Experimental Performance Evaluation of Different Data Models for a Reflection Software Architecture over NoSQL Persistence Layers

Sara Fioravanti
University of Florence
Florence, Italy
sara.fioravanti@unifi.it

Simone Mattolini
University of Florence
Florence, Italy
simone.mattolini@unifi.it

Fulvio Patara
University of Florence
Florence, Italy
fulvio.patara@unifi.it

Enrico Vicario
University of Florence
Florence, Italy
enrico.vicario@unifi.it

ABSTRACT

The recent rise of the NoSQL movement motivates investigation on the performance impact that new persistence approaches can bring in the model-driven re-engineering of a consolidated object-oriented software architecture.

We report comparative experimental performance results attained by combining a pattern-based domain logic with a persistence layer based on different paradigms and we describe how data model is persisted in various implementation based on MySQL, Neo4j, and MongoDB.

Keywords

Model-driven performance engineering, Reflection pattern, Relational databases, NoSQL databases, MySQL, MongoDB, Neo4j, Electronic Health Record (EHR) systems.

1. INTRODUCTION

Non-functional requirements of *changeability* and *adaptability* [1] have primary relevance for a large class of software intensive systems that are intended for managing great volumes of data with a high degree of variety in the structure of contents. The attainment of these qualities can be largely facilitated by the assumption of a tailored software architecture.

In particular, the *Reflection* architectural pattern [2] provides a mechanism that allows for dynamically changing data structure and system behaviour at run-time [3]. To this end, the domain logic is modeled using two different levels of abstraction: a *meta* level provides a self-representation of the system encoding knowledge about data type structures, algorithms, and relationships; besides a *base* level application logic carries concrete data whose interpretation is determined by the values of so-called *meta-objects*.

The *Observations & Measurements* analysis pattern [4] implements the reflection principle specializing the abstraction for the case of high variety in the data attributes of object types. This pattern-oriented architectural design brings a number of further benefits, mostly linked to the quality of the code, and notably to maintainability, reusability, and consolidated understanding of implementation choices and consequences.

However, design by patterns does not account for performance as first-class requirement, and naturally incurs in well-known performance anti-patterns [5, 6], which may become crucial when *volume* and *variety* must meet also *velocity* [7]. These drawbacks are largely exacerbated when the domain logic is persisted over a relational storage layer, due to the nature of the domain model and its mismatch with the relational tier [8].

In general, the persistence of a domain model with complex structure into a relational database comes with a number of performance penalties, that translate in longer time required for key persistence operations. These issues can be partially mitigated with ad-hoc optimizations in the design of the relational database [9], pertaining to the choice of a particular representation for class inheritance, the use of auxiliary tables to store additional information, and the smart use of data fetching.

The interposition of an object-relational mapping (ORM) layer between the domain logic and the storage layer can mitigate this problem. In the practice of development of Java enterprise applications, Java Persistence API (JPA) specification represents a mature and state-of-the-art ORM solution which grants many benefits [10]. First of all, it allows to persist domain classes with a minimal boilerplate code, thanks to simplified annotation facilities. Also, it provides full integration with the Java application stack, composed by other technologies such as EJB (for encapsulating the business logic) and CDI (for implementing the *Inversion of Control* pattern [11]). However, JPA further increases the degree of indirection and this can have negat-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPE'16, March 12-18, 2016, Delft, Netherlands
© 2016 ACM. ISBN 978-1-4503-4080-9/16/03...$15.00
DOI: http://dx.doi.org/10.1145/2851553.2851561

ive effects on the system performance, also due to the loss of design control on the impact that domain logic operations have on the storage process.

With the rise of the *Not Only Sql (NoSQL)* movement [12], other options in the design of the storage layer are now available, and provide various advantages, including reduced access time through the clustering of similar data [13], and increased adaptability to the variety and variability of data over time through the use of a *schemaless* structure. This motivates the investigation on engineering the performance of existing applications by changing the storage schema from a relational + ORM persistence stack to a NoSQL solution, while preserving the domain logic structure. In particular, this subtends a problem of re-modeling content representation in the schema of some NoSQL technology and quantitatively evaluating the performance gain that can be attained. In so doing, different NoSQL paradigms are more or less close to the domain model and suited for its main operations [14, 15], and a pattern-based organization of the domain logic can drive the refactoring of the data model towards more efficient performance results.

In this paper, we report on the performance engineering of a three-tier web-application focused on the replacement of a relational + ORM persistence stack through two different NoSQL technologies, describing how a reflection-based architecture can be modeled over the graph-oriented *Neo4j* [16] and the document-oriented *MongoDB* [17] databases, and comparing experimental performance results achieved by the different solutions.

To this end, in Sect. 2, we describe a reflection-based architecture that combines the *Observations & Measurements* and the *Composite* patterns to attain a high degree of adaptability and changeability and that is persisted over a relational + ORM stack (Sect. 2.1), and we describe how this was concretely exploited in the implementation of an Electronic Health Record (EHR) system [18, 19] which is in use since various years in a major Italian hospital (Sect. 2.2). In Sect. 3, we discuss how the domain model of the reflection architecture can be suitably represented over *Neo4j* and *MongoDB* databases (Sect. 3.1, 3.2) and we show how these representations are *information equivalent* to the original relational representation (Sect. 3.3). In Sect. 4 we report the result of experimentations aimed at measuring the performance gain, compared to the actual implementation, referred to a crucial application use case, applied on *real* data taken from the practice of use of the EHR system and on *synthetic* data generated so as to stress the most relevant dimensions of complexity. Performance results obtained in both datasets show a clear gain in performance by the MongoDB solution, and more generally, a better scalability of NoSQL technologies when the complexity of the data structures increases. Conclusions are drawn in Sect. 5.

2. A REFLECTION ARCHITECTURE FOR ADAPTABILITY

In this section we describe how the *Reflection* architectural principles [2] can be implemented through a powerful combination of the *Observations & Measurements* analysis pattern [4] and the *Composite* pattern [20] to implement an EHR system able to deal with medical concepts and clinical data characterized by complexity and volatility.

2.1 Exploiting the Reflection, Observations & Measurements, and Composite patterns

The *Reflection* architectural pattern [2, 3] permits the development of a domain logic with a high degree of changeability and adaptability [1] through a mechanism that allows for changing structure and behaviour of objects at run-time. To this end, the domain logic is split in two layers so as to support dynamic adaptation of the system in response to changing requirements. On the one hand, a *meta* level consists of a set of meta-objects providing information about system properties. On the other hand, a *base* level models the business logic and uses information provided by the meta level, in order to make the system more flexible when changes occur.

The *Observations & Measurements* analysis pattern [4] comprises an embodiment of the *Reflection* pattern, where meta-objects are used to create abstraction on the attributes carried by different object types. In this case, *measurements*, that allow to record quantitative information, and *observations*, that extend the expressiveness of the pattern for taking into account qualitative information, are both represented in a so-called day-to-day *operational* level. Their configuration, in terms of semantic definition, is constrained by a so-called *knowledge* level, where changes are typically more infrequent.

Hierarchical structured data resulting from repeated aggregation of basic observations and measurements can be cast in the representation through a mix-in of the *Composite* pattern [20], by allowing an observation or measurement be implemented as a collection of references to other observations or measurements.

