A Benchmark for Context Data Management in Mobile Applications*

Nadine Fröhlich Thorsten Möller Steven Rose Heiko Schuldt

Technical Report CS-2010-002
University of Basel
Email: {firstname.lastname}@unibas.ch

Abstract

Over the last few years, computational power, storage capacity, and sensing capabilities of mobile devices have significantly improved. As a consequence, they have undergone a rapid development from pure telecommunication devices to small and ubiquitous computing platforms. Most importantly, these devices are able to host context-aware applications, i.e., applications that are automatically adjusted to the current context of their user. This, in turn, requires sensing support on the device, the possibility to store context information, and to efficiently access this context information for the automated adaptation of applications. In this paper, we introduce a benchmark for context management in mobile context-aware applications. We present in detail the design and setup of the benchmark, based on an eHealth use case. The benchmark evaluation considers context queries on Android Nexus One cell phones and compares the performance of different settings including relational and object-oriented databases on the mobile device, and an RDF triple store on a stationary computer. The results show significant differences in the settings that have been evaluated and are thus valuable indicators for database selection and system design for mobile context-aware applications.

Keywords:
Database benchmark, context-aware applications, mobile applications, eHealth.

*This work has been partly funded by the Hasler Foundation (project *LoCa* – a Location and Context-aware eHealth Infrastructure).

Computer Science Department, Databases and Information Systems Group, Bernoullistrasse 16, CH – 4056 Basel Switzerland. URL: http://dbis.cs.unibas.ch/

1 Introduction

Over the last few years, mobile devices have undergone a rapid metamorphosis from pure telecommunication devices to small and ubiquitous computing platforms. This is the result of a multitude of technical developments: i.) new types of powerful and energy-efficient processors; ii.) the significant increase in local storage capacity due to inexpensive, low latency flash memory cards; and iii.) sophisticated sensing capabilities already embedded into off-the shelf devices (e.g., GPS sensors or acceleration meters). As a result of these developments, mobile devices are more and more in the focus of novel kinds of applications that aim at improving the way users access data and information.

A particular emphasis has been put in recent years on *context-aware applications*. These applications aim at automatically adapting the way information is accessed, processed, and/or presented to the current needs of their user, based on their context (e.g., preferences, location, devices at hand). As users are mobile, their context may rapidly change over time. Hence, in contrast to traditional earmarked applications running on stationary devices with users whose context is rather static, mobile applications are characterized by the intrinsic high frequency in which the users' context changes. For the management of sensed context and the exploitation of context for automated adaptation, e.g., by a rule engine, efficient and effective context data management is needed.

1.1 Motivation: eHealth Use Case

In the following, we will shortly introduce a concrete application scenario which has motivated the design of our benchmark for mobile context-aware applications. Consider, for instance, information access in a hospital where medical experts are usually pinched for time. On a ward round, the doctors' time should mainly be spent for the interaction with their patients, rather than for searching relevant patient data in the hospital's information system. In particular, retrieving electronic patient records should be as unobtrusive and efficient as possible. This can be achieved, for instance, by automatically adapting applications to the user's context, defined by the current location of a physician. When a physician enters a sick room, her mobile phone should automatically display a list of patients in that room. In addition, annotations to the health record added via the device will be synchronized with the underlying clinical information system. In case the physician wants to share medical images with her patient for which the mobile device's display is too small, the images should be automatically transferred to the patient's bed-mounted multimedia device. Thereby, due to the physician's context sensed by the mobile device, this multimedia device in the vicinity is automatically recognized.

1.2 Challenges and Contribution

For the type of dynamic adaptations described before, the current context of a user needs to be sensed and stored for immediate and/or later exploitation. The amount of context data to be stored can vary significantly, depending on the frequency in which the user's context changes and the frequency in which it is sensed. Although today's mobile devices are much more powerful compared to devices as of some years ago, resources are still limited. Therefore, databases for locally storing and retrieving context data need to be as efficient as possible. We have designed a benchmark tailored to context data management for mobile applications (aligned to the eHealth use case presented above) in order to identify which data store is best suited for this task.

This benchmark has been implemented using three different types of open source databases:

1. relational databases locally on the device and on a remote stationary server
2. an RDF triple store on a remote server, and
3. an object-oriented database locally on the device and on a remote stationary server.

We could only run the triple store remotely, with updates and queries submitted from a mobile device, as there is, to the best of our knowledge, currently no stable open source RDF-based implementation directly running on mobile devices. However, this configuration is very relevant, since context information will be subject to reasoning. To factor out communication costs when interacting with a remote server, we also ran the benchmark on the object-database and two relational databases remotely.

For the evaluation we used the context data model designed for the LoCa project [1], implemented for each of the data stores. Furthermore, we implemented a data generator to populate all databases with the same context data. The benchmark queries have been tailored to the different data models and schemas while keeping the original semantics as defined in the benchmark set-up.

This technical report is organized as follows: Section 2 reviews related work. The benchmark setup is discussed in Section 3. The implementation of the benchmark is described in Section 4 and Section 5 presents and discusses the evaluation results. Section 6 concludes. In Appendix A, we present in detail the RDF implementation of our context model. Appendices B–D provide details on the implementation of the benchmark queries in the different schemas, and Appendix E presents an overview of all single query results.

2 Related Work

Existing OLTP and OLAP-style database benchmarks usually address data center-scales rather than mobile environments. The TPC-C and TPC-E Benchmarks [2] give preference to a relational data model, SQL as the query language, and aim at enterprise-scale database systems. Likewise, the Berlin SPARQL Benchmark [3] focusses on RDF triple stores, and the LUBM Benchmark [4] evaluates OWL knowledge base systems, both bound to SPARQL as the query language. The Pole Position [5] benchmark compares relational and object-oriented database engines and object-relational (O/R) mapping implementations, but not in a mobile setting.

In general, benchmarks for mobile environments are rare. There exists an open source benchmark [6] designed for the Android platform which compares the object-oriented database Perst [7] with SQLite. However, the evaluations are too limited to assess the overall performance characteristics of these systems. All queries access only one relation without joins, aggregations, subselects, etc. Therefore, they cannot be directly applied to the kind of schema and queries that have to be considered for context-awareness.

