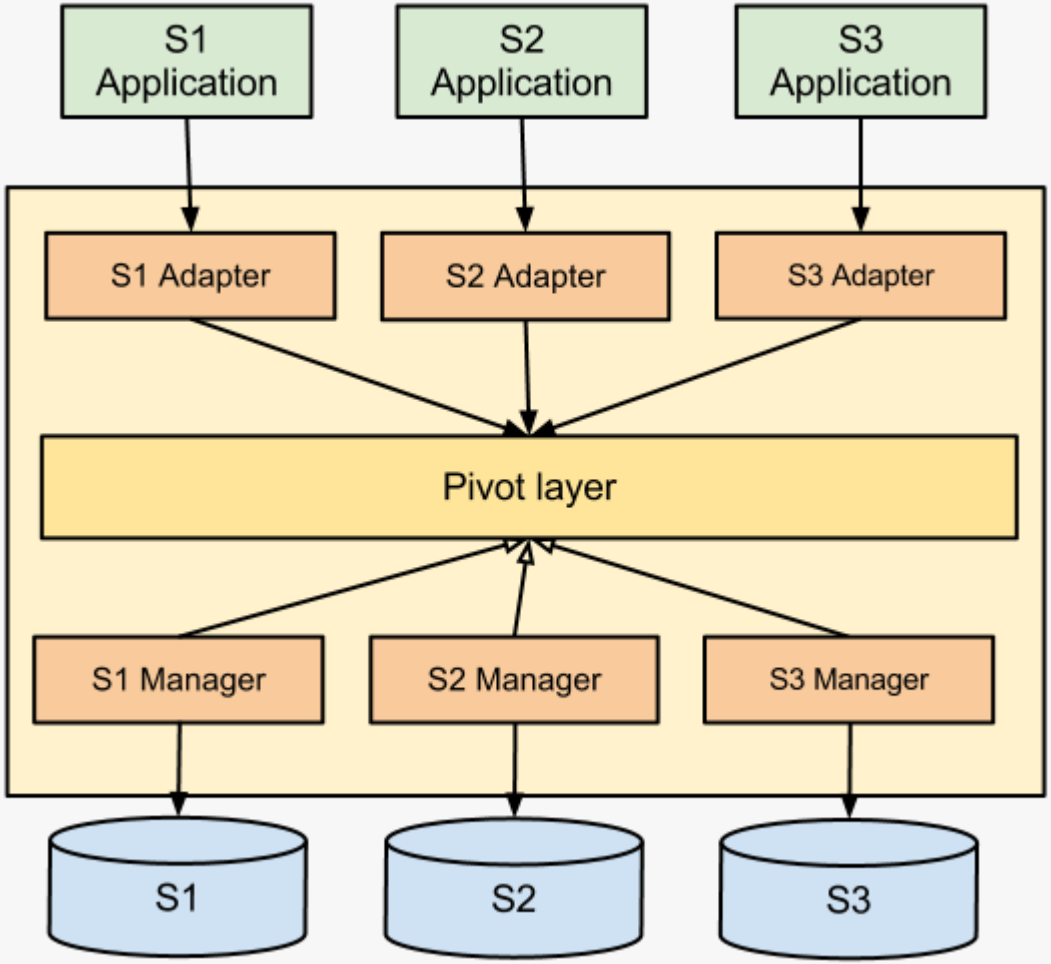
- Therefore, while in our previous approach we used as a "pivot" a very rich model, the supermodel, here a much simpler one would be needed

