Project Number 611125

D5.1 – Data Persistence Technology Evaluation Report

Version 1.0
1 June 2014
Final

Public Distribution

University of York

Project Partners: **ARMINES**, **Autonomous University of Madrid**, **BME**, **IKERLAN**, **Soft-Maint**, **SOFTEAM**, **The Open Group**, **UNINOVA**, **University of York**

Project Partner Contact Information

ARMINES Massimo Tisi Rue Alfred Kastler 4 44070 Nantes Cedex, France Tel: +33 2 51 85 82 09 E-mail: massimo.tisi@mines-nantes.fr	**Autonomous University of Madrid** Juan de Lara Calle Einstein 3 28049 Madrid, Spain Tel: +34 91 497 22 77 E-mail: juan.delara@uam.es
BME Daniel Varro Magyar Tudosok korutja 2 1117 Budapest, Hungary Tel: +36 146 33598 E-mail: varro@mit.bme.hu	**IKERLAN** Salvador Trujillo Paseo J.M. Arizmendiarrieta 2 20500 Mondragon, Spain Tel: +34 943 712 400 E-mail: strujillo@ikerlan.es
Soft-Maint Vincent Hanniet Rue du Chateau de L'Eraudiere 4 44300 Nantes, France Tel: +33 149 931 345 E-mail: vhanniet@sodifrance.fr	**SOFTEAM** Alessandra Bagnato Avenue Victor Hugo 21 75016 Paris, France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr
The Open Group Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org	**UNINOVA** Pedro Maló Campus da FCT/UNL, Monte de Caparica 2829-516 Caparica, Portugal Tel: +351 212 947883 E-mail: pmm@uninova.pt
University of York Dimitris Kolovos Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 32516 E-mail: dimitris.kolovos@york.ac.uk	

Contents

Document Control

Version	Status	Date
0.1	Document outline	24 March 2014
0.2	First draft	15 May 2014
0.3	First full draft	22 May 2014
1.0	QA review	1 June 2014

Executive Summary

This deliverable is the outcome of Task 5.1 (Data Persistence Technology Evaluation and Benchmarking) and presents the process and results of benchmarking candidate data stores that can underpin the heterogeneous model indexing framework that will be developed in Tasks 5.2 and 5.3 of the project.

The experiments were carried out in a two-phased approach; in the first phase, a wide range of stores were evaluated using a common abstraction layer (BluePrints), and in the second phase, the most promising stores (two Property Graph Databases and three Triple Stores) were benchmarked using their native APIs.

The obtained results indicate that the Neo4J Property Graph Database delivers the best performance in both persistence and querying when bulk insertion is used. As such, our plan is to use Neo4J as the infrastructure on which we will build the envisioned heterogeneous model indexing framework in Tasks 5.2 and 5.3.

1 Introduction

In recent studies, Model Driven Engineering (MDE) has been shown to increase productivity and significantly enhance important aspects of software engineering development such as consistency, maintainability and traceability. MDE is therefore increasingly applied to larger and more complex systems. However, the current generation of modelling and model management technologies are being stressed to their limits in terms of their capacity to accommodate collaborative development, efficient management and persistence of models larger than a few hundreds of megabytes in size. Thus, recent research has focused on scalability across the MDE technical space to enable MDE to remain relevant and to continue delivering its widely-recognised productivity, quality and maintainability benefits. The MONDO project proposes to look into the scalability issues for MDE. Typically, achieving scalability involves being able to construct large models and domain specific languages in a systematic manner; enabling large teams of modellers to construct and refine large models in a collaborative manner; advancing the state of the art in model querying and transformations tools so that they can cope with large models (of the order millions of model elements); and providing an infrastructure for efficient storage, indexing and retrieval of such models [51]. As part of the MONDO project, this report looks closely into the efficient storage, indexing and retrieval of large models in server based storage.

The most widely adopted format for model persistence is the XML Metadata Interchange (XMI) format. While XMI has been a significant step towards tool interoperability, it is not a particularly efficient model representation format. Being based on XML, XMI provides limited support for lazy or partial loading, features that are essential for managing large models. While there are existing works which attempt to explore alternatives to XMI, the MONDO project proposes systematic solutions for scalable model persistence, including exploring new and more efficient model interchange format; model fragmentation with an indexing system, which may involve persisting models in databases.

The aim of this report is to look into state-of-the-art large-scale data persistence technologies and provide a common framework and methodology for evaluating and benchmarking of such technologies for large-scale model persistence, so that a baseline can be formed for the model indexing and global querying system proposed by the MONDO project. The evaluation and benchmarking considers a broad range of database technologies including relational and NoSQL databases. The technologies will be narrowed down to the most efficient ones. Efficiency is compared in terms of processor time usage and disk space usage. In the first phase, experiments based on a common abstraction layer (BluePrints [74]) consider a substantial number of database technologies, and the framework and methodology can be directly applied to further stores not considered here. In the second phase, the experiments consider a narrow range of databases using their native APIs to evaluate performance without overheads. We evaluate the framework and methodology via a realistic case study that involves persisting large models reverse engineered from open source projects.

2 Background

2.1 Scalable Model Persistence

The most widely adopted format for model persistence is the XML Metadata Interchange (XMI) format. XMI is an Object Management Group (OMG) standard format designed to enhance tool-interoperability. XMI is based on XML, as such, models stored in single XMI files cannot be partially loaded. Thus, loading an XMI-based model (in particular, an EMF model) requires reading the entire document using a SAX parser and converting it into an object graph in the memory that conforms to the corresponding Ecore metamodel. As discussed in Section 2.2, XMI scales poorly for large models in terms of the time needed for parsing and the resources needed for keeping the entire object graph in memory.

As a result, to address the limitation of XMI, the MONDO project envisions a new efficient model representation format that will reduce the size of model files. The details of such format are beyond the scope of the discussion of this report. On the other hand, it is suggested that splitting large models over many cross-referenced physical files (model fragments) makes it easier to manage models in a typical collaborative development environment where artefacts are stored in a central repository. The main advantage of such approach is that it works well with existing modelling tools and with existing types of version control repositories. However, using this approach with current state-of-the-art technologies makes it impossible to compute queries of global nature such as "find all classes that are sub-classes of X" without going through all of the model fragments. Thus, as the number of model fragments grows, this approach becomes increasingly inefficient. To address this limitation, MONDO envisions a model indexing framework that can monitor the contents of remote version control repositories and index the models they contain in scalable database that will enable efficient computation of global queries. To efficiently support large numbers of model fragments, the persistence mechanism that will be used to underpin the model indexing framework needs to be highly scalable. In this report, persistence technologies that are potential candidates for this purpose, relational and NoSQL databases will be investigated. NoSQL has several categories of store, as discussed in Section 2.3.1, broadly including stores which move away from the traditional relational model.

2.2 Persisting Models in Relational Databases

Several approaches have been proposed for persisting models in relational databases. Examples of such approaches include the Connected Data Object (CDO) [22] project and Teneo-Hibernate [31]. In this category of approach, an EMF Ecore metamodel is used to derive a relational schema as well as an object-oriented API that hides the underlying database and enables developers to interact with models that conform to the Ecore metamodel at a high level of abstraction. This approach reduces the overhead of loading the entire model into memory before any model management operations can be performed. By storing the model into the database, partial and on demand loading can be supported. However, due to the nature of relational databases, such approaches, while out-performing XMI, are still significantly inefficient. Due to the highly inter-connected nature of most models, performing complex queries typically requires multiple expensive table joins to be executed and therefore do not scale well for large models. Teneo-Hibernate limits the evolution of data stored as the schema is

typically fixed and domain specific. Even though Teneo-Hibernate attempts to minimize the number of tables generated, the database consists of a complex schema and sparsely populated tables, which results in increased insertion and query time.

In a case study measuring the performance of relational databases for inserting and querying the GRABATS 2009 models [9], Teneo/Hibernate did not scale for EMF models larger than 283MB, and CDO with MySQL server did not scale for models larger than 626MB. The results of the case study suggests that relational databases may not be suitable for the purpose of persisting models. However, the BluePrints PostgreSQL [63] driver has been selected as the representative for relational databases for the evaluation and benchmarking reported in this document.

2.3 Persisting Models in NoSQL Databases

To overcome the limitations of relational databases for scalable model persistence, recent work has proposed using NoSQL databases instead [9, 34]. NoSQL (Not Only SQL or Not Relational) storage systems rose to prevalence when the traditional Relational Database (RDB) model for data storage could not support the needs of on-line applications. Relational databases are not designed to scale to the amount of data, number of transactions, distribution and number of users that contemporary on-line applications attract. The NoSQL movement itself has become popular due to large widely known and successful companies creating database storage implementations for their services, all of which do not follow the relational model. Such companies include Amazon (Dynamo database), Google (Bigtable database) and Facebook (Cassandra database). Increasingly, developers turn to NoSQL rather than using the rigid structure and features of traditional RDB systems. Such databases aim to also support millions of users, with large storage bases (petabyte-scale) whilst responding to large volumes of requests (millions per second). To achieve these goals, trade-offs are made in line with the CAP theorem [1], commonly sacrificing short-term consistency for the sake of partition-tolerance. Applications using relational data storage may benefit from a NoSQL approach, however, application of NoSQL requires careful consideration of the features and trade-offs of each technology. Currently there are many types of NoSQL databases, each type of NoSQL database is tailored for storing a particular type of data.

2.3.1 NoSQL Database Categories

This section presents a categorisation of NoSQL databases with features common to each category. The broad categories of NoSQL databases considered here are key-value, document, column-oriented, graph and triple stores. A brief introduction is given below with example implementations and general characteristics of each category.