Our benchmark compares the performance of different approaches for storing context data. In [8], context models are compared w.r.t. simplicity and flexibility, while we focus in our work explicitly on performance aspects. [9] identifies requirements on context stores, especially in terms of historical context data, which are considered in our benchmark.

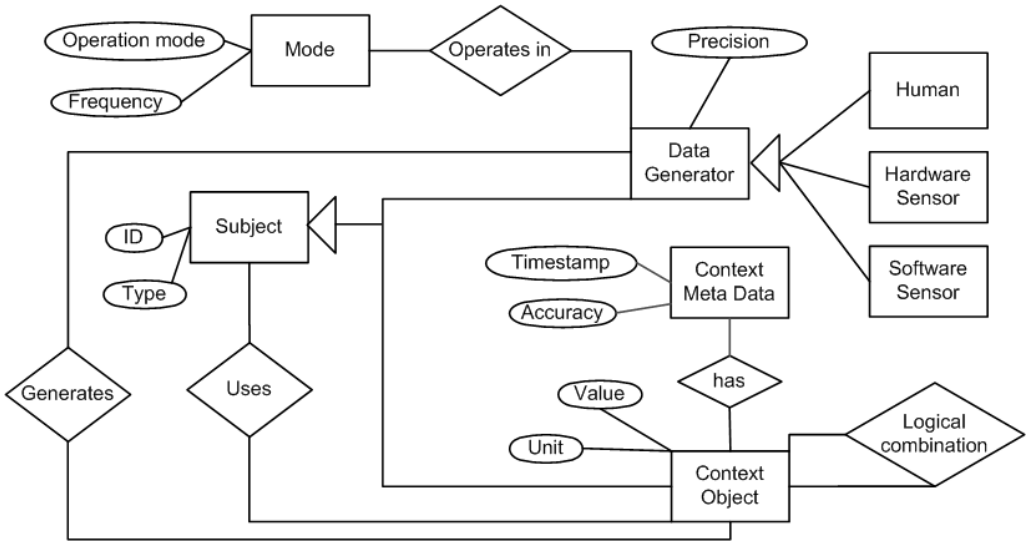


Figure 1: LoCa Context Model

3 Benchmark Design

In what follows, we introduce a detailed description of the data model designed for the benchmark and the query mix to be considered.

3.1 Benchmark Data Model

For the benchmark, we exploit the LoCa data model for context data which is defined in accordance with the most commonly used definition for context (Dey et al. [10]). The LoCa context model is introduced in detail in [1] and summarized in Figure 1 and Table 1.

3.2 Benchmark Test Load

The objective for the specification of the benchmark test load was to be as realistic as possible (in terms of the different types of queries, their mix, and the volumes of data), without giving preference to a very specific application. For this, we have analyzed typical eHealth environments (information access of physicians in the course of a day, based on statistical information of a medium-sized hospital) and have generalized these findings for the specification of our benchmark. Based on these numbers we estimated the amount of context data to be collected in the course of one year (see Table 2, column 3). On a mobile device, only context data of one user are stored so we scaled the data of column 3 down to one person (see Table 2, column 2). In order to make the benchmark as flexible as possible, we consider three settings with context data accumulated over the period of i.) a year, ii.) a quarter, and iii.) a week. In the latter two cases, the numbers from Table 2 are scaled down accordingly. The data for all data stores were generated by one single generator to ensure not only the same work load for all stores but also the exactly same base of test data.

*Cardinality changes over time, i.e., new context objects are created in the course of the benchmark execution.

Element	Description
Subject	The element for which the context is to determine, e.g., a mobile phone respectively its owner.
ContextObject	The context of the subject, e.g., the location of the physician. Context objects have a value, e.g., current GPS coordinates, which have a unit, e.g., degree (for longitude and latitude). Context objects are also subjects.
DataGenerator	The DataGenerator is the sensor which senses the context data. Beside hardware (GPS) and software sensors (diary) we include humans which manually give input to the system. DataGenerators have a precision.Also DataGenerators are Subjects.
Generates	The DataGenerator generates the ContextObjects for a Subject.
Mode	The entity mode hosts meta data of DataGenerators. It describes whether the generator produces a data stream with a special frequency or if it is requested.
ContextMetaData	The ContextMetaData are the meta data to the ContextObjects. Here the timestamp of sensing and the accuracy of the context data is stored.
LogicalCombination	ContextObjects can consist of ContextObjects and this is covered by LogicalCombinations. A GPS metering consists of latitude, longitude,...

Table 1: Description of Elements of the LoCa Context Model

3.3 Benchmark Queries

The benchmark queries we have defined reflect the way mobile users and context-aware applications interact with a context database. The query mix considers mainly read accesses to context data, as well as insert operations which create new context objects as the user's context evolves.

Q1: Return a subject by a given ID. Such queries are often used, for instance for the identification of a physician.

Q2: Return the last recorded context object of a given type for a subject, e.g., the last blood pressure value.

Entity	Data Sets per Person & Year	Data Sets Medium-Size Hospital / Year
DataGenerator	851	15,500
Human	1	5,000
HardwareSensor	833	25,000
SoftwareSensor	17	500
Mode	851	5,000
ContextObject	185,000*	925,000,000*
ContextMetaData	185,000*	925,000,000*
LogicalCombination	185*	925,000*
Subject	1,800	27,000 (patients)

Table 2: Cardinalities of Entities in Benchmark

Query	Percentage
Query 1	10 %
Query 2	7 %
Query 3	7 %
Query 4	6 %
Query 5	3 %
Query 6	13 %
Query 7	7 %
Query 8	6 %
Query 9	4 %
Query 10	36 %
Query 11	0.5 %
Query 12	0.5 %

Table 3: Percentage of Queries in the Benchmark Query Mix

Q3: Return the context object of a given type for a subject in a given time interval (day, week, month).

Q4: Return the last recorded context object of a given type for a subject, generated by a given generator. It considers details on the sensor, e.g., to find out whether a blood pressure meter has the desired precision.

Q5: Return the available data generators including type and precision that generate context objects of a given type for a subject (e.g., for the selection of a generator for a particular application).

Q6: Return a subject of a given type with the same context object as a given subject, for instance when searching for patients belonging to a sick room or for devices available in the sick room in which the physician is currently situated.

Q7: Return all available types of context objects to a given subject in alphabetical order.

Q8: Return all logical combinations of context data for a given subject. The query result shows which context data are part of other context data.