A contribution: SOS – Save Our Systems

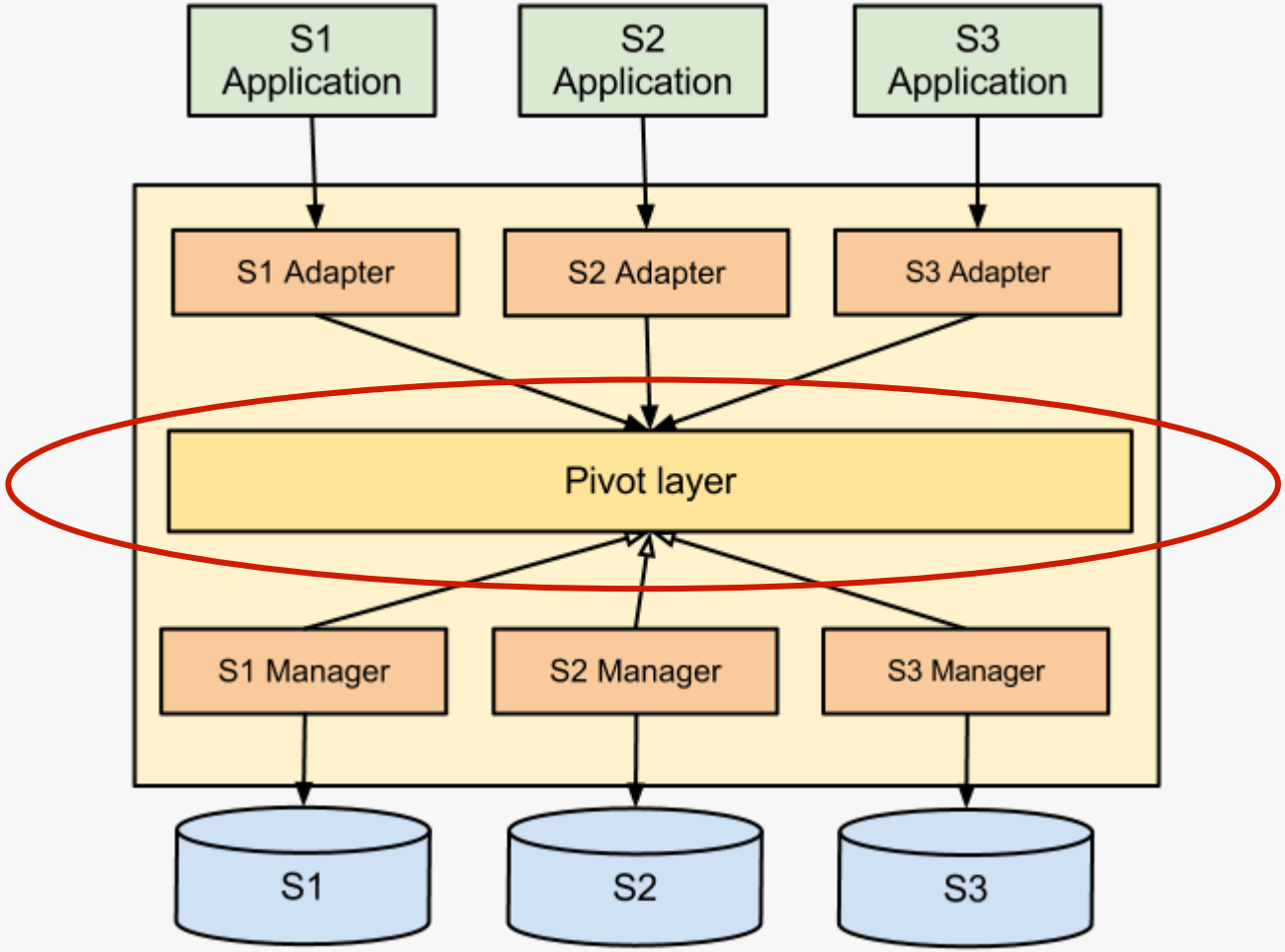
- **Goal:** seamless access to different NoSQL data stores.
  - Define **access**
  - Define **seamless**

- **Requirements:**
  - **Lightweight**: small footprint on performances
  - **Coherent**: with main NoSQL themes and features
    - Hint: do not reimplement SQL
  - **Scalable**: easily extendable to different technologies and DBMSes