Key-value Databases allow data to be dereferenced based on a key. All data is held in documents retrieved only via a primary key and not any other document value, making this the simplest data model. Although in-memory key-value databases have been available for some time [39], the release of Dynamo by Amazon [29] has led to several distributed and persistent key-value implementations. One clear advantage of key-value databases is the speed at which data can be accessed,

[1] The CAP theorem states that distributed systems can guarantee any two from the three properties of Consistency, Availability or Partition tolerance.

with some implementations claiming ∼100k reads and writes per second [38]. The values are stored without any schema, in a uniform document format such as JSON. As a simple data model, direct, traversable references within documents are not supported – requiring work-arounds at the application level. Examples include Amazon Dynamo [29], Apache Accummulo [41], Project Voldemort [7] and Riak [38].

Document Databases are somewhat similar to key-value databases with two important differences. Multiple document types can be stored in the same system, so for example it is possible to mix JSON and XML documents. Also, indexes are created for all attributes of the stored documents, allowing documents to be dereferenced by potentially any value, sacrificing write speed for more flexibility in data access. However, document databases also have the disadvantage that traversable references are not supported, and as such, workarounds at the application level are required to support inter-document links. Similar to key-value stores, documents may have any schema and JSON is typically used as a document interchange format. Examples include MongoDB [25], CouchDB [18] and ArangoDB [6].

Distributed Tabular Databases (sometimes called column-oriented or 'BigTable' [24] databases) have some of the features of document and key-value databases, yet have others in common with traditional relational databases. Data is referenced by a primary key (used to retrieve a row, or group columns) and columns in a row can be clustered on separate servers to increase retrieval performance. There is no fixed schema, and any two rows may have different columns. Direct references between rows are supported and any column can be used as a key via indexing, separately from the primary key. These databases have emerged based on the Google BigTable [24] design. A common feature in tabular databases is the support for distributed data processing facilities, such as MapReduce [24]. This allows user-written data management programs; results are automatically executed and gathered in parallel across the nodes of the database. Examples of tabular database include Apache Cassandra [53] and Apache HBase [43].

Property Graph Databases have some of the features of each of the above. Firstly, in common with all NoSQL databases, the graph has no fixed schema; data is stored in a graph of vertices and edges. Like document databases, a vertex can store documents, as properties and the documents are not restricted by any schema. Similar to key-value databases, vertices may be dereferenced based on a primary key and further, like tabular databases, secondary indexes allow vertices to be accessed based on any property. Graph databases are distinct from other NoSQL databases as relationships between data (edges) provide a direct reference to vertices, so traversal of edges does not require further queries to the store. This makes graph databases particular suited to analysis of data where links are traversed often, such as network analysis. Examples include Neo4J [78], OrientDB [62] and TitanDB [75].

Triple Store Databases are closely related to property graph databases, which work on the same principle but graphs are flattened into pairs of vertices with an edge between them. Vertices do not store documents; properties of a vertex are stored as additional triples. Triples may optionally conform to a schema set out in a namespace and duplicate triples are not supported. Exaples include Jena [50], Sesame [17] and [1]

2.3.2 Literature on Model Persistence Using NoSQL

There has been some past research on benchmarks and benchmarking for MDE technologies, including evaluations of constraint languages [52], high-performance query languages [11, 35], as well as transformation languages [76]. Many of the studies have considered EMF models. One conclusion that can be drawn is that substantial tuning needs to be carried out (e.g., to EMF or operations on EMF models) to achieve good performance with large EMF models. None of this past benchmarking work has considered performance of non-XMI datastores.

A comparative analysis of a small set of technologies used to store EMF models was performed, including benchmarks of prototypes based on the NoSQL databases Neo4J and OrientDB [8]. The benchmarking suggested that there is a significant benefit in using Graph-based NoSQL databases to store model data. Further analyses [9] considering CDO as well as integration with model management tools [67] showed that use of model management is not detrimental to the performance of NoSQL databases for storing models. This work however, did not present a systematic framework or methodology for comparing individual NoSQL databases.

Various tools attempting to provide a scalable approach to storing EMF models have been proposed, such as Morsa [34, 33], which aims to support scalable model persistence using MongoDB to store EMF models as sets of documents. MongoEMF [56] is an extension of the persistence API of EMF that stores EMF models using MongoDB. It supports basic create, read, update and delete operations as well as queries (including native MongoDB queries via inline JSON). Neo4EMF [59] supports lazy loading, storage, and unloading of large EMF models. It provides a NoSQL database persistence framework based on Neo4J. All of these tools focus on a single back-end driver and attempt to optimise their functionality with that in mind.

Other related work proposes extensible frameworks that persist large models in various back-end databases. The Connected Data Objects repository (CDO[22]) permits the storage and access of models in repositories supported by a range of back-end databases (both relational databases like MySQL[58] and NoSQL/Object databases such as MongoDB and ObjectivityDB[60]). EMF fragments [68] is a persistence layer for distributed data storages; this includes both NoSQL databases (like MongoDB) but also distributed file systems (like Hadoop). EMF fragments map model fragments to URIs, which permits storing models on a wide range of distributed data storages. It supports background fragmentation of models, based on client specifications of fragmentation points. Hawk [10] creates model indexes that are used to efficiently query large sets of XMI models by providing a framework for NoSQL (such as Neo4J) or other databases as back-ends. All of these tools tend to focus on providing an application-specific platform for persisting models.

3 Benchmark Candidate Selection and Model Persistence Requirements

This section identifies existing NoSQL database technologies under different NoSQL database categories. Then this section outlines the functional and non-functional properties of NoSQL databases

taken into account when selecting technologies for benchmarking. The performance of NoSQL databases can vary widely depending on the work load; experiments and benchmarks in this report focus on large-scale modelling, where the ability to persist models into the databases, and model traversal performance are critical. Other requirements are specific to the intended application, so the guidelines here should be adapted depending on the application.

3.1 NoSQL Database Technologies

A number of NoSQL databases have been identified as candidates for the purpose of this report. This section lists these database in the aforementioned categories.

For **Key-value Databases**, the identified candidates are $Aerospike$ [3], $Riak$ [66], $TokyoCabinet$ [21], $Voldemort$ [7], $Redis$ [65], $Hibari$ [45], $FoundationDB$ [40], $DynamoDB$ [29] and $Accumulo$ [2].

For **Document Databases**, the identified candidates are $MongoDB$ [55], $CouchDB$ [27], $ApacheJackrabbit$ [49], $CouchBase$ [27], $EJDB$ [32], $RavenDB$ [64], $SimpleDB$ [70], $FATDB$ [36] and $MarkLogic$ [54].

For **Distributed Tabular Databases**, the identified candidates are $Cassandra$ [53], $HBase$ [44], and $HyperTable$ [47].

For **Property Graph Databases**, the identified candidates are $TitanDB$ [75], $OrientDB$ [62], $Filament$ [37], $BrightstarDB$ [16], $HyperGraphDB$ [46], $WhiteDB$ [79], $ArangoDB$ [6], $Bitsy$ [14], $Neo4J$ [78], $InfoGrid$ [48], $VelocityGraph$ [23], DEX [30], $OQGraph$ [61], $Bigdata$ [28] and $VertexDB$ [77].

For **Triple Store Databases**, the identified candidates are $Jena$ [50], $Sesame$ [17], $AllegroGraph$ [5], $3Store$ [73], $Mulgara$ [57], $RDFGateway$ [42], $StarDog$ [72], $BigOWLIM$ [13], $4Store$ [1] and $BigData$ [12].

It is not possible to evaluate and benchmark all of the candidates identified above. Due to the nature of this research project, a series of selection criteria are also identified so that only feasible candidates which match up with the selection criteria are selected for further evaluation and benchmarking.

3.2 Database Licensing

An important selection criterion and non-functional requirement to consider is the licence of the databases. Legal advice should be sought in non-academic use-cases. The databases considered in this study run as stand-alone servers and are available under some form of free and open-source licence. However, some licences restrict redistribution and creation of derivative products. Permissive licences such as BSD, MIT, APL or EPL allow unlimited redistribution, even when integrated in commercial products. However GPL and AGPL are a copy-left licences and there are licence implications for software that works with GPL licensed code. For example, the source code of Neo4J is available under the AGPL. When a software product or service has a hard dependency on the Neo4J APIs, the source code of that product or service must also be released under the terms of the AGPL. Neo4J also has a commercial licence that allows distribution within commercial products and services. The GPL

permits open-source software to be used if the dependency may be satisfied by another software system, via a generic interface and API. This approach is taken in Section 5.1 by using the BluePrints API- users can select any BluePrints-compatible back-end store.

The store licences used in this report are as follows:

Licence	Example Databases
GPL	4Store
AGPL	BerkleyDB, MongoDB, Neo4J
BSD	Jena, PostgreSQL
APL	Cassandra, Sesame, TitanDB

Table 1: Licences for each store.

3.3 Other Selection Criterias

There are several application specific properties to consider for large scale modelling when using NoSQL databases:

- **Native referencing support**: Since the nature of the usage of databases is persisting models, it requires the support for database to allow referencing between entities due to the interconnected nature of the models. Some NoSQL databases do not support referencing among entities which may require additional effort for model persistence therefore incurring performance impact. The support for referencing among entities is an important criteria, as the nature of the MONDO project, databases with native referencing support are more suitable.
- **CAP [15] trade-off**: CAP refers to a set of basic requirements that describe any distributed system including database systems, they include $Consistency$, $Availability$ and $PartitionTolerance$. However, it is theoretically impossible to meet all 3 requirements. So the trade-offs need to be made among the three. Some NoSQL databases typically allow the users to make such trade-offs to cater with specific needs. However, some databases make the trade-offs for the users and it cannot be configured, therefore restraining flexibility. Due to the nature of the MONDO project and the potential applications of the indexing system on different levels of development process, databases with configurable CAP trade-offs are more suitable.
- **ACID properties**: ACID refers to a set of properties that apply specifically to database transactions, they are $Atomicity$, $Consistency$, $Isolation$ and $Durability$. ACID can be considered as a set of rules that a database can choose to follow that guarantees how it handles transactions and keeps data safe. Some NoSQL databases provide such properties to allow more reliable services. However most NoSQL databases do not implement ACID and vary in how durable they are with stored data. Due to the nature of the MONDO project and the indexing system it proposes, databases with support to ACID properties are desirable. However, since ACID properties sometimes means performance penalty, the requirement to have ACID properties can be relaxed in certain application domains.