Observations & Measurements has been frequently advocated as a scheme of great potential in the development of a variety of applications that are supposed to collect data with different structures and different versions over time. A notable class of applications, with major practical and economical impact, occurs in the creation of Electronic Health Record (EHR) systems [21]. In general, an EHR system [18] is a kind of Health Information Management (HIM) system supporting the acquisition, analysis, and maintenance of clinical information items about a patient, in digital or traditional form. For such a system, changeability and adaptability [1] are qualities of primary relevance, which largely condition the ability of a software product in fitting the needs of different medical specialities, and in accompanying their evolution over time due to changes of local organizational assumptions or even of the patient's health status.

In the rest of this Section, we describe a concrete software architecture that combines the *Observations & Measurements* pattern with the *Composite* pattern. This pattern-

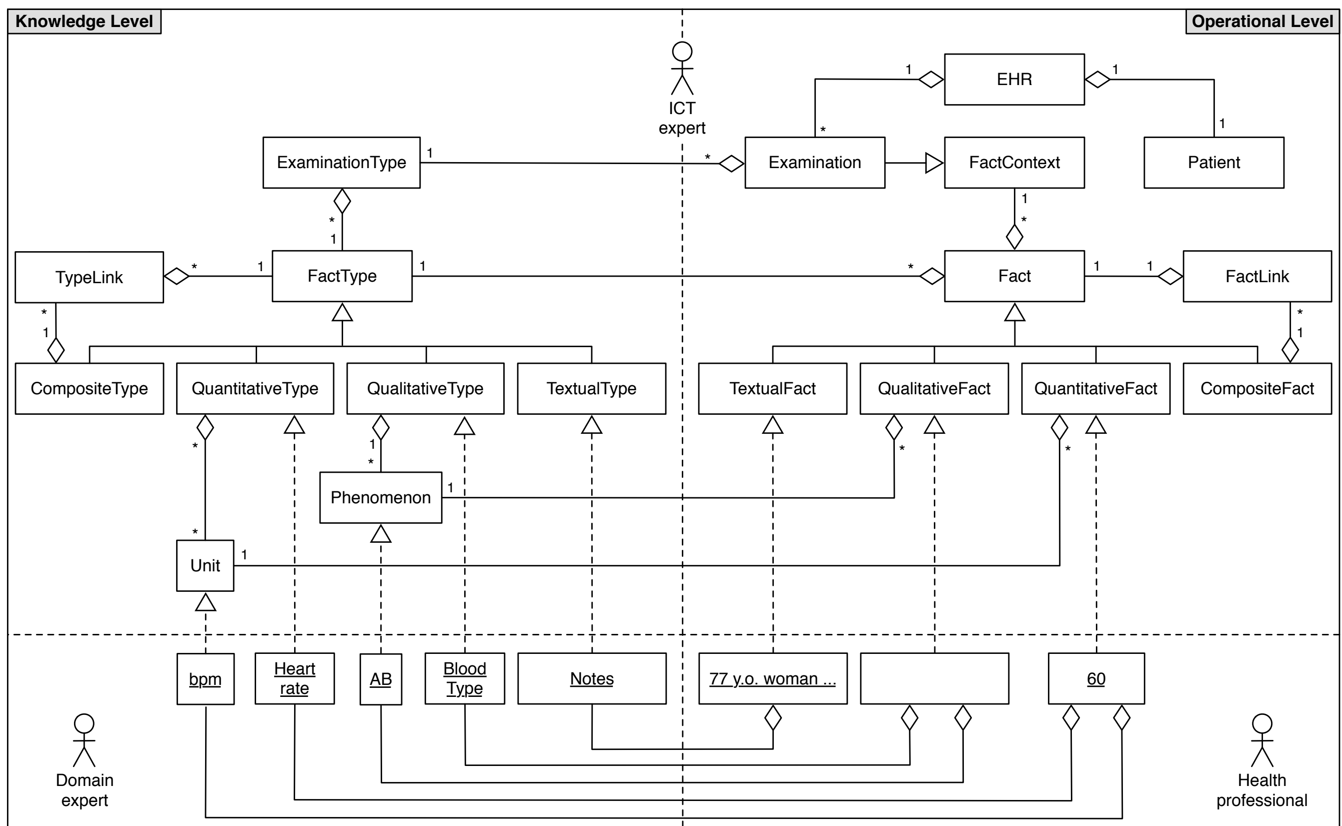

Figure 1: The domain model of the *Empedocle* EHR system based on the underlying meta-modeling paradigm, addressed in the architectural perspective by the *Reflection* pattern [2] and in the conceptual perspective by the *Observations & Measurements* pattern [4]. The meta-modeling approach allows a new medical concept to be accounted in the EHR system just through the instantiation of a new object from the class `FactType`, avoiding the need of programming new classes or class members and without any impact on the database schema or on its records. In the same manner, a new clinical concept can be recorded just through the instantiation of a new object from the class `Fact`.

based architecture was implemented within an EHR system named *Empedocle*, which is in use since more than 3 years in various units of the major hospital of Tuscany Region (Careggi hospital, in Florence). While referring to this case for the sake of experimentation concreteness, most of the subsequent discussion about the development of a graph-oriented or document-oriented database representation as well as about their impact on system performance are more generally applicable to most schemes that can be designed in the style of the *Reflection* architectural pattern.

2.2 The Empedocle EHR system

The UML class-object diagram of Fig. 1 provides a high-level specification of the domain model implemented in the core of the *Empedocle* EHR system.

At the *operational* level, an `EHR` represents a structured collection of health information items about a `Patient`, derived through a set of clinical `Examinations`. Specifically, during each `Examination`, a series of clinical information items like signs (i.e. objective evidences noticed), symptoms (i.e. subjective evidences reported by patient), and other clinical observations are captured by health professionals as instances of the `Fact` class.

Conversely, the *knowledge* level must be designed so as to accommodate the intrinsic variability of the medical domain, which depends on the evolution over time as well as on differences among medical specialities. To this end, all medical concepts can be defined directly by domain experts as instances of the `FactType` class.

The resulting high-level model abstraction allows to separate the representation of medical knowledge (i.e. the semantic of medical phenomena) from clinical data (i.e. the value assumed by a specified medical phenomenon in a specified time for a specified patient), and empower domain experts to contribute to this knowledge in the course of system life. Accordingly, four different categories of knowledge can be identified: `TextualType`, for free-text information (e.g. patient's *anamnesis*); `QualitativeType`, for values in a finite range of acceptable `Phenomena` (e.g. *blood type* with groups *A*, *B*, *AB*, and *0*); `QuantitativeType`, for quantities with a specified set of acceptable `Units` (e.g. *heart rate*, measured in *beats-per-minute*); and `CompositeType`, for composing `FactTypes` in a hierarchical structure through a *Composite* pattern implementation (e.g. *vital sign* including *temperature*, *blood pressure*, *heart* and *respiratory rate*). The same categories can be identified at the operational