A possible architecture



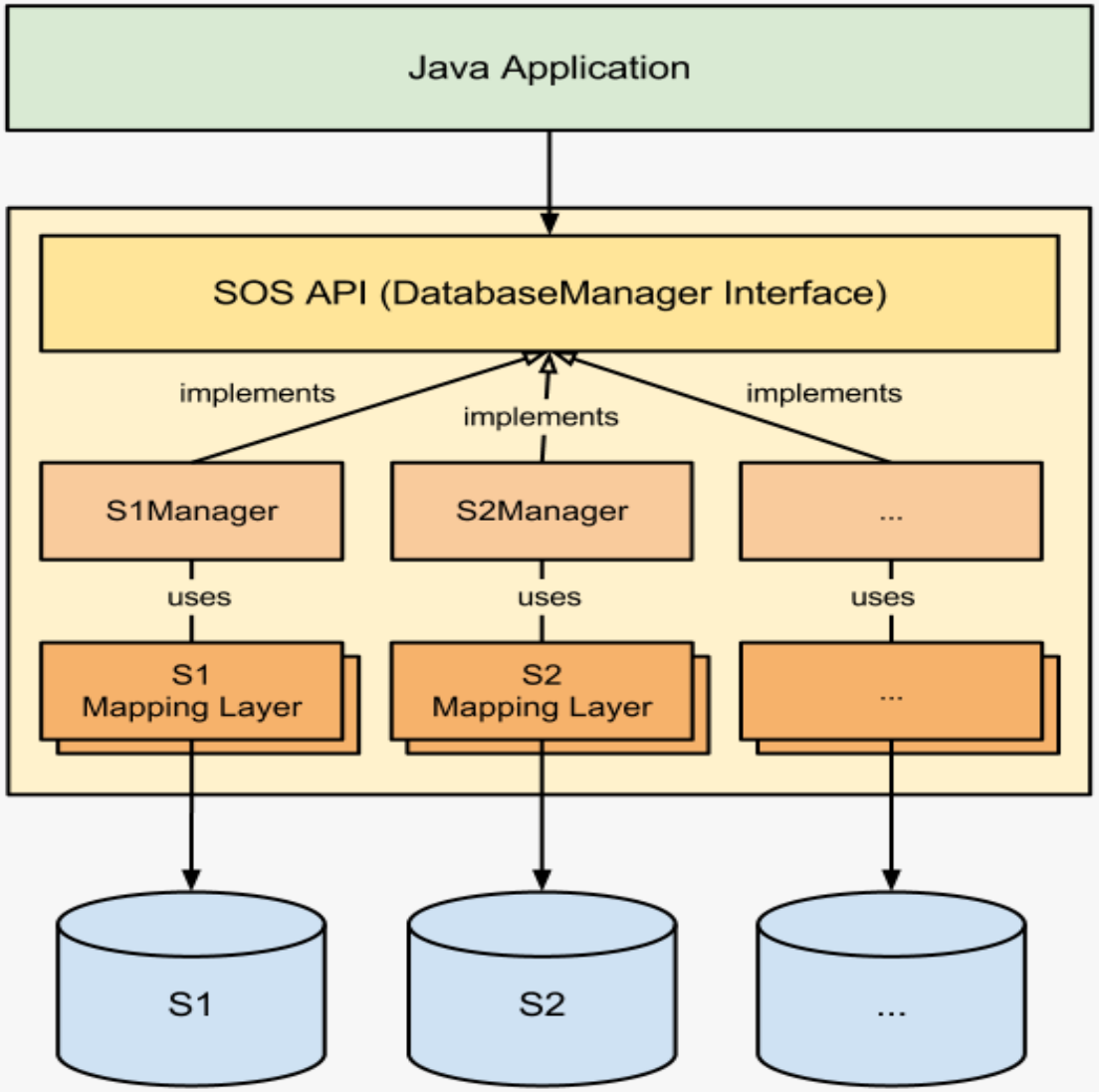
A possible architecture



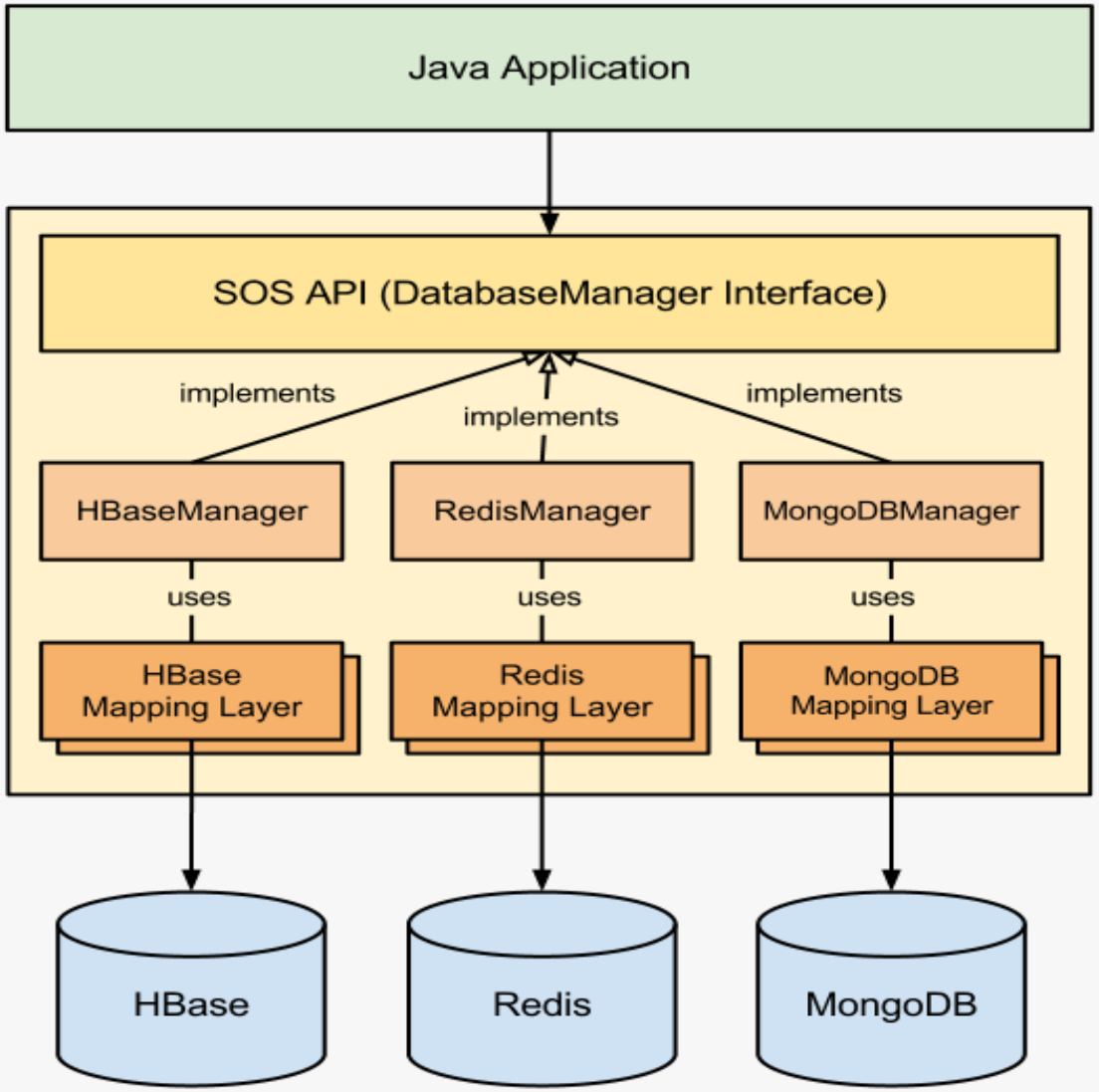
A simple pivot layer

- A common interface, with a simple set of methods involving single objects or (for retrieval) sets thereof
  - put
  - get
  - delete
- Motivation
  - the general, common goal of NoSQL systems is to support simple operations
- First implementation in Java