- **Disc-resident and in-memory storage**: To improve performance, some NoSQL databases may adopt an in-memory-only model and provide little or no on disc persistence support. This is prevalent in highly available multi-node databases where replicant servers handle failure but may not be appropriate for modelling applications with single-node, client side stores or where memory is limited. Due to the potential application of the indexing system on different levels of the development process, databases with both in-memory and on-disk storage mechanisms are desirable.
- **Application Programming Interface**: The API that a NoSQL database provides is also a significant selection criteria. Since the MONDO research project is based on Java, it is desirable that the NoSQL databases support Java APIs that can be ported directly to the MONDO infrastructure. Some NoSQL databases provides only HTTP-REST APIs which can incur significant performance overhead. Some NoSQL databases are written in Java therefore provides perfect Java APIs with no performance overhead. Since the MONDO project will be mostly based on the Java programming language, databases written in Java with Java native APIs are desirable.
- **Client-server and embedded architecture**: Some NoSQL databases provide an embedded file-based architecture to increase performance as well as the traditional client-server mode. Embedded databases are more suited for single client applications or offline use and are highly performant with batch operations. On the other hand, client-server architecture is also needed as for multiple client access. For the MONDO project, both features are desirable as the indexing system may work on different programming levels.
- **Querying support**: Stores may provide dedicated high-level querying languages, which must be interfaced to support modelling queries, or only simple data access APIs which require implementing modelling query logic directly in the client. For example, in Neo4J, the Cypher query language is a native, high-level query language for graphs, with a similar syntax to SQL. OrientDB provides a BluePrints interface, so graph queries are performed using Java Objects.

3.4 Shortlisted Database Candidates

After applying the selection criteria listed above, the identified Relational and NoSQL database technologies have been shortlisted. The shortlisted databases are as follows:

Category	Shortlisted
Relational Database	PostgreSQL
Key Value	N/A
Document	MongoDB, ArangoDB
Tabular	Cassandra
Graph	TitanDB, OrientDB, ArangoDB, Neo4J
Triple Store	Jena, Sesame

Table 2: Shortlisted NoSQL Database.

For Key Value Databases, none were chosen because they either lack of native reference support or they do not meet the license requirement. NoSQL databases do not strictly follow the categorisation, as it can be seen that ArangoDB crosses over the categories, it is both a Document database and a

Graph database. One thing worth mentioning is that Cassandra does not naturally support referencing, but it can be used as a backend by TitanDB, which treats Cassandra as a graph database with its own implementation.

3.5 NewSQL Database

A new line of research for data persistence is the term referred to as NewSQL technology. It is a class of modern relational database management systems that seek to provide the same scalable performance of NoSQL systems for online transaction processing workloads while still maintaining the ACID properties. Such characteristics make NewSQL an ideal technology for persisting models. However, at the moment there is no NewSQL database that is applicable to the selection criteria for this report.

4 Benchmarking Methodology

This section outlines the methodology used to evaluate the selected databases for storing and querying large-scale models. To avoid benchmarking unsuitable databases, a two-phase approach is used to narrow down the candidates. In the first phase, presented in Section 5, selected candidates are benchmarked using a common framework for NoSQL stores, based on the TinkerPop BluePrints framework [74]. Although the common framework BluePrint API adds performance overhead (varies among databases), it allows a wide range of databases to be benchmarked. The first phase involves inserting large models into databases and performing a specific query against the models inserted for different databases. The aim of phase one is to identify the general performance profile for each category of databases. The second phase involves exploring the performance of several promising NoSQL databases identified in the first phase using their native interface. By doing so it eliminates the overhead incurred by the BluePrints framework.

Rather than the synthetic, generated data sets used in [26], we used reverse-engineered software models from real projects as benchmarking data; we argue that this is preferable in cases where real-world large models are not readily available. The GRABATs 2009 data set[2] is a set of readily available large-scale models and is used Section 5 and Section 6. Alternatively, the MoDisco framework [19] can be used to create further data sets from Java code projects.

Section 3 sets out general and application specific requirements for large-scale model persistence. These requirements have been used to select initial candidates for the identified database technologies. In the first phase, model insertion and database query are performed, the ones that are not performant will be eliminated. The following phases involve defining the critical benchmarking tasks and minimum expected performance in each task for the intended application. After benchmarking has been carried out in each phase, databases that do not meet the minimum expected performance or do not have the required features should be eliminated. In Section 5, model persistence and model querying performance are set as critical benchmarking tasks, using the common framework. In Section 6, model persistence and model querying performance are set as critical benchmarking tasks, using the native interfaces provided by the databases.

[2] http://www.emn.fr/z-info/atlanmod/index.php/GraBaTs_2009_Case_Study

4.1 Benchmarking Dataset: GRABATS 2009 models

This section introduces the dataset used in experiments of Section 5 and Section 6, consisting of large models reverse engineered from open source projects.

4.2 Dataset Characteristics

	set0	set1	set2	set3	set4
XMI file size	9.2m	27.9m	283.2m	626.7M	676.9M
# of objects	69,680	197,699	2,082,841	4,594,899	4,961,779
# number of relations	69,806	197,965	2,083,272	4,595,522	4,962,567
# total elements	139,486	395,664	4,166,113	9,190,421	9,924,346

Table 3: Grabats dataset characteristics

To obtain meaningful benchmarking results, there is a need for representative large models. As mentioned earlier, instead of using synthetic models, we used models reverse-engineered from open source Java projects. The JDTAST metamodel used in Java Legacy Reverse-Engineering use case, presented in the GRABATS 2009 contest, is used, as well as the five models provided in the contest. The JDTAST metamodel has similar concepts to the Java programming language and allows representation of Java programs as models. Five large models have been extracted from existing Java code that conform to the JDTAST metamodel. These are EMF models serialised in XMI format and they range from 69680 model elements and 69680 relationships in a 9.2MB XMI file (set0), to 4961779 model elements and 4962567 relationships in a 676.9MB XMI file (set4). Detailed characteristics of all five models are displayed in Table 3.

4.3 Experiment Setup

For the database benchmarking experiment, a single, commercial class computer is used. The experiment computer is equipped with Intel Core i7 2.3GHz CPU, 8 GB of DDR3 memory and a 256GB Solid State Disc running OS X 10.9.1 and JDK 1.7.0. For each of the experiment, a maximum heap memory of 6 GB is allocated.

5 Phase One: Benchmarking a Wide Range of NoSQL Stores using a Common Framework

5.1 A Framework to Benchmark NoSQL Stores

The framework is designed to be modelling technology agnostic. Although at the current stage this framework interacts with EMF [20] models only, it can be extended to support other modelling technologies such as Express [4]. The framework presented takes inspiration from the Yahoo Cloud Service Benchmark (YCSB) framework [26], for On-Line Transaction Processing, and is an extensible and generic framework to evaluate NoSQL key-value stores using synthetic data and workloads. Similar to YCSB, the current work provides drivers for several NoSQL databases, but proposes several essential improvements towards benchmarking for large-scale modelling. Section 5.1, describes an architecture for benchmarking a broad range of model serialisation formats and NoSQL stores. Section 5.2 discusses how different types of stores can be used to persist large-scale models. Our approach also provides several ready-made driver implementations, which are benchmarked in Section 5.3.

After surveying different database technologies, it is found that the most of them support a common interface called TinkerPop BluePrints [74]. BluePrints is a property graph model interface with several concrete implementations. Databases that implement the BluePrints interfaces automatically support BluePrints-enabled applications. BluePrints is open source; implemented in Java; and currently has drivers for databases such as Neo4J [78], Sail [17], Sparksee [71], Accumulo [2], ArangoDB [6], FoundationDB [40], MongoDB [25], Oracle NoSQL [69], OrientDB [62] and TitanDB [75]. Theoretically, BluePrints drivers can be developed for any database technology; the minimum requirement is that the database provides either a Java interface or a REST API.

The structure of the proposed benchmarking framework is shown in Figure 1. The framework supports a number of database technologies by implementing new or reusing existing BluePrints drivers. The persistence layer is responsible for loading models defined in different modelling technologies and persisting them into the designated database. The persistence layer contains definitions of different configurations of modelling and database technologies.

5.2 Model Persistence Formats For NoSQL

The BluePrints API exposes the underlying NoSQL databases as a property graph which we exploit to persist models in our benchmarking framework. The property graph model of NoSQL graph databases, such as Neo4J and OrientDB, employs a similar structure to models, making model representation straight-forward. In order to map from EMF to the BluePrints API, objects are represented as graph vertices and object-attributes are represented as properties of these vertices. Multi-valued attributes are stored as lists, which are sub-documents in the vertex properties. Relationships between objects are represented as edges in the property graph. Similar to object-attributes, reference features such as name, directionality, ends' names and containment are kept in the properties of the edge. Figure 2a shows a model that is represented as a property graph in Figure 2b.