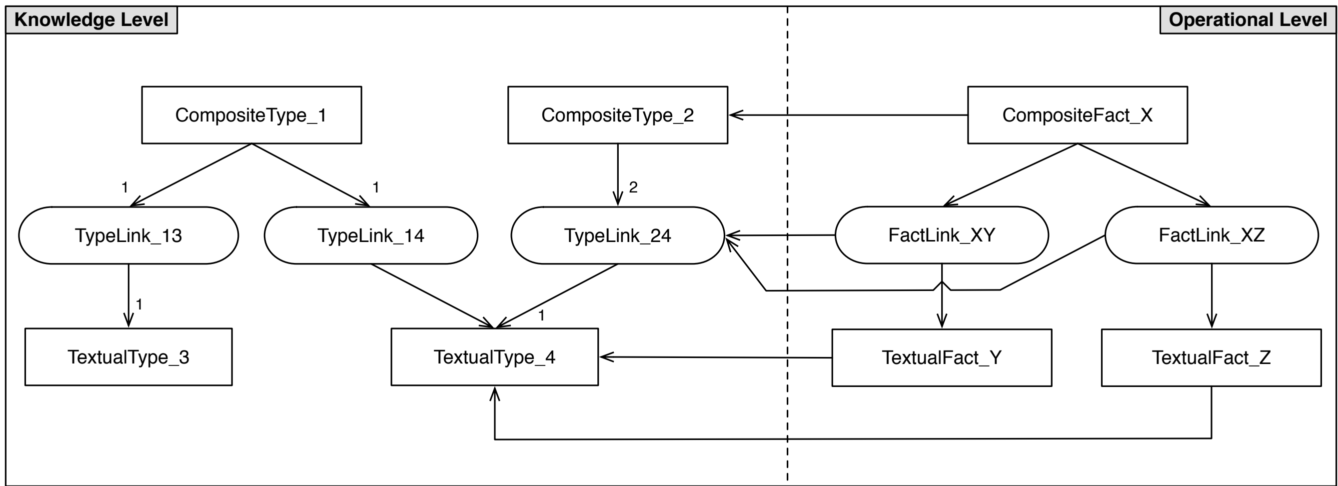

Figure 2: An example of `Examination` structure as represented using the domain model shown in Fig. 1: on the left, a direct acyclic graph obtained composing `FactTypes` and `TypeLinks`; on the right, a tree-like structure as resulting from the composition of `Facts` and `FactLinks`. Note that rectangles represent instances of `FactType` and `Fact` classes and define, respectively, medical concepts and clinical observations that are to be taken into account during a clinical examination. Rounded boxes represent instances of `TypeLink` and `FactLink` classes and are used to increase the expressiveness of each `Type`-to-`Type` and `Fact`-to-`Fact` association (for example, through the definition of its cardinality).

level: `TextualFact`, `QualitativeFact`, `QuantitativeFact`, and `CompositeFact`.

`ExaminationType` class represents the structure of an `Examination` in terms of which `FactTypes` (and related `Facts`) have to be considered during a medical examination; moreover, it specifies, through `TypeLink` and `FactLink` associations, the multiplicity of occurrence of each `Fact` in order to dynamically adapt the structure to multiple contexts-of-use that require a different number of instances to be recorded. In addition, the reuse of already defined *named* `FactTypes` is supported, so as to avoid their proliferation, just referencing them in multiple parts of the structure. Alternatively, *anonymous* `FactType` instances (i.e. `FactTypes` that do not need to be referenced by others) can be used, and the definition of their structures is directly included inside the parent structure. As relevant consequence, as depicted in Fig. 2, the `FactType` structure will result in a direct acyclic graph, while the derived `Fact` structure will result in a tree, usually with an increased number of nodes due to the multiplicity attribute.

This implies a more complex data model, with various drawbacks. On the one hand, while the number of `Facts` concretely recorded at run-time during a clinical session is bounded in semantic and multiplicity by the `FactType` definition, the real depth of an `Examination` cannot be known in advance, precluding the possibility to exploit optimized adhoc mechanisms for retrieving all the data, requiring instead to explore the entire structure. On the other hand, since the model is split in two levels, the whole `Examination` will be completely known only when both parts will be provided. For this reason, retrieving all the data collected during an `Examination` is not restricted to exploring the `Fact` tree, but requires to explore the related `FactType` graph, affecting system performances. Finally, the resulting model consists of a relative small number of classes for representing only concrete concepts; nevertheless, the high degree of abstraction is counterbalanced by the instantiation, at run-time, of an increased number of objects required for describing the actual domain. Usually, this does not represent a problem in small and static domains, but it becomes evident in domains characterized by complexity and volatility.

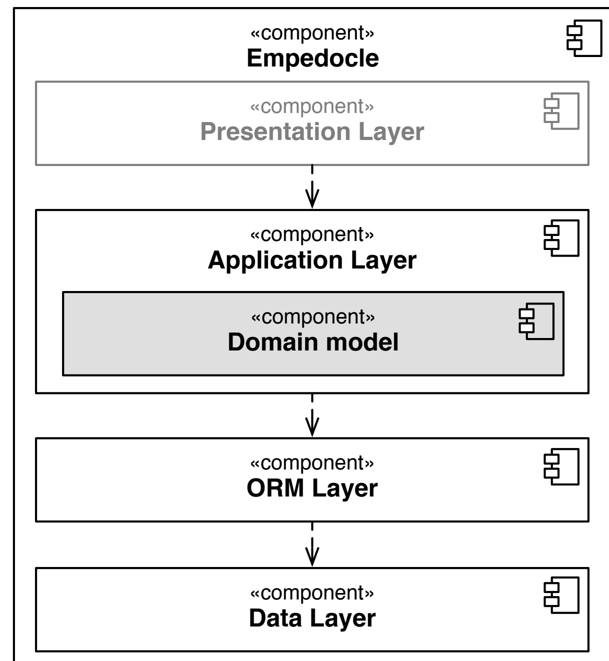

Figure 3: The software architecture of the *Empedocle* EHR system. The *Domain model* component implements the domain logic through the meta-level modeling approach as described in Fig. 1.

Fig. 3 shows the software architecture of the *Empedocle* EHR system, as currently deployed at the Careggi hospital, which follows the usual scheme of a 3-tier system: the *Data layer* provides mechanisms for storing and retrieving data from a relational database; the *Object-Relational Mapping (ORM) Layer*, implemented by *Hibernate*, reconciles the object/relational paradigm mismatch between objects and relational data [22]; the *Application layer* implements the domain model of Fig. 1 and other services; finally, for the sake of completeness, the *Presentation layer* implements interfaces and logic for the interaction with users, and includes a *Viewer Engine* for automated generation of EHR content GUIs.

The high degree of changeability and adaptability provided by the *Empedocle* architecture allows that user tasks and responsibilities in the context-of-use [23] be partitioned according to the summary use case diagram of Fig. 4.

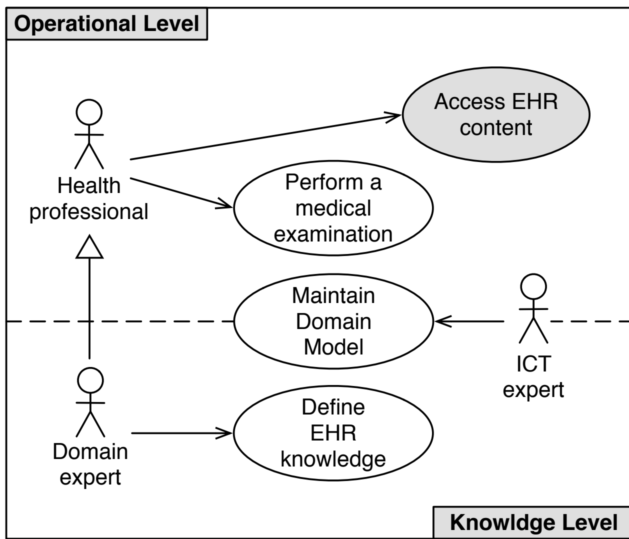

Figure 4: A typical outpatient scenario, specifying the major actors involved in the care process and their interaction with an EHR system. The highlighted use case represents a major scenario of interaction: the health professional actor accesses patient's EHR content in order to review past medical examinations and read collected clinical information.

Health professionals (e.g. general practitioner, medical specialist, registered nurse) take part to the care process at the operational level in different ways, in accordance with personal skills and specializations, including: *i)* the complete review of the patient's EHR content (e.g. clinical history, allergies, active problems, test results); *ii)* the acquisition of clinical data through a medical examination; *iii)* the formulation of the correct diagnosis; and *iv)* the development of a specific treatment plan.