SOS: Save Our Systems



SOS, concretely



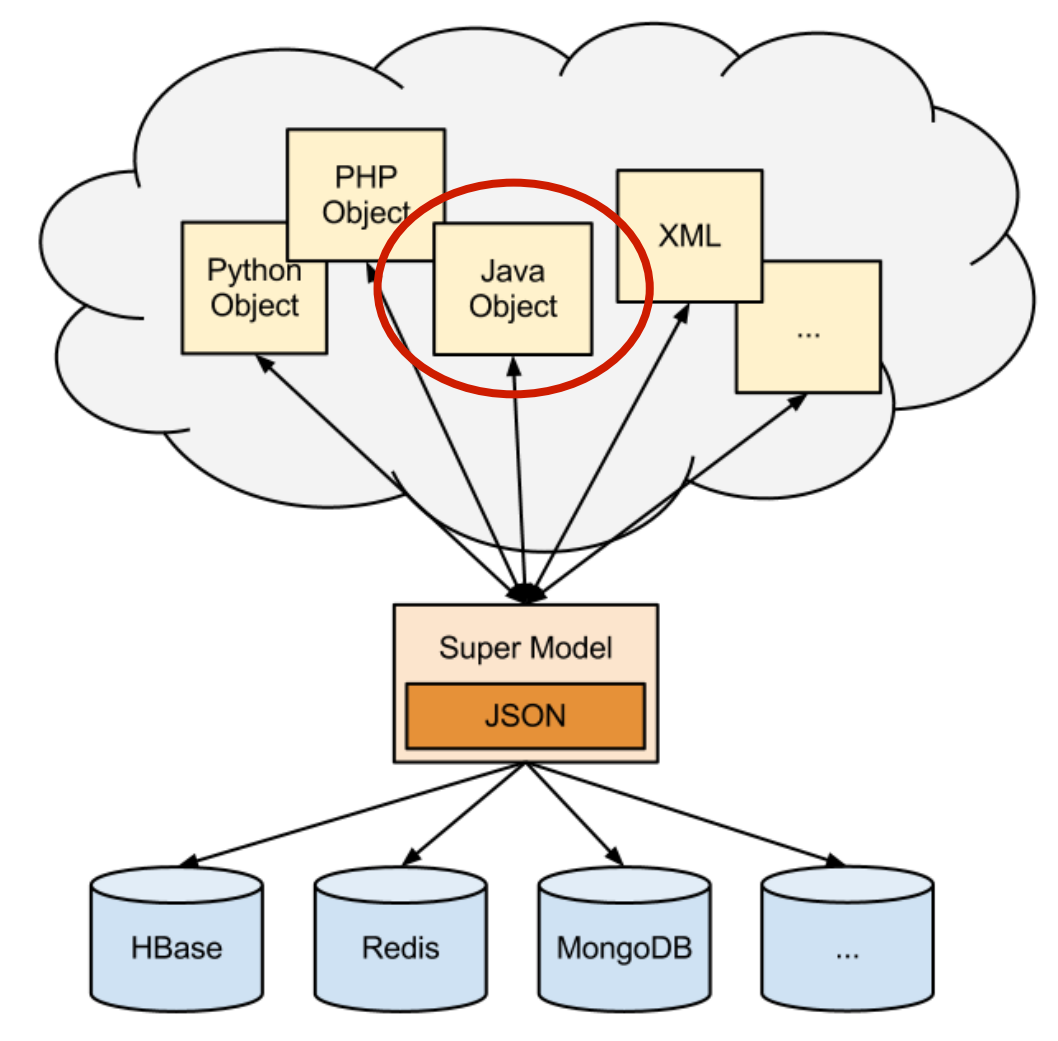
Issues

- Do objects have a structure? Should we handle it?
- How much sophisticated is the retrieval (get) operation?

Object structure

- In general, to support a very basic interface, we could just treat objects as blobs, serializing them
- However, objects often have a complex structure, which can be modeled in tree form, with sets and structures, possibly nested, as well as simple attributes
- Our interface gets the native objects and the implementation serializes them into JSON

Implementation of the structure



The get operation

- Various forms in mind
  1. `Object get (String collection, String ID)`
  2. `Object get (String collection, Path p)`
  3. `Set<Object> get (Query q)`
- Currently, the first two implemented
  1. Straightforward
  2. Currently retrieval of simple fields, in the future reconstruction of objects
  3. Many interesting challenges, related to query processing and performances

In summary, at the interface level

- The notion of a model has been useful, to some extent