Q9: Return the number of data generators that generate context data per subject. By this query one is able to control the generators belonging to one subject.

Q10: Insert a context object. This operation reflects all context change of a user, as sensed by the device.

Q11: Update all context objects belonging to one logical combination.

Q12: Delete a context object, e.g., for correcting mistakes of human data generators.

The order of the queries in the mix will be randomly chosen, but their occurrence over a longer interval is determined by the percentages given in Table 3. It should be noted that all queries, except for no. 10 (creation of context object) run sequentially, while the insertion is done automatically in the background, in parallel to the dynamic adaptation (e.g., the execution of the other benchmark queries).

3.4 Performance Metrics

For the benchmark evaluation, we consider the following metrics, which are aligned with the Berlin benchmark [3].

3.4.1 Metrics for Single Queries

- *Average Query Execution Time (aQET$_x$)*: Average time for executing an individual query of type x ten times with different parameters against the system under test (SUT).
- *Minimum/maximum Query Execution Time (minQET$_x$, maxQET$_x$)*: A lower and upper bound execution time for queries of type x.
- *Queries per Second (QpS$_x$)*: Average amount of queries of type x that were executed per second. This value is computed from the aQET$_x$ values.

3.4.2 Metrics for Query Mixes

Overall Runtime (oaRT): Overall time it took the test driver to execute a certain amount of queries following the distribution in the mix against the SUT. Thereby, inserts are running in a parallel thread on the device. We decided to process thee runs each with 300 queries each.

4 Benchmark Setup

In this section, we describe in detail the setup of our benchmark evaluation. We especially focus on the implementation of model and queries for data stores of the three chosen paradigms.

4.1 Data Stores

We have selected three different kinds of data stores for the benchmark. All are well tested and stable open source systems that are widely adopted in practice.

- *Relational databases*
  i) H2 (v. 1.2.136) in embedded mode and remotely
  ii) MySQL (v. 5.0.51a) running remotely
  iii) SQLite (version 3.5.9) [11] in embedded mode
- *RDF/OWL triple store* Sesame (v. 2.3.1) [12] with Storage and Inference Layer SwiftOWLIM (3.0 beta 12) [13] remotely (servlet, deployed into apache-tomcat-6.0.24).
- *Object-oriented database* db4o (v. 7.12) [14] in embedded mode and running remotely

Initially, we have chosen only SQLite, Sesame, and db4o for the benchmark. However, Sesame/SwiftOWLIM currently does not run on smart phones and SQLite cannot be used in a client/server (c/s) mode. Therefore, we added H2 to our benchmark as it supports both embedded and c/s mode. This makes results for remote operation comparable among the triple store and the relational databases. Finally, we also evaluated MySQL in c/s mode.

The relational database SQLite demands zero configuration, has a small system footprint and no external dependencies to other libraries which makes it highly appropriate for mobile devices.

Furthermore, it is currently used in several applications (e.g., Firefox, Google tools). H2 [15] is an open source Java database also having a small system footprint (about 1 MB) and low memory requirements. It supports disk-based and in-memory databases in embedded and server mode. MySQL [16] is a highly popular and very widely adopted open source database, featuring a rich set of functionality.

As the LoCa approach considers the semantics of services when deciding on dynamic adaptations, we have also chosen a triple store for the benchmark. Out of the available open source RDF triple stores we have evaluated, Sesame has been the most promising tool as it is already widely used [12], actively maintained and has proven to perform well enough for our purpose [3]. Sesame allows for easy extension via the so-called SAIL-API. As the back-end store implementations shipped with Sesame do not provide the OWL based reasoning we would like to use in our model, we used an alternative third party implementation called SwiftOWLIM, an in-memory triple store which implements rule-based forward-chaining strategy for inferencing and supports a subset of OWL DL.

The object-oriented database db4o has been chosen to avoid the impedance mismatch as the LoCa system is completely implemented in Java.

All tests were performed on Nexus One (N1) mobile phones using Google Android version 2.1 [17] platform. The devices come with 512MB RAM, 512MB ROM, and a Qualcomm QSD8250 CPU with 1GHz. When working in remote mode, instances of Db4o, MySQL, and Sesame/SwiftOWLIM ran on a standard server (Intel Core 2 – 6600 2.4GHz, 4GB RAM, 250GB SATA Seagate ST3250820AS, Ubuntu 9.04 x86_64). Mobile phones were connected to a 802.11b/g 54MBit wireless access point (DLINK DIR615), to which the server was connected via 100MBit Ethernet.

4.2 Data Generator

To supply the different types of databases with equivalent datasets, we implemented a parameterizable data generator that incorporates output modules for every target paradigm. In the first step, the raw data is generated in-memory using an integrated object-oriented data model according to the context data model presented in Figure 1. The actual data values are generated by exchangeable value generators, all relations are picked at random (evenly distributed). In the second step, this data is then transformed into the target data formats by special output modules. An intermediate raw format is stored to be able to create additional outputs later, that are equivalent to the formerly generated ones. Additionally, every output module creates equal sets of query parameter values that are used throughout the benchmark run. This ensures a maximum of comparability between the different platforms.

4.3 Implementation of the Data Model

The schemas for the different data stores have been created according to the LoCa context model. The following subsections present the three different schemas in more detail.

4.3.1 Relational Schema

For the relational databases, a standard transformation has been applied. The inheritance is implemented using a horizontal partitioning to avoid joins and decrease execution time. Figure 2 shows the context model mapped to the relational schema.