Medical concepts related to clinical data collected into the EHR system are identified and steadily maintained at knowledge level by one or more *domain experts*, who are health professionals with specific domain expertise as well as aware about governmental and hospital directives, and about factors depending on specialization of activities and scientific aims.

Finally, the *ICT expert* plays a lead role in bridging medical and informatics domains, in cases where technical skills are required for supporting health professionals through the implementation of additional system requirements that demand structural changes in the domain model, at the operational as well as the knowledge levels.

We do not report here on the characteristics of other complementary roles which are involved in the organization and enactment of the clinical process (e.g. from health direction and administrative support), but that are not directly concerned with the topic addressed in this paper.

3. MODELING REFLECTION OVER A NOSQL PERSISTENCE LAYER

In the common practice of software development, the persistence layer deals with retrieving data from and storing data to a relational data store, usually through the interposition of an ORM layer. In this kind of approach, the persistence model is largely determined by the object-oriented design of the domain logic.

By contrast, when persistence relies on a NoSQL solution, design gives space to alternative choices in the definition of the storage data model, which is, to a large extent, independent from the structure of object types. In fact, the absence of a fixed schema provides multiple options concerning the definition of the database structure, facilitating the representation of heterogeneous data characterized by high variability over time. The overall design results more flexible, but inevitably more complex and harder to understand for software developers used to deal with traditional relational databases [24]; it also requires to take into account some specific aspects so as to realize data migration in the most opportune way [13].

In the rest of this Section, we describe two new data models as implemented using different NoSQL technologies, *Neo4j* [16], and *MongoDB* [17]. The choice of these two technologies was made so as to experiment with their data structure and promising performance improvement [25, 26], and to compare graph- and document-oriented NoSQL solutions applied to the case of a reflection software architecture that combines the *Observations & Measurements* and *Composite* patterns, as described in Sect. 2. Finally, the validity of the proposed models is proved, in terms of integrity of persisted data and equivalence of data representations.

3.1 A model for Neo4j

Neo4j [16] relies on a *graph-oriented* structure, which can natively represent the domain logic of a reflection architecture, whose data structures are direct acyclic graphs and trees [27]. As a schemaless database, the data model in Neo4j is inherently defined by the *nodes* and *relationships* persisted in the database. Every node and relationship can also be characterized by an arbitrary number of *properties*.

From version 2.0, Neo4j developers tweaked its schemaless nature by introducing *labels* and *indexes*, two concepts that help modelling data in a more organized way, without losing the database original adaptability. Specifically, labels can be used to group together nodes, and each node can optionally be labeled with one or more text descriptions, and indexed

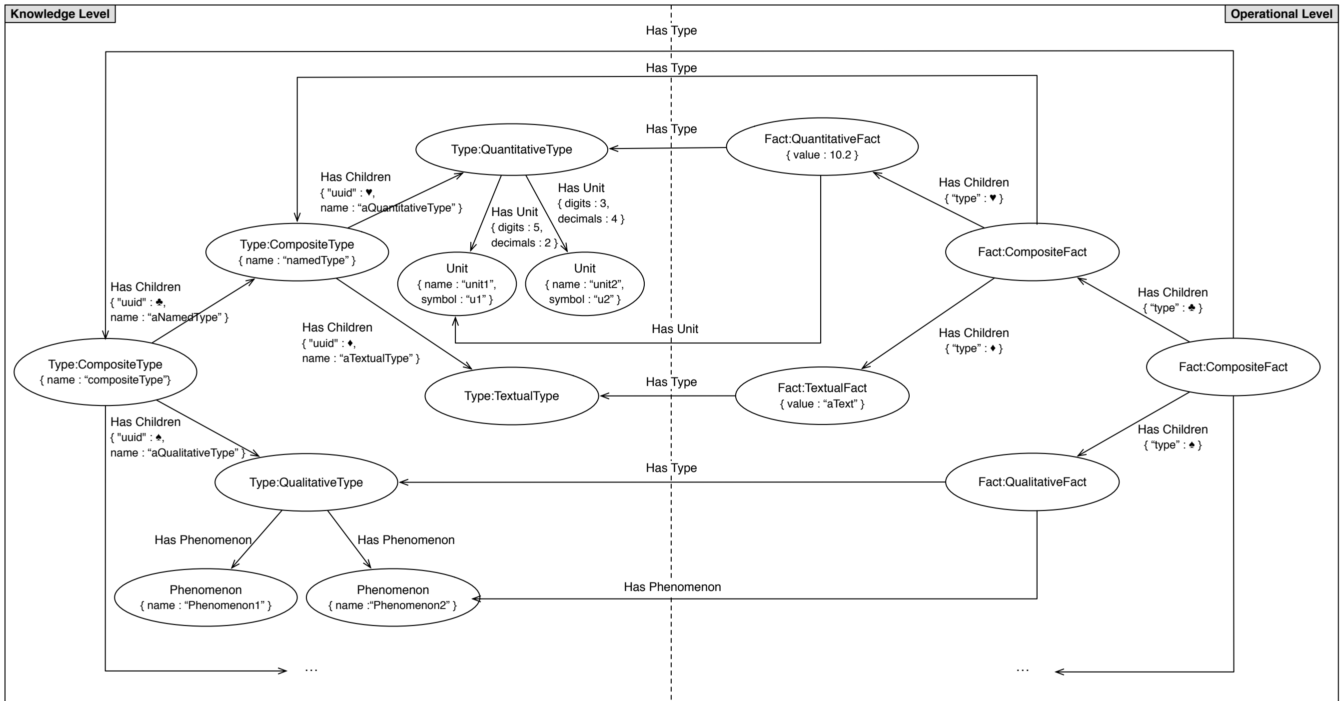

Figure 5: The representation of an instance of the domain model described in Sect. 2 on the graph model of Neo4j database. Oval shapes represent nodes, and arcs between nodes represent relationships, with labels written in bold, and properties reported between braces. For example, a *Type:CompositeType* node is characterized by multiple labels: the first one specifies that it is an instance of `FactType` class, the second one identifies its role in the hierarchy. The *Has Children* relationship identifies children nodes. For reasons of readability, *uuid* property values have been replaced with symbols.

to improve query expressiveness and flexibility. Moreover, indexes can be defined on properties of labelled nodes, to improve performance during query operations, similarly to the relational case. Both labels and indexes are optional.

In our concrete case, modeling the domain logic in Neo4j comes down to: *i)* identifying the node structure that forms the model; *ii)* defining the relationships between nodes; *iii)* defining the properties that characterize nodes and relationships, and *iv)* labeling with the appropriate qualifiers.

Specifically, as depicted in the schema of Fig. 5, each class that is an entity in the original model has been represented as a node in the target model (i.e. `FactType` and `Fact` hierarchy classes, and `Phenomenon` and `Unit` classes). The resulting nodes have been labeled with a correspondent qualifier and, in addition to that, nodes that are part of the `FactType` and `Fact` hierarchies contain an extra label to identify their role in the class hierarchy (e.g. *Fact:QualitativeFact* qualifies a `QualitativeFact` inside a `Fact` hierarchy).

As it can be observed in the schema, the *name* property is used for identifying, at the knowledge level, a *named* `FactType`. The value *property* is used to record, at the operational level, the value assumed by a `TextualFact` or a `QuantitativeFact` node: a string of text in the first case, and a double precision number in the latter case. In this model, there is basically no difference between *named* and *anonymous* `FactTypes`: both are modeled using a node, and the only distinction between them is the presence of the *name* property.