Another perspective

- Models are also useful for defining abstractions at lower levels, so to handle performance issues, at a level that is a bit higher than the actual implementation one
  - so that arguments hold (at tleast to a certian degree) for different systems

Data organization in NoSQL systems

- "Simple complex" objects
  - Complex because they often have a nested stucture, and involve contraints
  - Simple as they are handled as a whole
- Scalability works well because there are many small objects and operations do not involve many of them (no joins, no complex distributed transactrions)
- So an interesting goal could be to be able to find the "optimal" degree of "smallness", so that the objects are large enough to be coherent and small enaough to allow for parallelism and scalability

"Tricks" in NoSQL applications

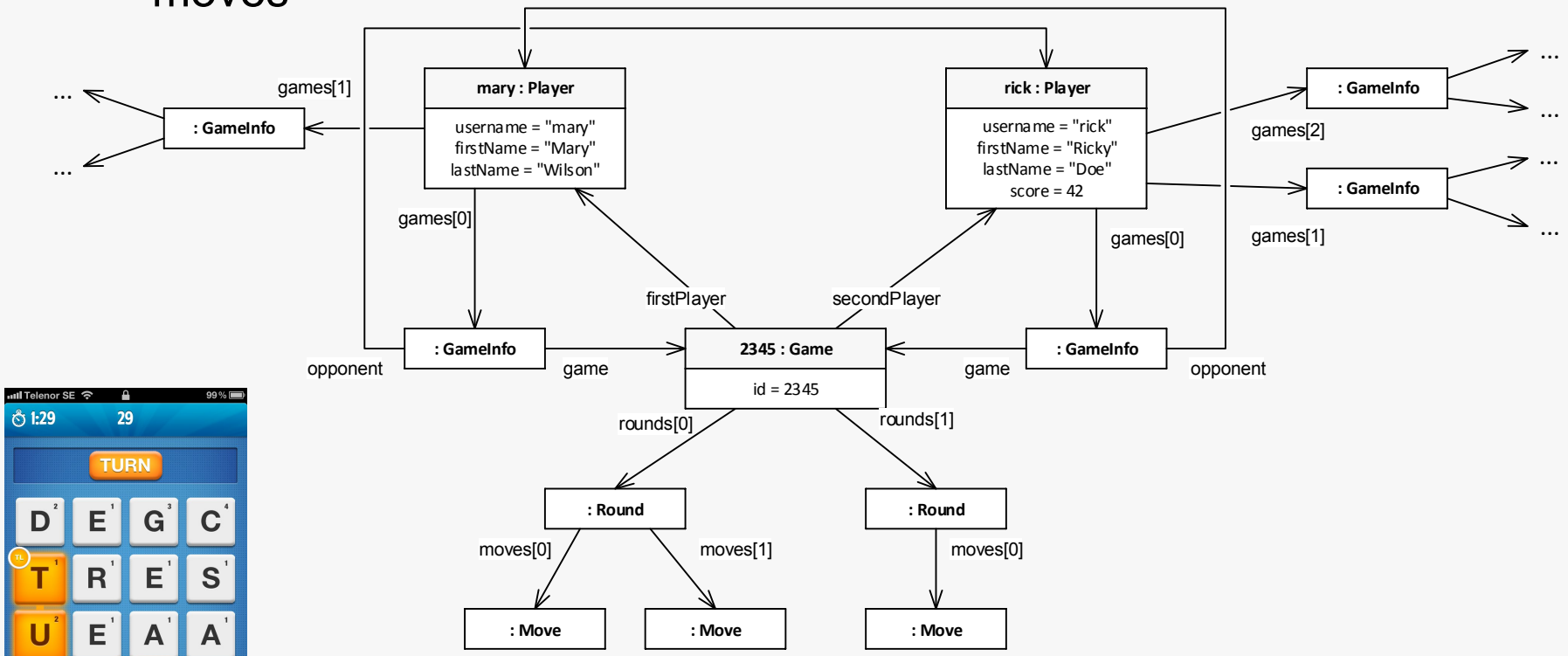
- Developers often encode structure descritpion in keys, with practices that common but not well documented ..
- It would be worth to support their management, hiding the dirty aspects …