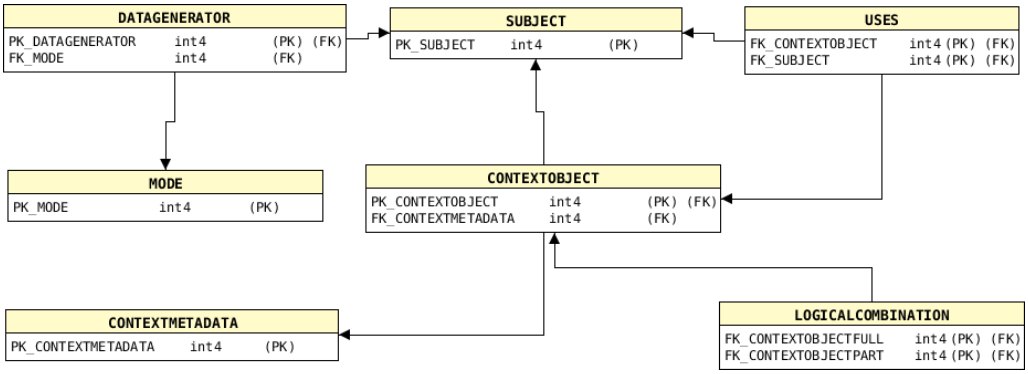


Figure 2: LoCa Context Model as Relational Schema

4.3.2 Triple Store Schema

For the transformation to RDF, an OWL DL ontology has been defined[1]. Entities are directly mapped to concepts (classes), relations among entities are mapped to either one or two object roles (properties) depending on whether both directions can be navigated, and attributes to data roles. Furthermore, we exploited additional modeling expressivity available in OWL: cardinality restriction constructors, disjointness of concepts, and functional, transitive, and irreflexive properties whenever appropriate. Figure 3 shows a visualisation with gruff [18]. A description of the triples in a terse form can be found in Appendix A. The used format is called Turtle (Terse RDF Triple Language) [19]. The syntax is written in a compact and natural text form.

4.3.3 Object-Oriented Schema

For db4o, we transformed the context model to a Java class structure by mapping entities to classes and by using collections for the relationships. To work in an object oriented manner, we applied the composite pattern to implement the inheritance hierarchy. Figure 4 shows the object oriented implementation of our context model in UML.

4.4 Implementation of Benchmark Queries

All benchmark queries have been formulated in the query languages supported by the respective data stores. For relational databases we used SQL, for Sesame SPARQL, and for db4o SODA. Appendices B, C, and D show the implementation details for all benchmark queries.

4.4.1 Query Implementation for Relational Databases

In case of H2 and MySQL, we used suitable JDBC drivers to access the databases. SQLite comes as built-in embedded database with its own API that offers different ways to access the database, raw query (nearly plain SQL), and a structured interface for users with little SQL knowledge. For the benchmark, we used the raw query interface. Appendix B on page 19 shows the implementation details of the queries in SQL.

[1] Available via `http://on.cs.unibas.ch/owl/1.0/Context.owl`

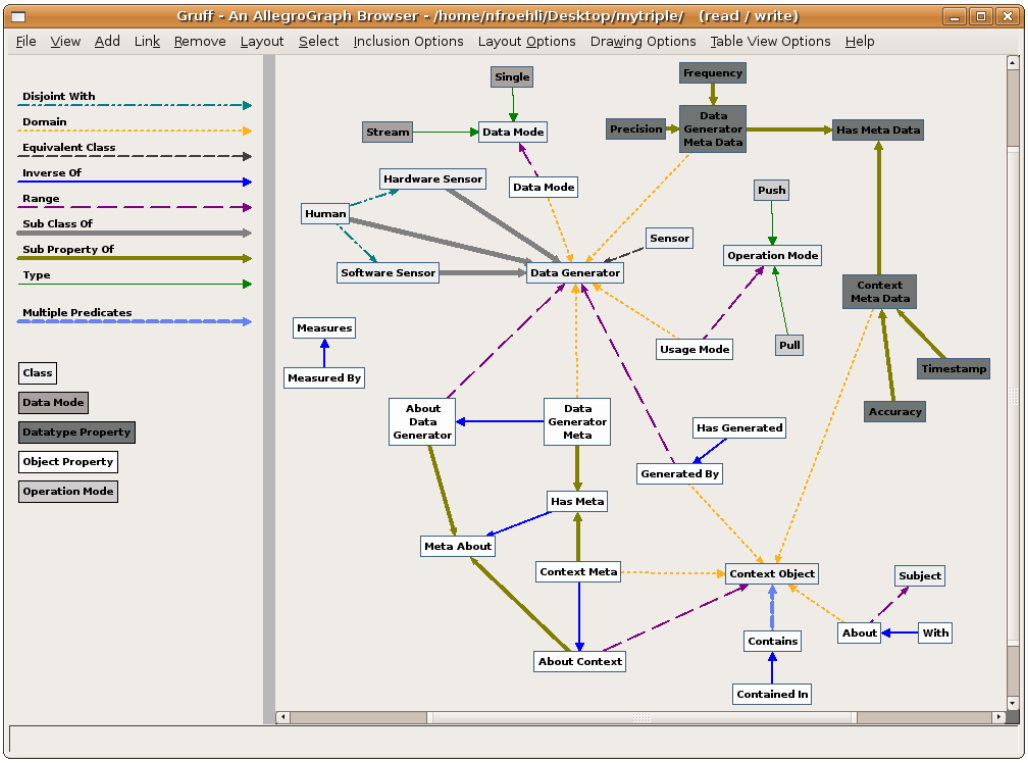


Figure 3: LoCa Context Model as OWL Graph

4.4.2 Query Implementation for Triple Stores

SPARQL has evolved as the de facto standard for querying RDF graph data. At the moment, there exist two major versions of SPARQL that are specified by the W3C. SPARQL 1.0 [20] has matured to a W3C recommendation and contains basic graph query constructs. At the time of writing SPARQL 1.1 [21] is still a working draft. It adds more advanced features like aggregations and data manipulation to the language. As our chosen triple-store only supports SPARQL 1.0, queries that change the data set were difficult to handle. While query Q12 can be easily implemented using a Sesame specific type of request ('transaction'), query Q11 would have required to programmatically perform needed updates on the data sets. Thus, we chose to drop this query in the triple-store implementation of our benchmark. Also aggregating queries had to be implemented partly on the client side. To access the RESTful SPARQL interface that SESAME provides, we used the HttpClient implementation integrated into the Android SDK. All result data was encoded as JSON and parsed using androids own parser implementation. In Appendix C on page 22 the implementation details of the SPARQL queries are shown.