Another characteristic of the graph model in Fig. 5 is the capability of modeling `TypeLink` and `FactLink` classes using relationships. These two classes were introduced in the original model to represent the parent-child relationship between `FactType` or `Fact` classes. For this reason, they can be naturally represented as a relationship in a graph-oriented model. In addition, since Neo4j represents relationships as directed arcs that can be traversed in both directions, this allows to simplify the model introducing a single relationship, called *Has Children*, for modeling `TypeLink` and `FactLink` classes, without any impact on query capabilities. Note that Neo4j allows to put a relationship only between two nodes, and this precludes the possibility to use a relationship to represent the reference between `TypeLink` and `FactLink`, as in the original model. Properties have been used to solve this problem, as follows: *i)* the *uuid* property of each `TypeLink` is used for storing an identifier value; *ii)* the same value is copied into the *type* property of the related `FactLink`. Properties have been used to solve this problem without transforming these two classes from relationships to nodes. Finally, the *Has Type* relationship is used to link together `Fact` and `FactType` nodes.

The self-explanatory *Has Unit* and *Has Phenomenon* relationships are ambivalent across the knowledge and the operational level, and are used to connect a `QuantitativeType` or `QualitativeType` node with a set of possible `Unit` or `Phenomenon` nodes, and the corresponding `QuantitativeFact` or `QualitativeFact` node with the selected `Unit` or `Phenomenon` node.

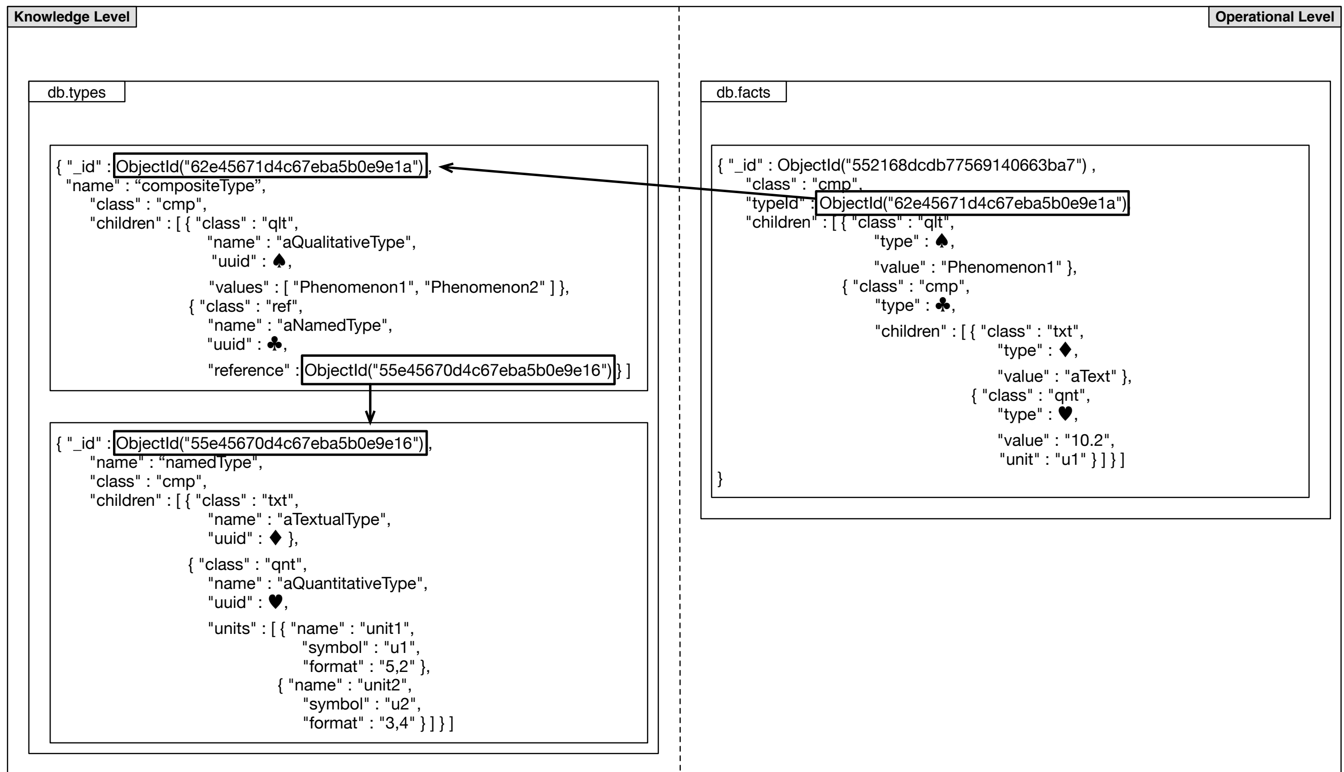

Figure 6: The representation of an instance of the domain model described in Sect. 2 on the document model of MongoDB database. The two sides of the figure show the collections used to persist `FactType` and `Facts` instances, named *db.types* and *db.facts*, respectively. At knowledge level, two *named* types have been persisted, with names *compositeType* and *namedType*. The first type includes the second one, as noted by the use of the *ObjectId* reference, and both of them include *anonymous* types as sub-documents. For reasons of readability, *uuid* property values have been replaced with symbols.

3.2 A model for MongoDB

MongoDB [17] data model is based upon a *document-oriented* structure. A document is a collection of attribute–value pairs, with values that can be basic types, array of values or nested sub-documents. Documents with similar characteristics are grouped together and stored in *collections*. Relation between documents can be represented using *references*, that produce a normalized data model, or by *embedding* related data in documents, producing denormalized models. In particular, the use of denormalization techniques [28] is promoted by document-oriented NoSQL solutions for discouraging the usage of JOIN queries, and solving typical performance issues that affect relational databases, preserving data consistency and completeness [29].

The schema of Fig. 6 illustrates the document-oriented model used in our concrete case, representing data in accordance with the domain model of Fig. 1. Usually, modeling an object-oriented domain logic using a document-based data model can be achieved in a direct way, but, in the case of study of a reflection architecture, this simplicity is weakened from the indirect structure of the model. The proposed solution attains a good balance, *mixing* together documents embedding approaches with references techniques [30] for obtaining a flexible data representation without performance degradation. In particular, the `FactType` hierarchy comprises a neat example of *mixed modelation*. In fact, while *named* `FactType` instances are persisted as documents, and are referenced by other documents using their *ObjectId*, *anonymous* `FactType` instances are persisted as embedded documents inside the named `FactType` document in which they are defined.

To efficiently recognize the subtyping-class of an instance in the `FactType` or `Fact` hierarchy, every persisted document has a property called *class* that can assume the following values: *i) txt*, for referring to a `TextualType` or `TextualFact` instance; *ii) qlt*, for referring to a `QualitativeType` or `QualitativeFact` instance; *iii) qnt*, for referring to a `QuantitativeType` or `QuantitativeFact` instance; and, *iv) cmp*, for referring to a `CompositeType` or `CompositeFact` instance. In so doing, it is sufficient to check the *class* property value of a document to recognize its nature, avoiding to pre-emptively explore its properties. In the case of *named* `FactTypes`, the *class* property is valued with the string value *ref*, and an additional property called *reference* contains the *ObjectId* of the *named* `FactType`.

This different behaviour in `FactType` persistence drops the need to persist the `TypeLink` class as a separate entity. For this reason, `TypeLink` and `FactType` classes are modeled in MongoDB as a single entity, and the *name* property of embedded documents inside `CompositeType` instances corresponds to the `TypeLink` *name* property of the original model.