Encoding structure in keys, an example

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]	{ … }
…	…

An example

- A fictious online, web 2.0 game which should manage various application objects, including players, games, rounds, and moves



Implementing the game

- We need to manage various application objects, including players, games, rounds, and moves
  - 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?

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

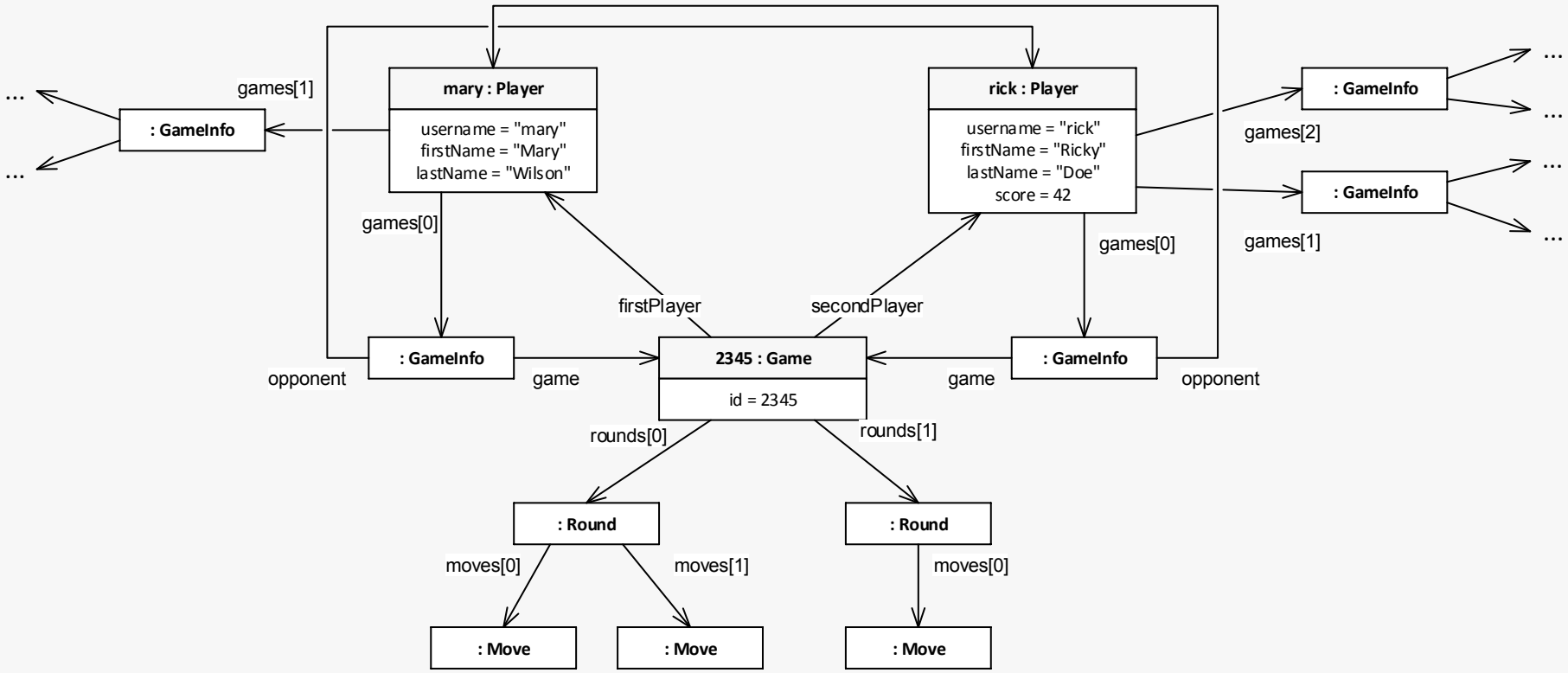
- We proposed the **NoAM approach to NoSQL database design**
  - tailored to the requirements of next-generation web applications
  - based on the NoAM abstract data model for NoSQL databases
  - a high level/system independent approach – the initial design activities are independent of any specific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