4.4.3 Query Implementation for Object Oriented Databases

Db4o supports at its APIs query by example (QbE), Native Queries (NQ) and SODA. The reference documentation [22] recommends the use of NQ as it is easier to use than SODA and not limited in functionality as QbE. But we decided to use SODA because it is, according to the db4o documentation [22], up to two times faster than optimized NQ and five times faster than

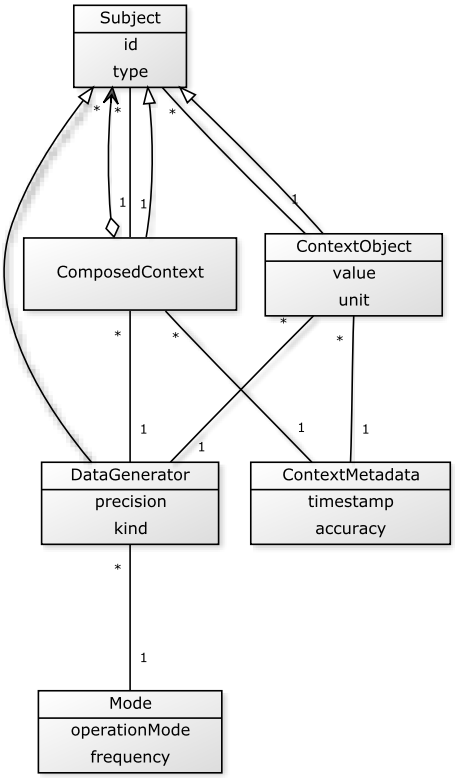


Figure 4: LoCa Context Model in UML

unoptimized NQ — the reason for this is that SODA is the underlying internal query API all the other query APIs are mapped to. Appendix D on page 26 shows the implementation details of the queries in SQL. In Appendix C on page 22 describes the implementation details of the SODA queries.

5 Benchmark Results

This section summarizes the evaluation results of our benchmark.

5.1 Query Mixes

Figure 5 and Table 4 show the results of the query mix evaluation in the different settings. The execution times of the relational databases vary considerably but except for SQLite, all query mixes could be processed. Only for H2 emb, we had to increase the JVM heap size to 80 MB which is however a rather unrealistic setting for the device. SQLite was comparably slow. H2 c/s and MySQL c/s manage to execute the query mix for one year in nearly the same time as SQLite needs for the query mix for one week (SQLite week: 386s, H2 year: 251s, MySQL year: 417s). This is most probably due to the lack of a sophisticated query optimizer for SQLite. Furthermore, H2 c/s proves to better perform on the largest data set (year), but MySQL c/s performs much better on the smaller data sets (week/quarter). As expected, the c/s architectures perform better than the embedded databases on our resource limited mobile devices. For SQLite, we have compared

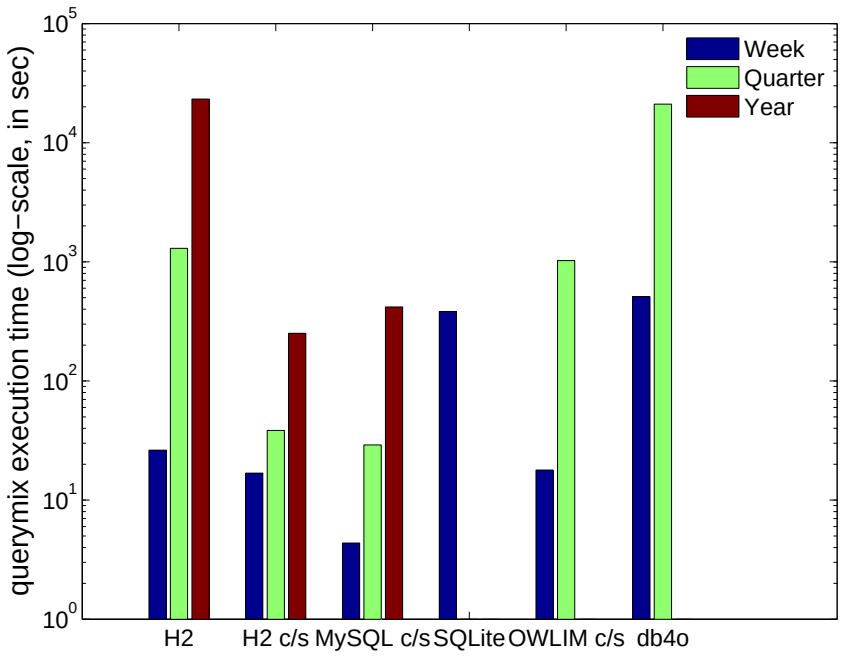


Figure 5: Overview of Query Mixes

System	Week	Quarter	Year
H2 emb	26 (0.9)	1,300 (23.8)	23,251 (267.3)
H2 c/s	17 (0.9)	38 (23.8)	251 (267.3)
MySQL c/s	4 (0.8)	29 (18.8)	418 (186.0)
SQLite emb	386 (0.8)	– (17.5)	– (172.9)
SQLite emb (r)	384 (0.8)	– (17.5)	– (172.9)
OWLIM c/s	18 (1.6)	1,025 (38.1)	– (384.1)
db4o emb	511 (1.5)	21,100 (22.3)	– (126.5)
db4o c/s	– (1.5)	– (22.3)	– (126.5)

Table 4: Execution Times of Query Mixes in Seconds (in parentheses: Database Sizes in MB)

a setting with referential integrity with a setting without. The benchmarks results show only marginal differences. OWLIM c/s shows only an average performance, only slightly better than H2 emb – however, it is not able to process a query mix on the large data set (year). For db4o we could not manage to process a complete query mix. With db4o c/s, query Q4 could not be executed although the query ran fine in the db4o embedded version – this seems to be the result of a mashaling/unmarshaling problem in db4o. The db4o emb version is slow, and needs a lot of space and JVM heap.

5.2 Single Queries

To find out why the query mixes showed significant performance variations on different systems, we have also analyzed the average execution time of frequently occurring queries. Query Q6 has a fraction of 13% in the query mix. For this query, the triple store performs much better than most relational databases and even db4o c/s is for the large dataset (year) in the range of MySQL (see