The BluePrints drivers for each database control how property graphs are represented using the primitives provided by each type of database. For example, in triple store databases, the basic data entities are triples. There is no mean to differentiate between object-instances and object-attributes for triples. Triples are in the form of $subject-predicate-object$, where subject and object are uniquely

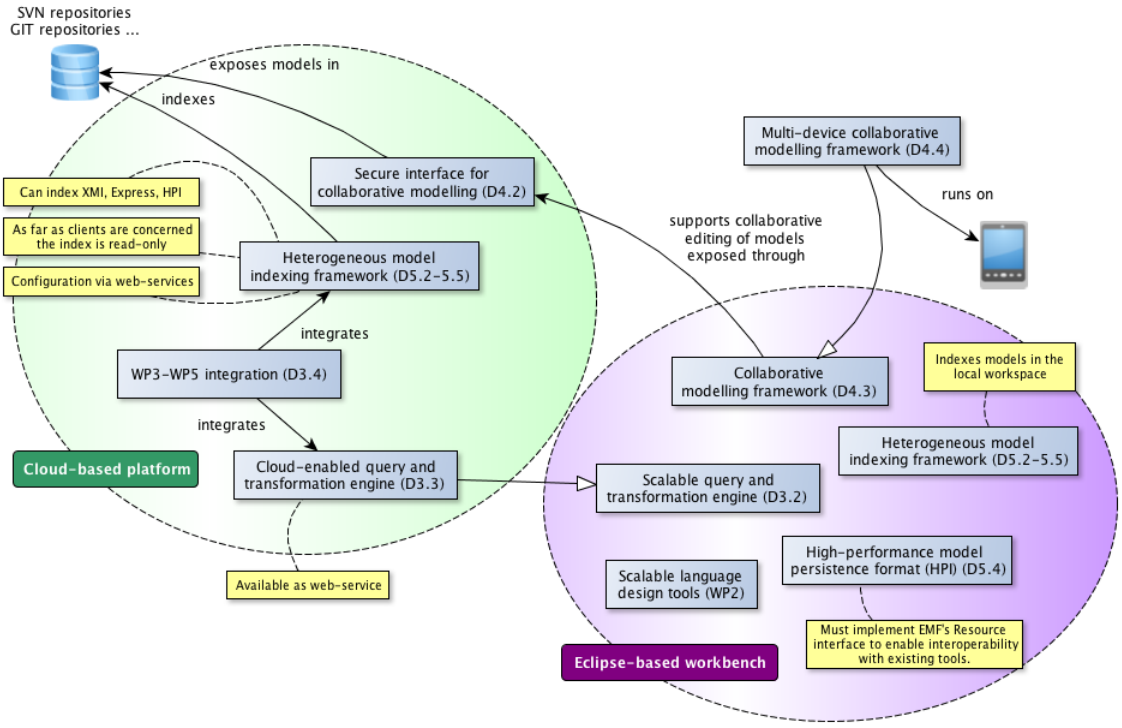
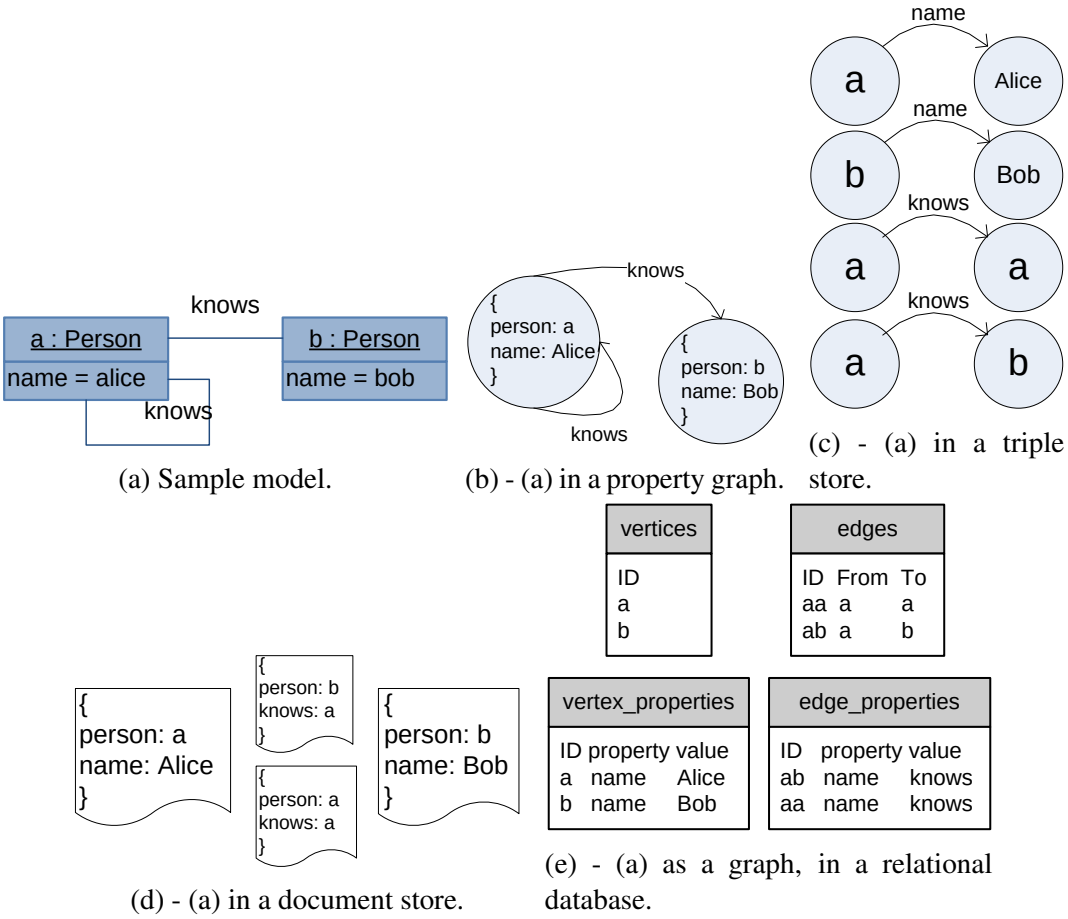


Figure 1: Structure of the framework.

identifiable entities. To store models in a triple store database, triples are used to represent the relationships between two objects; triples are also used to represent what attribute an object possesses, this is shown is Figure 2c. A notable aspect of this structure is that entities are only stored in triples, so that an object can appear as the subject of multiple triples. On the other hand, objects cannot be retrieved independently of triples.

The BluePrints API may be used with any kind of database. For example, although PostgreSQL is a relational database, it can replicate the schema-less design of a property graph database via an appropriate BluePrints driver. The four relational tables needed are Vertices, Edges, Vertex-Properties and Edge-Properties linked by traditional relational foreign keys. An example of model persistence using this structure is shown in Figure 2e.

Several document databases and key-value databases do not support references between entities, which precludes those databases from the study in this report, since it incurrs high performance overhead to establish links between two entities in such databases. Both types of databases have entities that can be used to store object instances and object attributes, as documents. In order to support key-value stores and document stores in the BluePrints property-graph model, special considerations must be made for edges. In some cases, the model persistence layer can include work-arounds to support edges with properties, for example in the case of the MongoDB BluePrints driver a dedicated document is created that links documents together, as shown in Figure 2d.



(a) Sample model.

(b) - (a) in a property graph.

(c) - (a) in a triple store.

(d) - (a) in a document store.

(e) - (a) as a graph, in a relational database.

Figure 2: Model Persistence Formats in NoSQL Stores

5.3 Results

This section discusses the results when the benchmarking framework is applied to the GRABATS datasets and the selected NoSQL databases. Table 4 and Figure 3 outline the results for persisting large models to each database. When persisting models, the 'Baseline' time is the taken to load, traverse and unload each model, without any interaction with the data store. This represents the overhead of the framework. In traversing the data store the baseline is the overhead of starting and executing the framework, without connecting to a data store. In Table 4, a dash (-) indicates a database that could not complete the benchmark, within the time or memory given. Some results shown in Table 4 and Table 5 may not appear in Figure 3 and Figure 4 to ease readability. Several interesting observations can be made about relative performance of the databases and the architecture of BluePrints drivers.

The Neo4J (versions 1.x and 2.x) and ArangoDB stores were unable to complete the benchmark for data sets set3 and set4, for the dataset is introduced in Section 4.1. This is due to the architecture of the BluePrints drivers. ArangoDB, uses a client-server mechanism, its BluePrints driver uses ArangoDB's REST API, however, the ArangoDB's implementation of BluePrints sends a separate HTTP request per object-level insertion and does not support batch processing. Both standard Neo4j

and ArangoDB drivers store a cache of each insertion (i.e. the whole graph) in memory, causing model insertion processes to run out of memory. The Neo4j store can only persist set3 and set4 when using a special 'batch' insertion driver and protocol. In Neo4J the batch driver includes the logic to modify the underlying database files directly, which is a write-only process, thus the database cannot perform any read operations during the insertion. Also in both batch and standard from, Neo4j 1.x performs better than Neo4J 2.x, possibly due to the stability and enhanced optimisation of the older version. After contacting the ArangoDB developers they managed to send back a bulk-insertion BluePrints driver, the theory behind this driver is to accumulate insertion requests and send a single HTTP request to the database with a given parameter (limit of the maximum request), which significantly improves performance. However, during the insertion the database is also write-only. This makes databases with bulk-insertion drivers clearly more suitable for persisting large models, where model insertion time is critical but the model will not be modified or accessed during insertion.

The TitanDB/Cassandra and TitanDB/BerkleyDB drivers are able to insert all data sets, without using a batch insertion, because the graph is not cached by the driver in memory. However, this impacts the performance, making the store take longer to insert large models. However, the drivers highlight the difference in using an embedded database as a back-end and using client-server back-end for the database. In TitanDB/Cassandra, the textual CQL language and protocol are used to store graph data, whereas TitanDB/BerkleyDB uses an embedded file based database to persist graphs. The difference is apparent in the benchmark results for set3 and set4, where TitanDB/BerkleyDB takes half the insertion time of TitanDB/Cassandra. TitanDB/BerkleyDB provides the fastest insertion time without using batch insertion, however, the driver is less stable than TitanDB/Cassandra and can run out of memory unexpectedly as performing insertion for set4 only succeeded 60% at all time.