Note that since the embedded documents are always *anonymous*, the `FactType` *name* property is specified only for the root document of a `FactType` instance.

The `Fact` hierarchy does not have the same need for reusability and referencing that characterize `FactTypes`. For this reason, `Fact` instances can always be represented as a single document, in which `Fact` children are embedded as sub-documents. In so doing, the number of queries for data retrieval is considerably limited.

`Fact` and related `FactType` instances are linked together with different strategies, based on the nature of the `Fact`. In the case of a `Fact` root document, the *typeId* property is used to store the *ObjectId* of the referenced `FactType` instance. Otherwise, when dealing with sub-documents of the `Fact` root, the *type* property is used to refer to the *uuid* value of the corresponding `FactType`. Consequently, for completely retrieving a `Fact` and its `FactType`, it is necessary to: *i)* query for the `Fact`; then, *ii)* query for the corresponding `FactType` using the *ObjectId* referenced by the `Fact` root; *iii)* link together the retrieved `Fact` and `FactType` instances using the *type* property.

For the sake of completeness, `Phenomenon` entities are modeled as embedded documents inside `QualitativeFact` and `QualitativeType` documents with the intent of minimizing the number of retrieval query in reading operations. In the same manner, `Unit` entities are modeled inside `QuantitativeFact` and `QuantitativeType` documents.

3.3 Information equivalence across data models

A comparison of the performance among different data storage implementations (i.e. from relational to a graph- or document-oriented model) requires that they are in some sense equivalent. Since data can be modeled in various ways through the use of different data structures offering the same *information capacity*, a notion of model equivalence, or hierarchy of equivalences [31], is required to be defined. In a general sense, two data structures can be considered equivalent in terms of information-capacities if they can be associated to the same number of states, such that each state of a data structure can be mapped to a *database state* of the other structure, preserving any relationship attribute value.

For the purpose of our experimentation, it is not necessary to prove the *complete* equivalence between two representations, but it is sufficient to prove the *query* equivalence of two models [32], i.e. the possibility to extract the same information from both models through query operations. Specifically, the equivalence problem consists in casting information data into structures (i.e. graphs or tree) of the same type. Comparing and matching graphs is a well-known NP-complete problem [33], and different approaches have been proposed to determine the distance between two graphs using specific heuristic [34, 35]. In our case, proving the equivalence of *Neo4j* and *MongoDB* data models with respect to the actual relational model means showing that they have the same representativeness of information. This means that the equivalence problem will be focused on showing that two data structures are exactly identical in the information they carry, rather than identifying similarities and differences between data models. Furthermore, it is not necessary to verify the *query dominance* for the new data model, but simply proving that it is possible to query the same `Examination` and `ExaminationType` structure across different representations.

In a practical manner, we consider equivalent two data representations of the same domain logic using different persistence models when the carried information can be serialized into an equivalent string of information. In so doing, given two different persistence models, named A and B, A and B are equivalent if it is possible to generate the same string serialization for each given `Examination` and `ExaminationType` instance represented in A and B. Consequently, if A is a valid model, and A and B are equivalent, than B is also valid. Note that we assume that the actual relational model is a valid reference model, from which we want to prove the validity of the converted NoSQL models.

We have started by choosing a dataset with an arbitrary number of clinical information data persisted in the relational model. Then, we have retrieved all the `Examinations` and `ExaminationTypes` instances contained in the dataset, and we have serialized the information data in a string representation. Finally, for testing the equivalence, we have converted information data from the relational model to the target NoSQL model, serializing again the information data, and comparing the resulting string with the string obtained from the relational model at the previous step. The validation process is considered successful, if we are able to obtain an equivalence between the reference relational model and the target NoSQL model for every string of information.

Fig. 7 illustrates an example of the string produced during the serialization process applied to the information data as so represented using the models depicted in Figs. 5 and 6. The structure of the serialization is deliberately similar to a JSON document, due to its simple and readable syntax. This string serialization can be also used to verify which are the essential properties that a model must implement to be valid.

Note that the completeness of the new representation is also granted by the structures of target models. In fact, the conversion from *MySQL* to *Neo4j* model is the most natural way since it allows to maintain nodes and relationships according to the structure of the original tree or graph. Moreover, the MongoDB document representation is modeled in a way that can be considered an inverse operation of vertical decomposition during normalization process, as discussed in [32] and [36].

```
"compositeType" : {
  "aNamedType" : {
    "aQuantitativeType" : "10.2 u1"
    "aTextualType" : "aText"
  }
  "aQualitativeType" : "Phenomenon2"
}
```

Figure 7: An example of serialization of a clinical `Examination`. The pattern used to serialize the information is as follows: *type.name : fact.value*. `CompositeFact` values are described by the list of values assumed by children `Facts` defined between braces.

4. EXPERIMENTATION AND RESULTS

An experimentation was carried out to evaluate the performance of the three different implementations based on MySQL+Hibernate, Neo4j, and MongoDB, and their sensitivity to the characteristics of the dataset.

The evaluation was focused on the *Access EHR content* use case (see Fig. 4), in which a health professional actor access past medical examinations related to a specific patient in order to review collected clinical information. This use case has turned out to have the most relevant impact on the perceived performance in the context-of-use and, at the same time, constitutes a major scenario of interaction in EHR systems.

4.1 Methodology of experimentation

We can expect that the response time of different storage schemes be dependent on the complexity of the collection of domain logic objects that are read-from or written-to the persistence layer. Due to the pattern-based architecture of classes in the domain logic, objects are organized in an almost tree-like structure, and their complexity can thus be characterized in terms of number of nodes and depth of the tree in the examination structure.

For this reason, we experimented with two different kind of datasets: *i)* a *real* dataset of clinical examinations acquired in the *Empedocle* EHR system for which we provide a description of the statistics about the number of nodes and the depth of the tree structure; *ii)* a *synthetic* dataset for which we can control the statistics so as to stress the indexes of complexity.

The *real* dataset consists of about 13 000 examinations [1] that belong to the same medical speciality and thus share the same structure. Table 1 summarizes the complexity of the examination structure, i.e. the number of `FactTypes` included in each examination, which is the number of meta-objects in the knowledge level. The structure of the examination includes 243+110+99 fields, which are organized in a graph whose depth (intended as the maximum distance from the root node) is equal to 8, and which includes 144 `FactTypes` that act as composition nodes.

Depth	8	
Number of nodes	596	
	CompositeType	144
	QualitativeType	243
	QuantitativeType	110
	TextuaType	99

Table 1: Characteristics of the considered examination structure in the dataset, with additional details about the distribution of type nodes contained in the structure. Of the 596 nodes that form the examination type, 452 nodes are leaf nodes, which actually contain a value.

Note that, at the operational level, the complexity of the tree structure depends on the course of each specific examination, and its statistic is resumed in Figs. 8 and 9. Fig. 8 reports the distribution of examinations per number

[1]The real dataset was conveniently anonymized by omitting patients' personal information, and by obfuscating textual observations recorded during each clinical session.