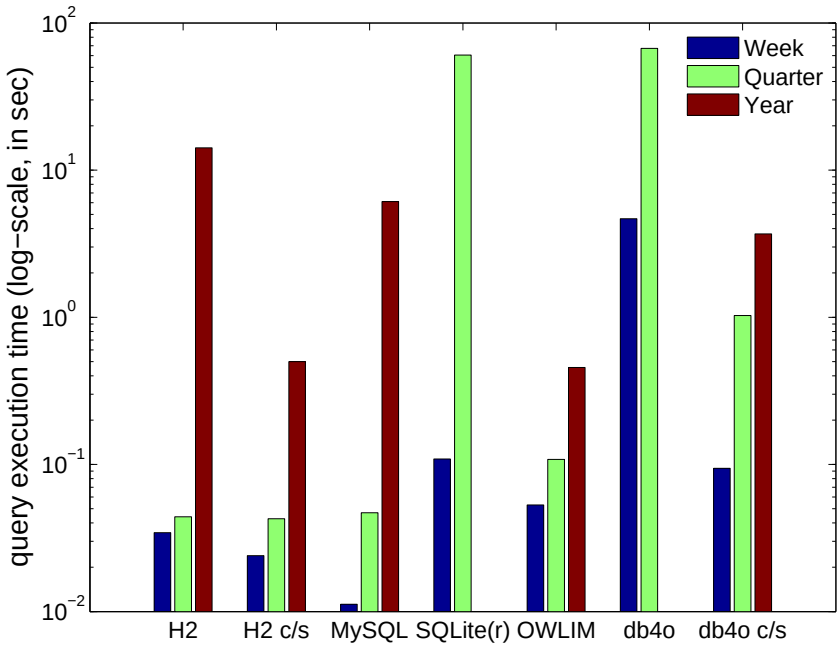


Figure 6: Single Query Evaluation for Query 6

Figure6).

Query Q1 has also a major influence on the query mix (10%). SQLite performed best for this query while db4o emb performed comparably bad. db4o c/s performed better but could not reach the times of the relational databases; its performance is comparable to OWLIM (see Figure 7).

Query Q9 is one of the most controversial queries in the benchmark since some systems have not been able to properly execute it in reasonable time (see Figure 8). This query runs fine on H2 c/s and MySQL but lasted extremely long on SQLite. One reason for this is might be the less elaborated optimizer of SQLite. H2 could perform this query but we needed to increase the JVM heap to 64 MB. OWLIM performs fair on small data sets but gets very slow with increased data sets.

An overview of the query execution times of all queries (Q1 to Q12) can be found in Appendix E on page 30.

5.3 Summary of Results

The analysis of the performance for individual queries shows that the chosen systems have different strengths. A system that is very slow for one query can be fast for another query. For deciding what system is best we have to care about the overall performance shown in the query mixes where the fraction of queries is considered. In the mixes, read-only queries (without Query Q10) are running in parallel to inserts (Query Q10). The execution of inserts usually lasts a few milliseconds (2 to 35 ms), except for OWLIM where the execution of inserts lasts more than 10,000 milliseconds. The performance in c/s mode was mostly considerably better than in embedded mode. H2 needed for the query mix with the context data accumulated in one year 23,251 s while H2 c/s needed 251 s. For db4o we could not run a complete query mix. But when comparing the single queries it is obvious that db4o c/s is mostly faster than db4o embedded. Query Q9 seems to be an outlier

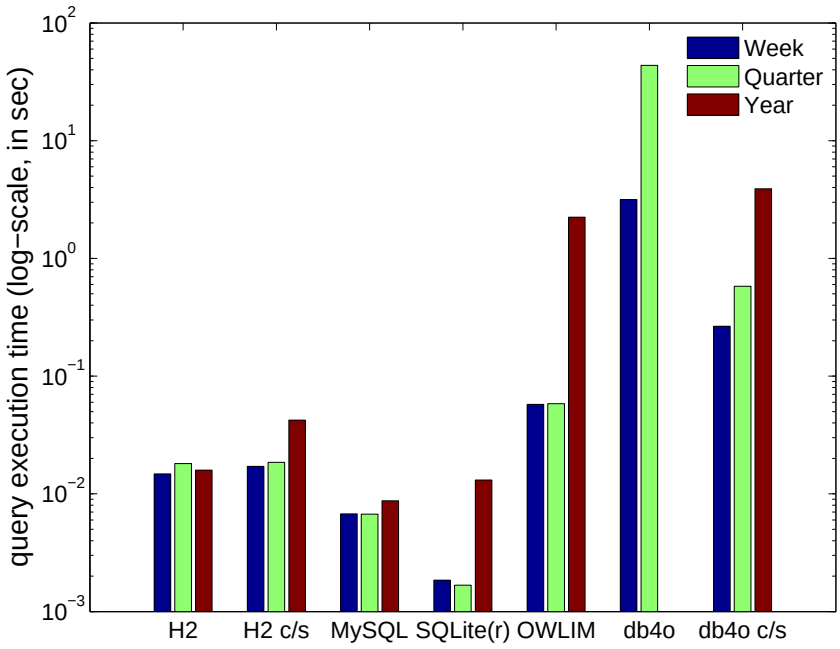


Figure 7: Single Query Evaluation for Query 1

as it performs better when executed in embedded mode than in c/s mode (see Figure 8). Although additional communication cost incur in c/s mode, powerful server-side hardware can partly or completely compensate this. Only for db4o, communication costs are in some cases higher than the benefit of more powerful servers. Finally, the benchmark shows that it is not feasible to store context data for a longer period in an embedded database on a mobile device.

5.4 Lessons Learned

In this section, we summarize the lessons learned, with a particular focus on usability and expressiveness of the systems we considered in our benchmark.

SQLite is small and easy to use, but its optimizer is not as developed as that of more mature database systems. We had to manually optimize some of the queries to improve the query performance.

H2 supports referential integrity by default and works well in embedded and client/server setting. Most notable, H2 caused least configuration problems.

MySQL the data store with the richest set of features, is also the most demanding system regarding configuration for remote access.