The OrientDB and MongoDB BluePrints drivers use a binary protocol to communicate with the server and do not keep a cache of the graph in memory, so they are able to insert the larger data sets without a batch driver. The insertion performance for set3 and set4 is lower than Neo4J or ArangoDB because both the drivers for OrientDB and MongoDB allow transactions. To tune the performance, we specify that the databases should commit changes for every 50,000 changed elements otherwise the database will not be able to handle committing changes for almost 10 million elements for set4. Thus, at the commit stage the databases perform integrity checks and therefore incurrs performance over-heads. The OrientDB and MongoDB stores are suited to workloads where model elements may be accessed or modified during insertion and model insertion time is critical.

The Sesame database driver has a similar architecture to the Neo4J driver, directly accessing the data store in embedded mode[3] to create or modify a file-based store that may later be served by a triple store. The PostgreSQL driver uses the text-based JDBC protocol to create a simple schema for storing graphs on a database server. However, Sesame and PostgreSQL were unable to complete insertion benchmarks for set3 and set4, as both cache the graphs in memory during insertion and do not support bulk insertion. These drivers are therefore not suited to large scale model persistence.

Table 5 and Figure 4 outline the results for traversing all stored elements. In Table 5, a dash (-) indicates those data stores which could not be benchmarked as the model was not persisted in the previous benchmarking phase. Both Sesame and PostgreSQL have better relative model traversal performance than many of the others stores for set0 and set1. Sesame, performs better than Neo4J in

[3] A SPARQL (REST-like) client-server architecture is available, with more overhead than the embedded triple store used in our experiments.

Store	set0	set1	set2	set3	set4
Baseline	6s	15s	54s	218s	303s
Neo4J 1.x	9s (44MB)	29s (129MB)	-	-	-
Neo4J 2.x	21s (44MB)	40s (129MB)	-	-	-
Neo4J Batch 1.x	9s (14MB)	18s (44MB)	118s (334MB)	431s (744MB)	852s (805MI
Neo4J Batch 2.x	6s (15MB)	19s (45MB)	256s (334MB)	756s (767MB)	940s (830MI
OrientDB	16s (35MB)	39s (98MB)	438s (955MB)	1031s (2.11GB)	1127s (2.28(
TitanDB (BerkeleyDB)	11s (48MB)	25s (147MB)	196s (1.29GB)	814s (3.01GB)	997s (3.32G
TitanDB (Cassandra)	38s (75MB)	90s (165MB)	876s(1.06GB)	1967s(1.75GB)	2419s(1.81G
MongoDB	33s (807MB)	100s (1.28GB)	983s (4.77GB)	2537s (6.67GB)	2839s (6.98(
Sesame	87s (33MB)	265s (97MB)	3140s (734MB)	-	-
PostgreSQL	460s (206MB)	1216s	-	-	-
ArangoDB	420s (71MB)	1620s (194MB)	-	-	-
ArangoDB Batch	8s (71MB)	18s (104MB)	174s (641MB)	-	-

Table 4: Model persistence performance - time taken in seconds and disc space used, per dataset.

smaller sets but not in set2, which indicates that its traversal capabilities may not scale, even if the larger stores can be persisted. These databases are not suitable for large scale model persistence in the current form, however, they may be suitable where traversal performance is important in data sets similar to set0 and set1, and insertion time is not critical.

As with model insertion, Neo4J 1.x performs better than Neo4J 2.x for model traversal, which suggests the newer version is less optimised than the earlier stable release. MongoDB suffers at model insertion-time due to the indexes automatically created to address each element stored documents, which also negatively impacts on disc space used for each dataset. The indexing is beneficial, as MongoDB has similar relative performance for the larger set3 and set4 sets (within $\sim$10 seconds) and better performance than Neo4J in the set0, set1 and set2 datasets. OrientDB also has better model traversal performance than Neo4J in set0 and set1, which seems to indicate that Neo4J does not scale down to smaller datasets as well as MongoDB or OrientDB. However, OrientDB does not perform as well as either MongoDB or Neo4J on set2, set3 or set4, indicating issues with scaling OrientDB up to larger models.

TitanDB with the relational BerkleyDB back-end and ArangoDB were the only two stores where a model was persisted but could not later be traversed, for the model in set4 and set2 respectively, where the driver ran out of memory. This seems to indicate the limits of TitanDB when using BerkleyDB. When using Cassandra with TitanDB, traversal performance is lower than with BerkleyDB. The worst traversal performance is found in ArangoDB, where separate HTTP REST requests are used to retrieve individual model elements, creating a significant overhead.

5.4 Threats to Validity

Standard desktop specification machines have been used to perform benchmarking, so the results are representative of a desktop environment. In future work, the NoSQL systems will be bench-

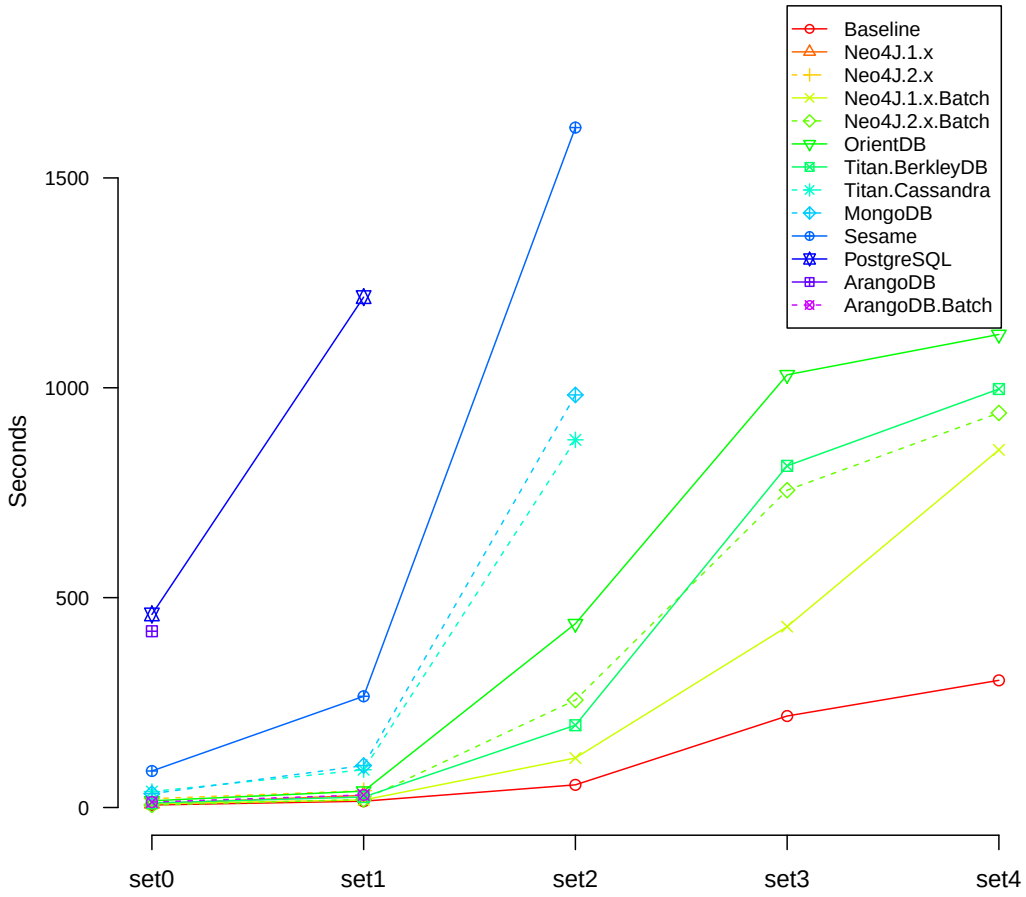


Figure 3: Model persistence performance - time taken in seconds, per dataset.

marked using servers with greater resources available and the results compared with desktop-type performance.

In these experiments, the BluePrints API has been used to create a common interface to load data into and to query the databases. This allows us to treat the databases as 'black-box' and benchmark a broad range of them. This approach relies on the availability of high-quality, optimised BluePrints drivers for each database. In the case of ArangoDB and Neo4J, batch insertion drivers are available, and Neo4J also has targeted different versions of the underlying databases. However, most drivers do not support batch insertion, even where the underlying database does, which has a negative impact on performance. Similarly, the underlying databases have various parameters that could be modified to improve performance but the configuration parameters are not always exposed by the drivers. In future work, each driver and database will be investigated for possible optimisations, based on the obtained benchmark results.

Large models conforming to the JDT meta model have been used to benchmark a wide range of databases. The JDT meta model stores Java source code in a model based format and the models are created from reverse engineering large Java code-bases. The largest model has 5 million class-

Store	set0	set1	set2	set3	set4
Baseline	1s	1s	1s	1s	1s
Neo4J 1.x	4s	6s	12s	24s	28s
Neo4J 2.x	6s	7s	16s	27s	37s
OrientDB	3s	6s	53s	192s	870s
TitanDB (BerkleyDB)	6s	14s	186s	2414s	-
TitanDB (Cassandra)	14s	53s	400s	1212s	1263s
MongoDB	2s	2s	16s	40s	44s
Sesame	2s	3s	17s	-	-
PostgreSQL	2s	2s	-	-	-
ArangoDB	60s	180s	-	-	-

Table 5: Model traversal performance - time taken in seconds, per data set.

instance elements with around 5 million references between class-instances and other models in the data set are similarly connected. The models used in benchmarking were selected based on the size and availability, so the results presented here may not be representative of other metamodels. Researchers and practitioners are invited to make available large models for benchmarking in this framework.

5.5 Conclusions

We have presented a framework and methodology for benchmarking NoSQL datastores in the context of large-scale modelling applications. The framework builds on the BluePrints property graph model interface and provides a layered architecture to avoid repetition and to allow future support of non-EMF modelling technologies. The framework advocates the selection of candidate databases via a multi-stage process, which involves consulting the literature and existing benchmarks.

Our experiments have investigated the persistence and traversal performance of models of different sizes, for a realistic case study that involves persisting large models reverse engineered from open source projects. The experimentation revealed that OrientDB, Neo4J (both v1.x and v2.x) and TitanDB (using BerkeleyDB) scale up well in terms of persisting large models, while Neo4J (both v1.x and v2.x) and MongoDB scale up well for querying such models. Also, unexpectedly many of the evaluated databases were unable to persist/traverse models larger than set2 in time and under the provided resources.