- 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

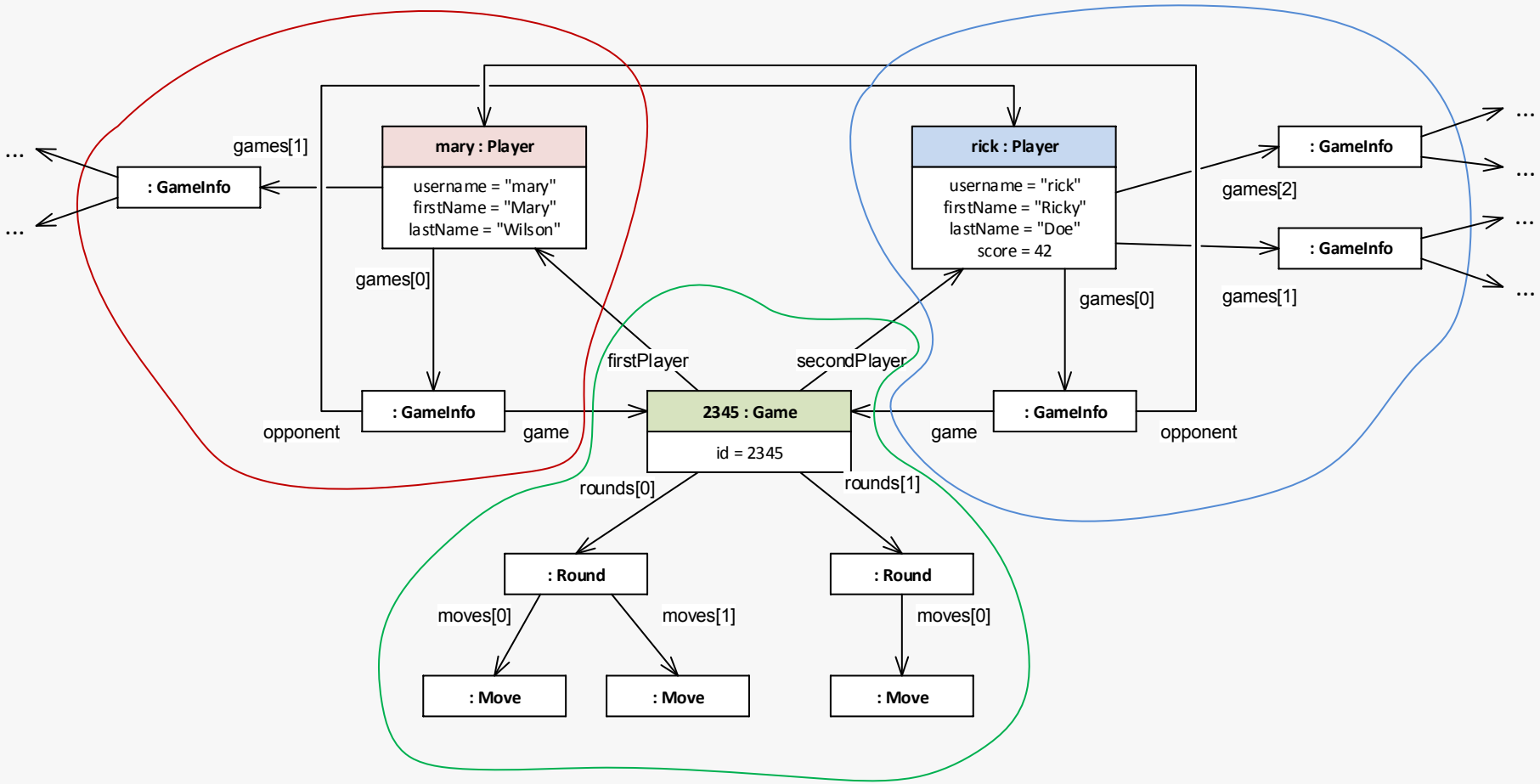
Application data

- 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…

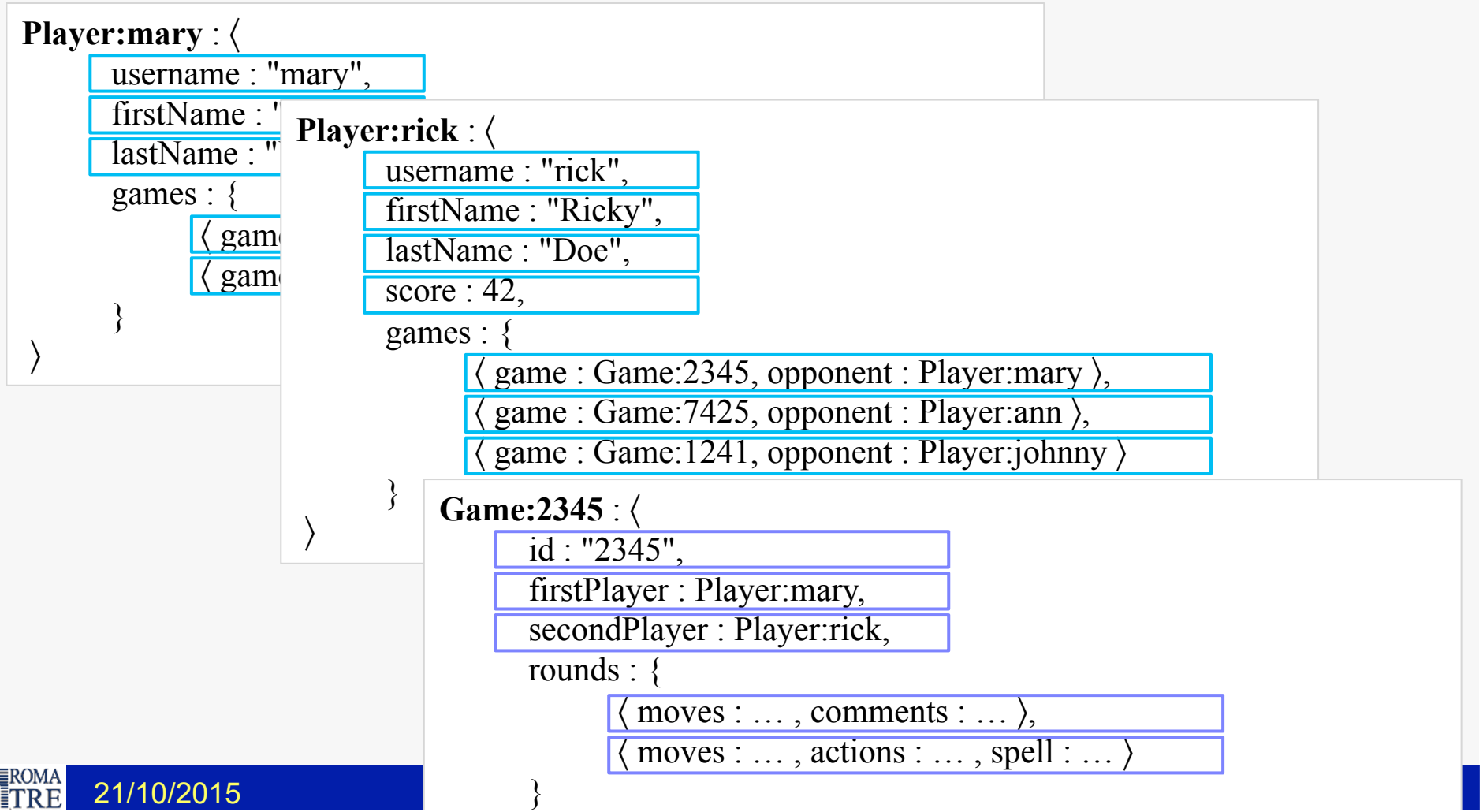
Player:mary : ⟨
username : "mary",
firstName : "
lastName : "
games : {
⟨ game
⟨ game
}
⟩

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!*) …



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]	{ … }
…	…

- 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

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*

Conclusions

- We argued tha models can be useful in general in area that does not consider them much
- We illustrated two experiences:
  - an interface to overcome and handle heterogeneity
  - a design methodology that considers performances

Thank you!