OWLIM Sesame's built-in data store can be easily created and maintained using the web based workbench. However, manual configuration was needed to create repositories working with an OWLIM store after the binaries were integrated into the Servlet distribution. As we chose the RESTful HTTP interface to access the stores, some conveniences of common database access layers had to be implemented on top of the standard http client, e.g., to circumvent

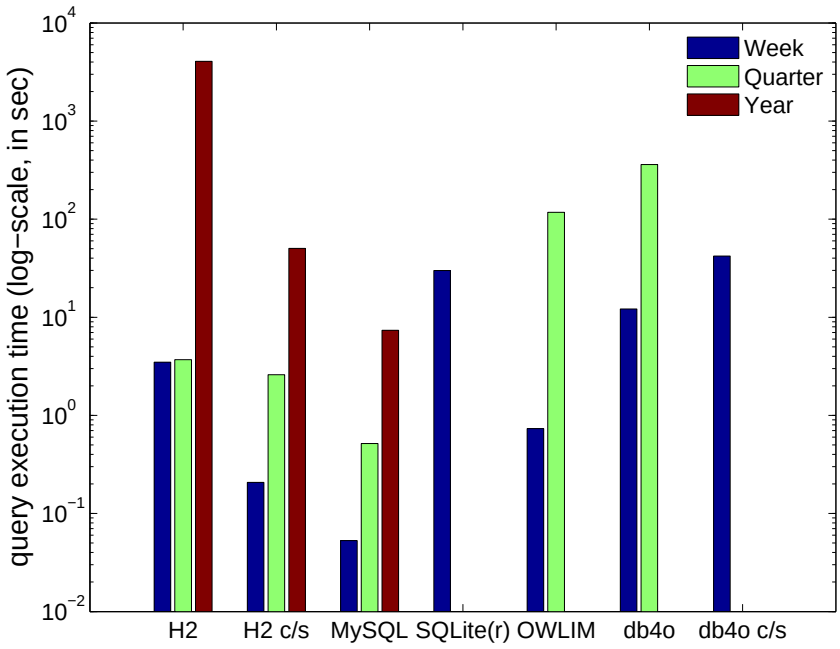


Figure 8: Single Query Evaluation for Query 9

memory restrictions when large result sets were received. This was the case for Q8, where a lacking aggregation feature of SPARQL 1.0 had to be implemented on the client side.

db4o was comparatively slow, especially for Q9 and it demanded much memory and JVM heap space (up to 64M/80M). For speeding up the tests, we used the SODA query language that is much faster than the recommended query interface (NQ). Actually, the performance of db4o also strongly depends on the design of the data model as inheritance and collections with a lot of associated objects slow down the query execution dramatically. We optimized our model, but in order to reach an appropriate speed we would have to completely redesign our model, especially by abandoning the object-oriented design (inheritance). In c/s mode, one query (Q3), did not run at all, although we did not experience any problems in embedded mode. Although db4o performs well on a standard PC in c/s mode, we were faced with major problems regarding performance and stability in embedded mode on the mobile device. These problems are well taken by the db4o developers as they are currently aiming at decreasing the needed stack size to better support Android-based platforms.

6 Conclusions

Managing context data on mobile devices is an essential prerequisite for supporting dynamic, context-aware adaptations of applications. In the paper, we have presented a benchmark designed for the management of context data and we have reported in detail on the benchmark evaluation which considers relational and object-oriented databases and a triple store in embedded and/or client/server mode. The set-up considers different context data sets accumulated by mobile users on their devices in a realistic setting in the course of a week, a month, and a year. Relational data

stores in c/s mode performed best in the benchmark. For smaller data sets, the performance of embedded relational databases and the triple store (c/s) is sufficient.

References

[1] Nadine Fröhlich, Andreas Meier, Thorsten Möller, Marco Savini, Heiko Schuldt, and Joël Vogt. LoCa – Towards a Context-aware Infrastructure for eHealth Applications. In *Proc. of the 15th Int'l Conference on Distributed Multimedia Systems (DMS'09)*, 2009.

[2] TPC - Transaction Processing Performance Council. http://www.tpc.org/.

[3] Berlin SPARQL Benchmark (BSBM), 2009. http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark.

[4] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, October 2005.

[5] PolePosition – Open Source Database Benchmark. http://polepos.sourceforge.net/.

[6] McObject Benchmarks Embedded Databases on Android Smartphone. http://www.mcobject.com/march9/2009.

[7] Perst – An Open Source, Object-oriented Embedded Database. http://www.mcobject.com/perst/.

[8] Thomas Strang and Claudia Linnhoff-Popien. A context modeling survey. In *In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004, Nottingham/England*, 2004.

[9] Matthias Baldauf and Schahram Dustdar. A survey on context-aware systems. *Int. Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, June 2007.

[10] A. K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, February 2001.

[11] SQLite. http://www.sqlite.org/.

[12] Sesame – Open Source Framework for Storage, Inferencing and Querying of RDF Data. http://www.openrdf.org/.

[13] OWLIM. http://www.ontotext.com/owlim/.

[14] db4o. http://www.db4o.com/.

[15] H2 Database Engine. http://www.h2database.com/html/main.html.

[16] MySql. http://mysql.com/.

[17] Android Open Source Project. http://source.android.com/.

[18] Gruff. http://www.franz.com/agraph/gruff/.

[19] Turtle - Terse RDF Triple Language. http://www.w3.org/TeamSubmission/turtle/.

[20] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

[21] SPARQL 1.1 Query Language. http://www.w3.org/ TR/2010/WD-sparql11-query-20100601/.

[22] db4o Reference. http://developer.db4o.com/Documentation/Reference/ db4o-7.12/java/reference/.

A Appendix - Context model in Turtle Syntax

In this section, we present our context model in Turtle, a textual syntax for RDF. This syntax is written in a compact and natural text form and includes abbreviations for common usage patterns and data types [19].

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://on.cs.unibas.ch/owl/1.0/Context.owl#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@base <http://on.cs.unibas.ch/owl/1.0/Context.owl> .

#################################################################
#    Object Properties
#################################################################

:about rdf:type owl:ObjectProperty ;
       rdfs:domain :ContextObject ;
       rdfs:range :Subject .

:aboutContext rdf:type owl:ObjectProperty ;
              rdfs:range :ContextObject ;
              rdfs:subPropertyOf :metaAbout .

:aboutDataGenerator rdf:type owl:ObjectProperty ;
                    rdfs:range :DataGenerator ;
                    rdfs:subPropertyOf :metaAbout .

:containedIn rdf:type owl:ObjectProperty ;
             owl:inverseOf :contains .

:contains rdf:type owl:IrreflexiveProperty ,
                   owl:ObjectProperty ,
                   owl:TransitiveProperty ;
          rdfs:range :ContextObject ;
          rdfs:domain :ContextObject .

:contextMeta rdf:type owl:ObjectProperty ;
             rdfs:domain :ContextObject ;
             owl:inverseOf :aboutContext ;
             rdfs:subPropertyOf :hasMeta .

:dataGeneratorMeta rdf:type owl:ObjectProperty ;
                   rdfs:domain :DataGenerator ;
                   owl:inverseOf :aboutDataGenerator ;
                   rdfs:subPropertyOf :hasMeta .