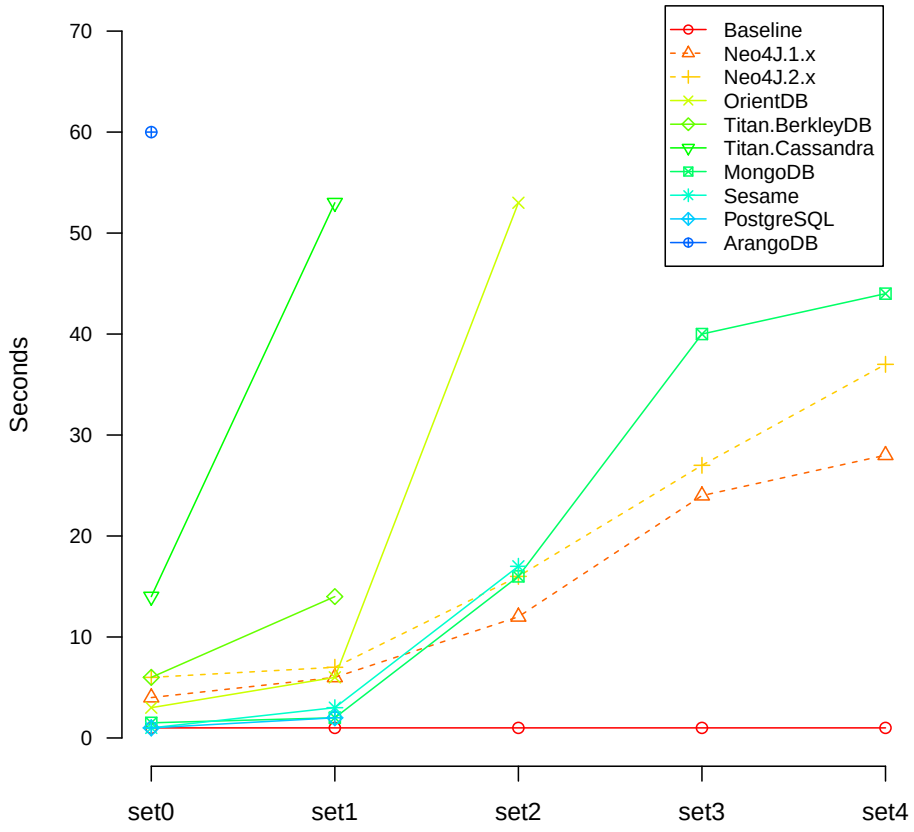


Figure 4: Model traversal performance - time taken in seconds, per data set.

6 Phase Two: Benchmarking Selected Stores using Native APIs and Tools

6.1 Graph Databases

Due to the highly-interconnected nature of typical models used in MDE, key-value databases as well as tabular databases, after benchmarking, are not suitable for persisting large scale models. In this section, two database technologies are evaluated. They are OrientDB, which is a document based and hybrid-graph database; and Neo4J, which is a pure graph database. Instead of using BluePrints drivers for these two databases, their native APIs are used to persist models and perform queries against them. In this section, Neo4J and OrientDB are evaluated for performance using their native APIs to persist models and perform queries against them.

6.1.1 Methodology and Implementation

In the experimentation, both the technologies share the same principle in terms of methodology and implementation. The principles behind graph-based databases are listed as the following; in a graph-based database, it typically contains:

- $Nodes$ representing model elements (or model element instances) in the model stored. The attributes of the model elements (as defined by the $EClass$) are stored as properties of the nodes.
- $Relationships$ from model element nodes to other model element nodes. These represent the $EReferences$ of one model element to other model elements.
- $Nodes$ representing $EClasses$ of the metamodel(s) the stored models are instances of. These nodes only have an id property denoting the unique identifier (URI) of the metamodel they belong to, followed by their name, ie: $org.amma.dsl.jdt.core/IJavaElement$ is the id of the $EClass\ IJavaElement$ in the $org.amma.dsl.jdt.core$ Ecore metamodel. These lightweight nodes are used to speed up querying by providing references to model elements that are instances of this EClass as EClasses that inherit from it.
- An index containing the ids of the EClasses and their appropriate location in the database. This allows typical queries (such as the GRABATS query) to use an indexed $EClass$ as a starting point, in order to find all model elements of a specific type, and then navigate the graph to return the required results.

It is worth to note that the above data does not contain detailed metamodel information, this is due to the fact that metamodels are typically small and sufficiently fast to navigate using the default EMF API. Action that require use of such metamodel information, like querying an element for certain properties or whether a reference is a containment require access the metamodel, to retrieve this information. Thus, the above technique for models persistence contains enough information required to load a model and evaluate EMF queries whilst the metamodels are also available at query-time.

6.1.2 Interfaces and Tools in OrientDB and Neo4J

Since the release of version 1.4.1, OrientDB provides their implemented BluePrints driver that outperforms its previous and deprecated API (the $OGraphDatabase$ API). Other than the BluePrints driver, OrientDB also provides an API which creates a document based database using the $ODatabaseDocumentTx$ API. Additionally, OrientDB also provides an API that creates a flat database by using the $ODatabaseFlat$ API to achieve maximum performance. In this configuration, entries are treated as Strings and no queries can be performed at all. In terms of performance comparison, according to OrientDB, the original API OGraphDatabase achieves 60% of the maximum performance capacity; BluePrints driver OrientGraph achieves 80% of the maximum performance capacity; document based API ODatabaseDocument achieves 80% of the maximum performance capacity; and flat database API ODatabaseFlat achieves 100% of the maximum performance capacity. Due to the nature of the experiment, the graph database is preferable than document based database. Additionally, the graph database and document database APIs achieve the same performance. Although the purpose of this section is to evaluate the native APIs for selected databases, the BluePrints

	set0	set1	set2	set3	set4
Neo4J	5s (14MB)	9s (44MB)	63s (334MB)	188s (744MB)	266m (805MB)
OrientDB	16s (35MB)	39s (98.3MB)	438s (955.1MB)	1031s (2.11GB)	1127s (2.28GB)

Table 6: Model load time performance and disc-space take per model.

	set0	set1	set2	set3	set4
Neo4J	0.276s	0.621s	3.083s	5.163s	5.502s
OrientDB	2.12s	4.84s	52.66s	153.0s	225.31s

Table 7: Model query time performance and disc-space take per model.

driver developed by OrientDB can be considered as its native API as it allows fine grained performance tuning. Thus, we decide to benchmark the BluePrints driver and refine the algorithm and configuration to achieve best possible performance.

Neo4J native APIs supports batch insertion by directly writing strings into the database to improve performance. However, during batch insertion the database is in a write-only state, thus data cannot be retrieved from the database. This brings a challenge when inserting Edges into the databases, as the insertion of an Edge requires the entities on both ends of it - which means accessing the database for the look ups of the nodes. However, a work-around has been implemented that involves using a temporary in-memory HashMap to keep track of nodes inserted into the database, so that retrieving nodes does not involve reading from the database. This will eventually introduce the challenge when models grow significantly large in the order of gigabytes.

6.1.3 Results

The results for loading GRABATS 2009 models into OrientDB and Neo4J are presented in Table 6. For model insertion, OrientDB scales linearly both in terms of time taken for the insertion and disk space taken.

6.2 Triple Stores

In this section, the Sesame, Jena and Four-Store triple stores are evaluated for performance using the native APIs to persist and query data. Triple stores are selected because of the performance of the Sesame store in the traversal benchmarks in Section 5. Triple stores use common semantic-web formats for representing data and provide tools for bulk loading data. SPARQL a high-level, declarative SQL-like language is used as a common language for querying triple stores using graph patterns.

	set0	set1	set2	set3	set4
XMI file size	9.2MB	27.9MB	283.2MB	626.7MB	676.9MB
Turtle file size	21MB	200MB	500MB	1.1GB	1.2GB
Conversion time	11s	21s	3m	9m	10m

Table 8: Model Conversion Time and Space Costs

6.2.1 Methodology and Implementation

In order to carry out benchmarking, models must be converted to a format that can be read by triple stores. Triple stores generally support RDF/XML, RDF/JSON, Notation 3, N-Triple or Turtle on-disc triple notations and programming APIs, SPARQL interfaces, HTTP REST interfaces or specific bulk insertion tools for importing triples into the store. Turtle is designed as read only-format, storing triples one per line without any XML tags or formatting rules, making it less verbose, less complex and therefore more efficient than RDF/XML.

Large models in Turtle notation can be split into multiple smaller files and at any point in the file to faster bulk uploading. The bulk insertion tools can either be used directly via command line tools or via a programming language API for efficiently inserting large number of triples into a store. Turtle is used as the intermediate format to import models using the bulk insertion tools. A Java driver is created to convert EMF XMI documents into triples in turtle notation.

Model conversion time and disc space taken in turtle format is shown in Table 8. Once the models are converted into the triple format, the data import tool of each store is used to load the model into a newly created database. The GRABATS query is shown in Listing 6.2.1, used by each database. The query defines a triple-traversal pattern which identifies singleton classes (classes with public, static methods, with the same return type as the containing class) in the data set.