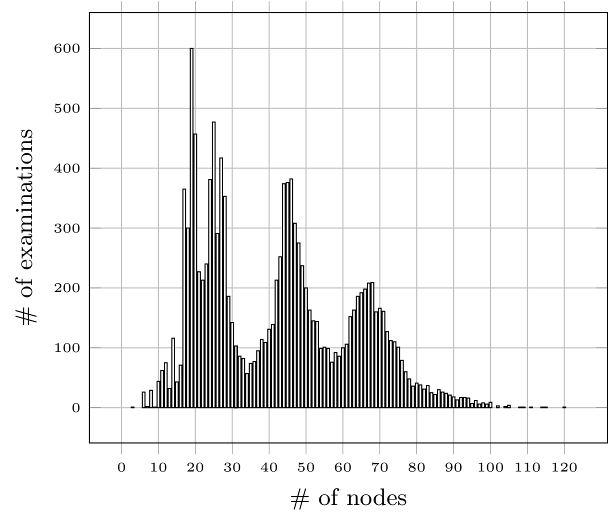

Figure 8: The histogram describes the distribution of examinations as the number of nodes varies. Note that about 35% of the examinations in the dataset are in the neighbourhood of 23±6, with peaks in 19, 20, 25 and 27. This shows clearly how, usually, only a small part of the examination structure, comprising 596 nodes, is actually filled out by health professionals. Only about 9% of the examinations in the dataset have more than 70 nodes filled out.

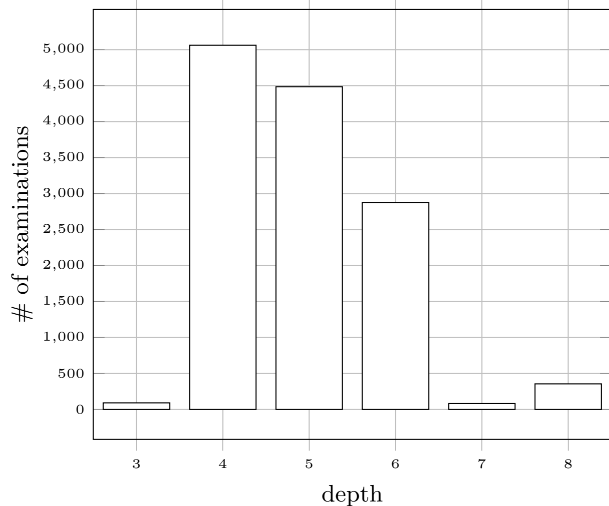

Figure 9: The histogram describes the distribution of examinations as the depth increases. Note that 96% of the examinations in the dataset have depth comprises between 4 and 6.

of nodes. Fig. 9 characterizes the distribution of examinations per depth of the tree structure. From these statistics, it is possible to note that the size of the tree-like structure (composed by `Facts`) is always much lower than the size of the corresponding graph structure (composed by `FactTypes`), which depends on the fact that, during a standard clinical session, not all the observations allowed by the examination structure (≈ 600) are actually recorded.

The *synthetic* dataset contains generated examinations with a full binary tree structure, with depth ranging from 2 to 8. For each depth, a fixed number of 100 examinations has been generated. Being a full binary tree, the number of nodes n in each of the trees of depth d is given by:

$$n = 2^{d+1} - 1,$$

ranging from 3 to 511 nodes. The synthetic dataset does not correspond to a real situation in the present context-of-use of our EHR system, but it can become a possible scenario in the evolution of the use of the *Empedocle* EHR system, and, for this reason, represents a relevant part of the motivation for this performance engineering investigation. In the more general perspective of a reflection architecture, this corresponds to the case where different courses can be described on a structure with different degrees of completeness.

The evaluation has been carried out with reference to a major scenario of interaction: a health professional accesses a patient's EHR content in order to review past medical examinations and read collected clinical information. To do that, each examination in the dataset has first been retrieved and, then, a read-only operation has been performed in order to simulate the real interaction of users with the EHR system through the interfaces exposed by the *Presentation Layer*.

As a metric of performance, we evaluated the total time required to complete the selected scenario, from data retrieving to data serialization, for all compared models. Note that the examination retrieval also implies the retrieval of the associated type structure. In the process of measuring the total time, we do not distinguish time passed by the various phases of data retrieval and process: specifically, we measure the time to complete the whole use case, that comprises database retrieval operations, interactions with Java database APIs or with the ORM persistence layer present only in the relational case.

Experimental results not reported here indicate that the ORM layer, implemented by *Hibernate* in the current application stack, does not significantly impact the overall performance, since it is optimized for the underlying database technology [37, 38]. No comparison has been carried out regarding storage space requirements for the considered technologies, not representing a critical aspect for the EHR system in the case under consideration.

The experiments were conducted on a computer with the following characteristics: Debian 3.2.60 operating system, with 2 x Intel Xeon E5640 @ 2.66 GHz 64-Bit CPU, and 32 GB of RAM Memory.

4.2 Results

Table 2 reports the results of the experimentation on the real dataset, showing the mean value (μ), measured in ms, and the coefficient of variation (CV) of the time spent to complete the read-only operation for a single examination in the three implementations under test. These statistical indexes were evaluated by repeating the task for 100 times on all 12 953 examinations in the dataset.

In comparison with the MySQL + Hibernate implementation, Neo4j reduces the retrieval time by approximately 1.5 times, and MongoDB reduces it by more than 33 times. Performance with relational database such as MySQL are deeply linked to the number of JOIN in the executed queries. For this reason, in the considered model, the retrieval of specific classes of `Facts` has a different impact on the complexity of the query. In particular, since `CompositeFacts` represent hierarchical structures in the `Facts` tree, querying operations result in a higher number of JOINs, which produces a significant impact on performance, documented by [5] as "*N+1 queries*" data access anti-pattern or "*Circuitous Treasure Hunt*" problem. In a similar way, `QuantitativeFacts` and `QualitativeFacts` also produce more complex queries, since an additional JOIN operation is required to retrieve related `Phenomena` and `Units`.

	μ (ms)	CV	*min* (ms)	*max* (ms)
MySQL + Hibernate	76.06	0.031	70.79	81.94
Neo4j	51.29	0.0024	50.94	51.57
MongoDB	2.27	0.064	1.82	2.57

Table 2: Comparison between MySQL+Hibernate, Neo4j, and MongoDB, evaluated using the *real* dataset comprising 12953 examinations. Table reports the mean value (μ) and the coefficient of variation (CV) for the execution of a single examination, as well as the minimum (*min*) and maximum (*max*) execution time registered during the 100 iterations for a specific technology.

Table 3 shows the results of experimentation on the *synthetic* dataset. We report the mean value (μ), measured in ms, and the coefficient of variation (CV) of the time spent to complete the read-only operation for a single examination in the three implementations under test, evaluated by repeating the task for 100 times on all 100 examinations in the dataset. Results indicate that MongoDB attains by far a better performance and slower sensitivity to the examination depth. It should also be noted that the MySQL + Hibernate implementation performs better than Neo4j for examination with depth lower than 7.

Depth	**MySQL + Hibernate**		**Neo4j**		**MongoDB**	
	μ (ms)	CV	μ (ms)	CV	μ (ms)	CV
2	6.93	0.12	18.76	0.12	1.07	0.05
3	9.54	0.11	19.41	0.09	1.2	0.06
4	12.6	0.1	21.57	0.09	1.38	0.07
5	18.87	0.09	26.04	0.08	1.64	0.07
6	28.17	0.09	33.94	0.05	2.2	0.07
7	48.18	0.08	44.03	0.05	3.05	0.08
8	121.29	0.04	72.93	0.05	4.88	0.07

Table 3: Comparison between MySQL+Hibernate, Neo4j, and MongoDB, evaluated using the *synthetic* dataset comprising 100 examinations with increasing depth. Table reports the mean value (μ) and the coefficient of variation (CV) for the execution of a single examination. Results in the table show that the MySQL + Hibernate implementation performs better than Neo4j for examination with depth lower than 7, while MongoDB attains by far a better performance and slower sensitivity to the examination depth.