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX dom:<http://org.amma.dsl.jdt.dom#>
PREFIX ptypes:<http://org.amma.dsl.jdt.primitiveTypes#>
SELECT
        DISTINCT (COUNT(*) AS ?count)
WHERE {
        ?td a dom:TypeDeclaration .
        ?td dom:name ?tdname .
        ?tdname ptypes:fullyQualifiedName ?rtfullname .
        ?td dom:bodyDeclarations ?md .
        ?md dom:modifiers ?m .
        ?md dom:modifiers ?m2 .
        ?m ptypes:public "true"^^ptypes:Boolean .
        ?m2 ptypes:static "true"^^ptypes:Boolean .
        ?md dom:returnType ?rt .
        ?rt dom:name ?rtname .
        ?rtname ptypes:fullyQualifiedName ?rtfullname
}
```

Listing 1: GRABATS query in SPARQL format.

6.2.2 Interfaces and Tools in Sesame, Jena and 4Store

Sesame is a Java based framework for triple storage access, querying and reasoning. It supports a variety of storage back-ends. Various databases claim to be Sesame-compatible and so are accessed using the Sesame API and tools. Triple back-ends storage including on-disc, in-memory, commercial storage servers and traditional relational databases. Sesame provides a Java based command-driven tool (sesame-console), Java APIs as well as a HTTP-REST-like SPARQL interface to create, modify and query triples. Sesame also provides the workbench, a HTTP user interface for manipulating triples, the interface allows triple-import via files. The console wraps the Java API for convenience, translating user-entered commands into Sesame Java-API calls. The console also supports bulk uploading of data, which can also natively load data from triple files within a ZIP file, to reduce the memory footprint. The console can also be used to query triples once data has been loaded.

Apache Jena is a Java based triple store that uses a native format for storing triples. Jena also provides console tools, a Java API and a HTTP SPARQL interface to interact with the stored triples. The console tools wrap the Java API, similar to Sesame, however the Jena console tools are stand alone shell scripts as opposed to the integrated Java console of the Sesame workbench. The Jena bulk upload tool is notable in spliting large files into smaller batches and automatically optimising batch size based on available RAM, to improve performance. Jena APIs includes the AQL toolkit, which can be used to build SPARQL queries using Java Objects, avoiding string munging[4] that is required when building queries that contain quotes. The Fuseki SPARQL server in Jena includes an API to import file, however large files must be split as there is a time-out for files uploaded via this interface.

The 4store triple store and tools are written in C. The command line tools or the HTTP SPARQL interface can be used to access and manipulate triple stores. The command line tools connect to the store using a binary protocol to perform queries. The HTTP API includes extensions for bulk triple upload via files, however as with Jena the upload file size per request is limited. 4Store also allows for a distributed storage architecture, that allows data to be stored across many nodes to improve performance.

6.2.3 Model Persistence Considerations for Triple Stores

When persisting models using triple stores some application-level considerations must be made in order to preserve semantic integrity. Firstly, even though triple stores have a notion of type and schemas, as with other approaches validation of models for meta model conformance is not directly supported. Direct manipulation of triples can lead to inconsistent models. So conformance must be checked by client applications during model manipulation. The meta models of a model are not stored explicitly, only meta model elements used in a model are stored, within instances of the meta element. Object and relationship types are inferred from labels and when needed, for example in queries.

In triple store, literals are stored as Strings with the type name, so certain special characters - double-quotes, new lines, carriage return, tab and backslash - must be escaped. Finally, duplicate triples are not allowed which means duplicates are not allowed within certain structural features, such as multi

[4] Where complex strings that include Java special characters must be created in order to query the store.

	set0	set1	set2	set3	set4
4Store	4s (119MB)	19s (200MB)	540s (500MB)	5279s (2.2GB)	5827m (2.3GB)
Sesame (in-memory)	11s	15s	124s	360s	496s
Sesame (disc)	12s (35MB)	24s (102MB)	234s (731MB)	731s (1.5GB)	805s (1.7GB)
Jena	23s (34MB)	44s (96MB)	293s (727MB)	842s (1.6GB)	990s (1.8GB)

Table 9: Model persistence performance and disc-space taken per model.

	set0	set1	set2	set3	set4
4Store	0.030s	0.042s	2s	10s	12s
Sesame (in-memory)	0.047s	0.021s	0.723s	3s	4s
Sesame (disc)	0.54s	1s	2s	11s	13s
Jena	2s	2s	18s	102s	134s

Table 10: Model query performance in triple stores.

valued list attributes or relationships with 'many' multiplicity. To support models with duplicate instances in multi valued attributes, the duplicates can be made distinct for example by enumeration.

When uploading large models to triple stores, the main bottleneck is hard drive performance and communication with the interface. More efficient bulk upload may be possible, as the model conversion and triple upload steps may be combined, to avoid reading from and writing to disc. Multi threaded upload in the client can overcome communication bottleneck as well as memory backed storage such as a ram disc for the triple storage. Further performance gains many be possible by turning off well-formedness checks, as currently each character of each literal is checked for special characters.

6.2.4 Results

This section discusses the results of benchmarking Jena, Sesame and 4Store triple stores using native interfaces and the GRABATS data set and queries. Table 9 and Figure 5 outline the results for persisting large models to each store. These results do not include the time to convert models to triples, give in Table 8. Results for querying the triple stores, results are shown in Table 10 and Figure 6. The rest of this section outlines some interesting observations on the relative performance of each triple store.

In terms of insertion and query performance, 4store has better relative results for smaller datasets than the other triple stores. Querying the larger data sets is only slower than the Sesame in-memory store and has similar performance to the Sesame on disc store for larger sets, so has relatively good performance in querying. However, inserting large models in set3 and set4 into 4store, even using the bulk insertion tool, is significantly worse in 4store than any other store. The results indicate 4store has high over-head when creating triples and indexes on-disc, and it also uses the most disc-space per

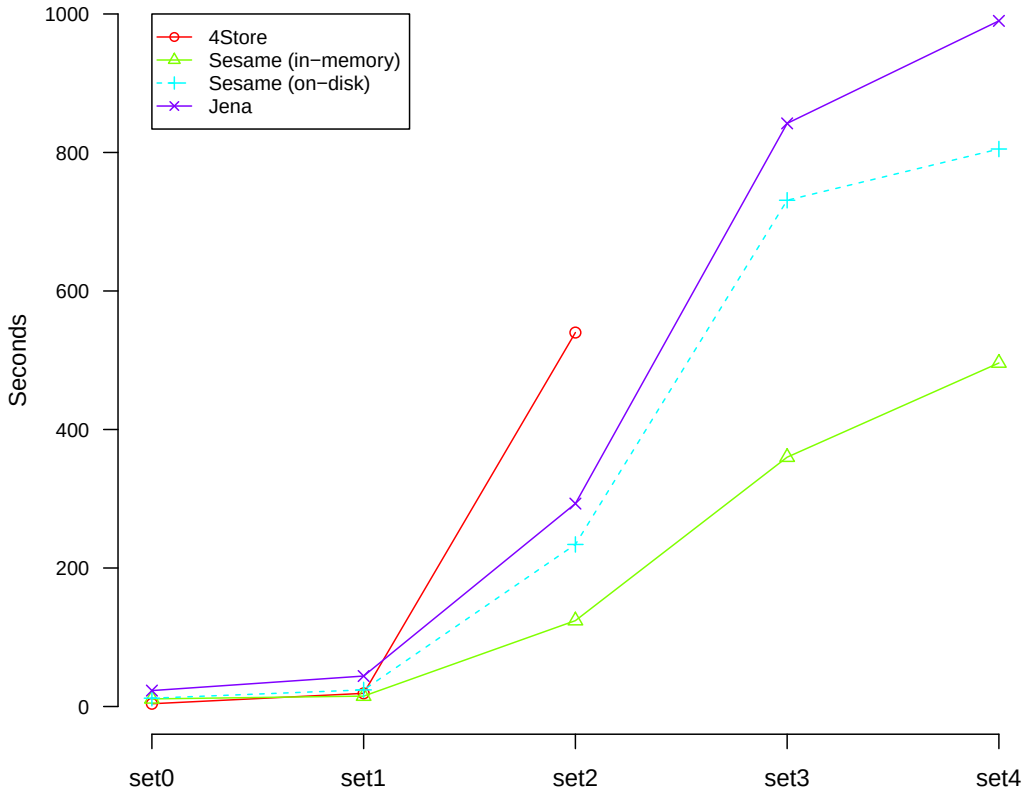


Figure 5: Model persistence performance in triple stores- time taken in seconds, per dataset.

dataset. The indexes in 4store are not configurable and the bulk insertion tool is may not be optimised for single node, disc-based configurations.

Sesame in-memory query performance for all data sets is better than any other store, as there is no disc reading involved. The in-memory store can optionally persist the index of triples to disc when the store is shut-down, which means the index can be reused in subsequent executions. The insertion time for the Sesame in-memory store is also better than any store and scales well to larger data sets. Sesame on-disc model insertion is better than any other on-disc store, and also scales well to larger data sets. The results for Sesame are using the default triple indexes, however these may be customised to improve performance.