5. CONCLUSIONS

In this paper we described a consolidated software architecture, pattern-based, which persistence data layer was originally based on the MySQL relational database and JPA. Using a model-driven approach we described new data persistence models based on promising NoSQL technologies, such as *Neo4j* [16] and *MongoDB* [17]. These models have been engineered to balance in the best way elements such as: ease of conversion, embedding and references, granting data integrity and equivalence in the information representation with the relational model.

We presented experimental results on performance gain achieved through the use of such databases. The comparison is based on the study of the real world scenario of our EHR system, called *Empedocle*, based on the *Observations & Measurements* [4] and *Composite* [20] patterns, where the main requirement is the structure flexibility. Since the considered NoSQL databases do not have a fixed schema, non-functional requirements of changeability and adaptability can be easily achieved. In addition, they constitute a good solution for big clusters of data which structure is subject to change over time.

Performance results obtained during experimentations in *real* and *synthetic* datasets indicate a clear gain in performance through the use of MongoDB database, and more generally, a better scalability of NoSQL solutions when the depth of the examination structures grows, due to the increased number of JOINs and reference operations affecting the MySQL solution. Moreover, both tested NoSQL technologies offer advantages in terms of flexibility in the data model, scalability and reliability.

Results also indicate a counter-intuitive conclusion: the graph-oriented data model of Neo4j allows a more natural and direct data conversion, which also permits a simpler implementation; however, the document-oriented data model of MongoDB produces by far better performance results. Specifically Neo4j, which modeling is more natural in our software architecture context, presents a performance increase of 1.5 times compared to MySQL + Hibernate. On the other hand, MongoDB, which required a bigger engineering investment to convert our data model balancing between redundancy, adaptability and performance, presents a gain of almost 33 times compared to MySQL + Hibernate.

The present investigation is completely open to explore the performances of NoSQL databases in other use cases, not only limited to read-only operations, but also extended to write and update scenarios, whose impact on the application is less relevant but nonetheless interesting to have a full comparison between the various models [39].

6. REFERENCES

[1] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC, 2001.

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture: A System of Patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[3] Joseph W Yoder, Federico Balaguer, and Ralph Johnson. Architecture and design of adaptive object-models. *ACM Sigplan Notices*, 36(12):50–60, 2001.

[4] Martin Fowler. *Analysis patterns: reusable object models.* Addison-Wesley Professional, 1997.

[5] Connie U Smith and Lloyd G Williams. Software performance antipatterns. In *Workshop on Software and Performance*, pages 127–136, 2000.

[6] Davide Arcelli, Vittorio Cortellessa, and Catia Trubiani. Antipattern-based model refactoring for software performance improvement. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, pages 33–42. ACM, 2012.

[7] Laney Douglas. 3d data management: Controlling data volume, velocity and variety. *Gartner. Retrieved*, 6, 2001.

[8] Scott Ambler. *Agile database techniques: effective strategies for the agile software developer.* John Wiley & Sons, Inc., New York, NY, USA, 2003.

[9] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. *High performance MySQL: optimization, backups, and replication.* " O'Reilly Media, Inc.", 2012.

[10] Saleem N Bhatti, Zahid H Abro, and Farzana R Abro. Performance evaluation of java based object relational mapping tool. *Mehran University Research Journal of Engineering and Technology*, 32(2):159–166, 2013.

[11] Robert C Martin. The dependency inversion principle. *C++ Report*, 8(6):61–66, 1996.

[12] Michael Stonebraker. SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4):10–11, 2010.

[13] Aaron Schram and Kenneth M Anderson. MySQL to NoSQL: data modeling challenges in supporting scalability. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 191–202. ACM, 2012.

[14] Bogdan G Tudorica and Cristian Bucur. A comparison between several NoSQL databases with comments and notes. In *Roedunet International Conference (RoEduNet), 2011 10th*, pages 1–5. IEEE, 2011.

[15] João R Lourenço, Bruno Cabral, Paulo Carreiro, Marco Vieira, and Jorge Bernardino. Choosing the right NoSQL database for the job: a quality attribute evaluation. *Journal of Big Data*, 2(1):1–26, 2015.

[16] Neo4j the world's leading graph database. http://neo4j.com/.

[17] MongoDB for GIANT idea. https://www.mongodb.org/.

[18] ISO/TR. *ISO/TR 20514:2005. Health informatics — Electronic health record — Definition, scope and context.* ISO/TR, 2005.

[19] Paul C Tang, Joan S Ash, David W Bates, J Marc Overhage, and Daniel Z Sands. Personal health records: definitions, benefits, and strategies for overcoming barriers to adoption. *Journal of the American Medical Informatics Association*, 13(2):121–126, 2006.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[21] Thomas Beale, Sam Heard, Dipak Kalra, and David Lloyd. OpenEHR architecture overview. *The OpenEHR Foundation*, 2006.

[22] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on*, pages 36–43. IEEE, 2009.

[23] ISO. *ISO 9241. Ergonomics of human-system interaction*. ISO, 2010.

[24] Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, and Riccardo Torlone. How I learned to stop worrying and love NoSQL databases. In *SEBD Italian Symposium on Advanced Database Systems*, 2015.

[25] Florian Holzschuher and René Peinl. Performance of graph query languages: comparison of Cypher, Gremlin and Native Access in Neo4J. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 195–204, New York, NY, USA, 2013. ACM.

[26] Wei Xu, Zhonghua Zhou, Hong Zhou, Wu Zhang, and Jiang Xie. MongoDB improves big data analysis performance on Electric Health Record system. In *Life System Modeling and Simulation*, pages 350–357. Springer, 2014.

[27] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference*, page 42. ACM, 2010.

[28] Anuradha Kanade, Aarthi Gopal, and Shantanu Kanade. A study of normalization and embedding in MongoDB. In *Advance Computing Conference (IACC), 2014 IEEE International*, pages 416–421. IEEE, 2014.

[29] Gansen Zhao, Qiaoying Lin, Libo Li, and Zijing Li. Schema conversion model of SQL database to NoSQL. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on*, pages 355–362. IEEE, 2014.

[30] Ilya Katsov. NoSQL data modeling techniques. *Highly Scalable Blog*, 2012.

[31] Richard Hull. Relative information capacity of simple relational database schemata. *SIAM Journal on Computing*, 15(3):856–886, 1986.

[32] Paolo Atzeni, Giorgio Ausiello, Carlo Batini, and Marina Moscarini. Inclusion and equivalence between relational database schemata. *Theoretical Computer Science*, 19(3):267–285, 1982.

[33] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[34] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, June 1996.

[35] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance evaluation of the VF graph matching algorithm. In *Image Analysis and Processing, 1999. Proceedings. International Conference on*, pages 1172–1177. IEEE, 1999.

[36] Catriel Beeri, Philip A Bernstein, and Nathan Goodman. A sophisticate's introduction to database normalization theory. In *Proceedings of the fourth international conference on Very Large Data Bases-Volume 4*, pages 113–124. VLDB Endowment, 1978.

[37] Elizabeth J O'Neil. Object/relational mapping 2008: Hibernate and the Entity Data Model (EDM). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356. ACM, 2008.

[38] Qinglin Wu, Yanzhong Hu, and Yan Wang. Research on data persistence layer based on hibernate framework. In *Intelligent Systems and Applications (ISA), 2010 2nd International Workshop on*, pages 1–4. IEEE, 2010.

[39] Zachary Parker, Scott Poe, and Susan V Vrbsky. Comparing NoSQL MongoDB to an SQL db. In *Proceedings of the 51st ACM Southeast Conference*, page 5. ACM, 2013.