Jena has similar model insertion performance to Sesame and scales to larger sets but is slightly worse in terms of insertion for larger sets than Sesame. In Jena, more indexes are created, reflected in the slightly longer insertion time and disc space used. The extra time and disc space used by Jena for indexing is not reflected in the query time as Jena is slower in all data sets and does not scale well to larger models in set3 and set4. This indicates sub-optimal indexes are created by default in Jena. As

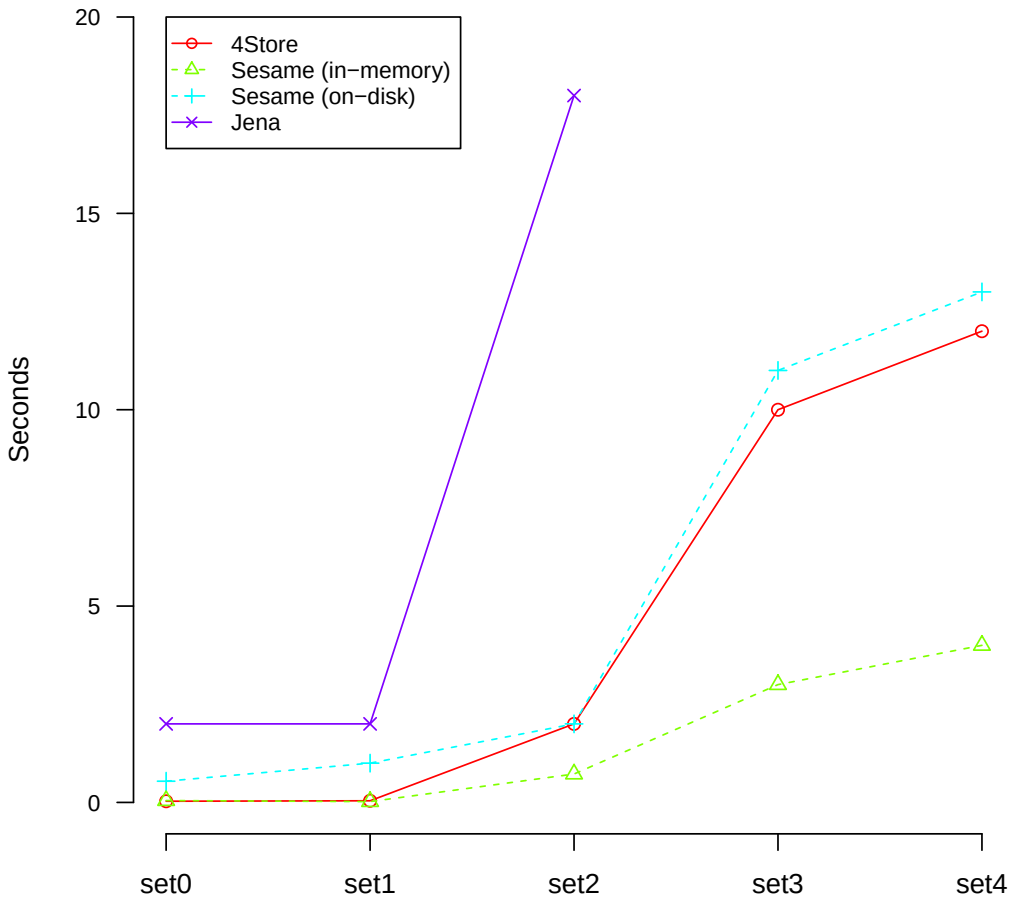


Figure 6: Model query performance in triple stores- time taken in seconds, per dataset.

with Sesame, Jena indexes may be customised when a store is initialised, and changing the default indexes could lead to improved performance in Jena.

Conclusion

This document has presented results on benchmarking candidate data stores that can underpin the heterogeneous model indexing framework that will be developed in Tasks 5.2 and 5.3 of the project.

The experiments were carried out in a two-phased approach; in the first phase, a wide range of stores were evaluated using a common abstraction layer (BluePrints), and in the second phase, the most

promising stores (two Property Graph Databases and three Triple Stores) were benchmarked using their native APIs.

The obtained results indicate that the Neo4J Property Graph Database delivers the best performance in both persistence and querying when bulk insertion is used. As such, our plan is to use Neo4J as the infrastructure on which we will build the envisioned heterogeneous model indexing framework in Tasks 5.2 and 5.3.

References

[1] 4store. http://4store.org/.

[2] AccumuloDB. https://accumulo.apache.org/.

[3] AeroSpike. http://www.aerospike.com/.

[4] Yamine Ait-Ameur, Frederic Besnard, Patrick Girard, Guy Pierra, and Jean Claude Potier. Formal specification and metaprogramming in the EXPRESS language. In *Intern. Conference on Software Engineering and Knowledge Engineering SEKE*, volume 95, pages 181–189, 1995.

[5] AllegroGraph. http://franz.com/agraph/allegrograph/.

[6] ArangoDB. https://www.arangodb.org.

[7] A. Auradkar, C. Botev, S. Das, D. De Maagd, A. Feinberg, P. Ganti, Lei Gao, B. Ghosh, K. Gopalakrishna, B. Harris, J. Koshy, K. Krawez, J. Kreps, Shi Lu, S. Nagaraj, N. Narkhede, S. Pachev, I. Perisic, Lin Qiao, T. Quiggle, Jun Rao, B. Schulman, A. Sebastian, O. Seeliger, A. Silberstein, B. Shkolnik, C. Soman, R. Sumbaly, K. Surlaker, S. Topiwala, C. Tran, B. Varadarajan, J. Westerman, Z. White, D. Zhang, and J. Zhang. Data Infrastructure at LinkedIn. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1370–1381, April 2012.

[8] Konstantinos Barmpis and Dimitrios S. Kolovos. Comparative Analysis of Data Persistence Technologies for Large-Scale Models. In *XM@MoDELS*, 2012.

[9] Konstantinos Barmpis and Dimitrios S. Kolovos. Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models. *Journal of Object Technology, to appear*, 2014.

[10] Konstantinos Barmpis and Dimitris Kolovos. Hawk: towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 6:1–6:9, New York, NY, USA, 2013. ACM.

[11] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A Graph Query Language for EMF Models. In *ICMT*, pages 167–182, 2011.

[12] Bigdata. http://www.systap.com/bigdata.htm.

[13] BigOWLIM. https://www.ontotext.com/owlim.

[14] Bitsy. https://bitbucket.org/lambdazen/bitsy/wiki/Home.

[15] Eric A Brewer. Towards robust distributed systems. In *PODC*, 2000.

[16] BrightstarDB. https://brightstardb.com/.

[17] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *The Semantic Webï¿½ISWC 2002*, pages 54–68. Springer, 2002.

[18] Martin C. Brown. *Getting Started with CouchDB - Extreme Scalability at Your Fingertips.* O'Reilly, 2012.

[19] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 173–174. ACM, 2010.

[20] Frank Budinsky. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004.

[21] Tokyo Cabinet. http://fallabs.com/tokyocabinet/.

[22] (CDO). Connected Data Objects. http://www.eclipse.org/cdo/documentation/index.php.

[23] CelocityGraph. www.velocitygraph.com.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[25] Kristina Chodorow and Michael Dirolf. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage.* O'Reilly, 2010.

[26] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[27] CouchDB. http://couchdb.apache.org/.

[28] Big Data. http://www.bigdata.com/blog/.

[29] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[30] DEX. sparsity-technologies.com/dex.

[31] eclipse modeling framework technology (emft) Teneo. http://www.eclipse.org/modeling/emft.

[32] EJDB. http://ejdb.org/.

[33] Javier Espinazo-Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. A repository for scalable model management. *Software & Systems Modeling*, pages 1–21.

[34] Javier Espinazo-Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *MoDELS*, pages 77–92, 2011.

[35] Javier Espinazo-Pagán and Jesús García Molina. Querying large models efficiently. *Information and Software Technology*, 56(6):586–622, 2014.

[36] FATDB. http://fatcloud.com/overview.html.

[37] Filament. http://www.bigdata.com/blog/.

[38] Bryan Fink. Distributed computation on dynamo-style distributed storage: riak pipe. In Torben Hoffman and John Hughes, editors, *Erlang Workshop*, pages 43–50. ACM, 2012.

[39] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[40] FoundationDB. https://foundationdb.com/.

[41] Adam Fuchs. Accumulo–Extensions to Google's Bigtable Design, 2012.

[42] RDF Gateway. http://www.intellidimension.com.

[43] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 1 edition, 2011.

[44] HBase. https://hbase.apache.org/.

[45] Hibari. https://github.com/hibari/hibari.

[46] HyperGraphDB. http://www.hypergraphdb.org/index.

[47] HyperTable. http://hypertable.org/.

[48] InfoGrid. http://infogrid.org/trac/.

[49] Apache Jackrabbit. http://jackrabbit.apache.org/.

[50] Jena. http://jena.apache.org/.

[51] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot. A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 2:1–2:10, New York, NY, USA, 2013. ACM.

[52] Mirco Kuhlmann, Lars Hamann, Martin Gogolla, and Fabian Büttner. A benchmark for OCL engine accuracy, determinateness, and efficiency. *Software and System Modeling*, 11(2):165–182, 2012.

[53] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[54] MarkLogic. http://www.marklogic.com/?gclid=CI-apdr-k7wCFeKWtAodOCgANw.

[55] MongoDB. http://www.mongodb.org/.

[56] MongoEMF. https://github.com/BryanHunt/mongo-emf.

[57] Mulgara. http://www.mulgara.org/.

[58] MySQL. http://www.mysql.com.

[59] Neo4EMF. http://neo4emf.com.

[60] ObjectivityDB. http://www.objectivity.com/products/objectivitydb.

[61] OQGRAPH. http://openquery.com.au/node/23.

[62] OrientDB. http://www.orientechnologies.com/orientdb.

[63] PostgreSQL. http://www.postgresql.org/.

[64] RavenDB. http://ravendb.net/.

[65] Redis. http://redis.io/.

[66] Riak. http://basho.com/riak.

[67] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, Fiona A.C. Polack. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Proc. 14th IEEE International Conference on Engineering of Complex Computer Systems*, , Potsdam, Germany, 2009.

[68] Markus Scheidgen, Anatolij Zubow, Joachim Fischer, and Thomas H Kolbe. Automated and transparent model fragmentation for persisting large models. In *Model Driven Engineering Languages and Systems*, pages 102–118. Springer, 2012.

[69] M Seltzer. Oracle nosql database. *Oracle White Paper*, 2011.

[70] SimpleDB. http://aws.amazon.com/simpledb/.

[71] SparkSee. http://sparsity-technologies.com/.

[72] StarDog. http://stardog.com/.

[73] 3 Store. http://sourceforge.net/projects/threestore/.

[74] TinkerPop. Blueprints. https://github.com/tinkerpop/blueprints/wiki.

[75] TitanDB. http://thinkaurelius.github.io/titan.

[76] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for Graph Transformation. In *VL/HCC*, pages 79–88, 2005.

[77] VertexDB. https://github.com/stevedekorte/vertexdb.

[78] Jim Webber. A programmatic introduction to Neo4j. In Gary T. Leavens, editor, *SPLASH*, pages 217–218. ACM, 2012.

[79] WhiteDB. http://whitedb.org/.