:dataMode rdf:type owl:ObjectProperty ;
          rdfs:domain :DataGenerator ;
```

```
          rdfs:range :DataMode .

:generatedBy rdf:type owl:FunctionalProperty ,
                      owl:ObjectProperty ;
             rdfs:domain :ContextObject ;
             rdfs:range :DataGenerator .

:hasGenerated rdf:type owl:ObjectProperty ;
              owl:inverseOf :generatedBy .

:hasMeta rdf:type owl:ObjectProperty ;
         owl:inverseOf :metaAbout .

:measuredBy rdf:type owl:ObjectProperty ;
            owl:inverseOf :measures ;
            owl:propertyChainAxiom ( :with :generatedBy ) .

:measures rdf:type owl:ObjectProperty ;
          owl:propertyChainAxiom ( :hasGenerated :about ) .

:metaAbout rdf:type owl:ObjectProperty ;

:usageMode rdf:type owl:ObjectProperty ;
           rdfs:domain :DataGenerator ;
           rdfs:range :OperationMode .

:with rdf:type owl:ObjectProperty ;
      owl:inverseOf :about .


###################################################################
#    Data properties
###################################################################

:accuracy rdf:type owl:DatatypeProperty ;
          rdfs:subPropertyOf :contextMetaData ;
          rdfs:range xsd:nonNegativeInteger .

:contextMetaData rdf:type owl:DatatypeProperty ;
                 rdfs:domain :ContextObject ;
                 rdfs:subPropertyOf :hasMetaData .

:dataGeneratorMetaData rdf:type owl:DatatypeProperty ;
                       rdfs:domain :DataGenerator ;
                       rdfs:subPropertyOf :hasMetaData .

:frequency rdf:type owl:DatatypeProperty ;
           rdfs:subPropertyOf :dataGeneratorMetaData ;
           rdfs:range xsd:float .

:hasMetaData rdf:type owl:DatatypeProperty ;
```

```
:precision rdf:type owl:DatatypeProperty ;
           rdfs:subPropertyOf :dataGeneratorMetaData ;
           rdfs:range xsd:nonNegativeInteger .

:timestamp rdf:type owl:DatatypeProperty ;
           rdfs:subPropertyOf :contextMetaData ;
           rdfs:range xsd:dateTime .


#################################################################
#    Classes
#################################################################

:ContextObject rdf:type owl:Class ;

:DataGenerator rdf:type owl:Class ;
               owl:equivalentClass :Sensor ;

:DataMode rdf:type owl:Class ;
          owl:equivalentClass [ rdf:type owl:Class ;
                                owl:oneOf ( :Stream :Single )
                              ] ;

:HardwareSensor rdf:type owl:Class ;
                rdfs:subClassOf :DataGenerator ;
                owl:disjointWith :Human ;

:Human rdf:type owl:Class ;
       rdfs:subClassOf :DataGenerator ;
       owl:disjointWith :SoftwareSensor ;

:OperationMode rdf:type owl:Class ;
               owl:equivalentClass [ rdf:type owl:Class ;
                                     owl:oneOf ( :Push :Pull )
                                   ] ;

:Sensor rdf:type owl:Class ;

:SoftwareSensor rdf:type owl:Class ;
                rdfs:subClassOf :DataGenerator ;

:Subject rdf:type owl:Class ;


#################################################################
#    Individuals
#################################################################

:Pull rdf:type :OperationMode , owl:NamedIndividual .
:Push rdf:type :OperationMode , owl:NamedIndividual .

:Single rdf:type :DataMode , owl:NamedIndividual .
```

```
:Stream rdf:type :DataMode , owl:NamedIndividual .
```

B Appendix - Queries in SQL

This section shows the implemention of the benchmark queries in SQL. In the queries shown here the parameters are marked by args[i]. We took the same parameters for each query independent of the used query language.

Query 1:

```
SELECT pk_subject
FROM subject
WHERE pk_subject = args[0];
```

Query 2:

```
SELECT distinct o.pk_contextobject, o.type, o.contextvalue, o.unit, m.timestamp
FROM contextobject o
     JOIN contextmetadata m ON o.fk_contextmetadata = m.pk_contextmetadata
     JOIN uses u on o.pk_contextobject = u.fk_contextobject
WHERE u.fk_subject = args[1]
      o.type=args[0]
ORDER BY timestamp desc
LIMIT 1;
```

Query 3:

```
SELECT distinct o.pk_contextobject, o.type, o.contextvalue, o.unit, m.timestamp
FROM contextobject o, contextmetadata m, uses u
WHERE o.fk_contextmetadata = m.pk_contextmetadata AND
      o.type= args[0] AND
      o.pk_contextobject = u.fk_contextobject AND
      u.fk_subject = args[1] AND
      m.timestamp > args[2] AND
      m.timestamp < args[3]
ORDER BY timestamp desc;
```

Query 4:

```
SELECT distinct o.pk_contextobject, o.type, o.contextvalue, o.unit, m.timestamp
FROM contextobject o, contextmetadata m, generates g, uses u, datagenerator d
WHERE o.fk_contextmetadata = m.pk_contextmetadata AND
      o.type = args[0] AND
      o.pk_contextobject = u.fk_contextobject AND
      o.pk_contextobject = g.fk_contextobject AND
      g.fk_datagenerator = d.pk_datagenerator AND
      d.pk_datagenerator = args[2]AND
      u.fk_subject = args[1]
ORDER BY timestamp desc
LIMIT 1;
```

Query 5:

```
SELECT distinct d.pk_datagenerator, d.type, d.precision
FROM contextobject o JOIN uses u ON o.pk_contextobject = u.fk_contextobject
     JOIN subject s ON s.pk_subject = u.fk_subject
     JOIN generates g ON o.pk_contextobject = g.fk_contextobject
     JOIN datagenerator d ON g.fk_datagenerator = d.pk_datagenerator
WHERE s.pk_subject = args[1] AND
      o.type = args[0];
```

Query 6:

```
SELECT distinct s.pk_subject
FROM subject s, uses u1, uses u2, contextobject o
WHERE s.pk_subject = u1.fk_subject AND
      o.pk_contextobject = u1.fk_contextobject AND
      o.pk_contextobject = u2.fk_contextobject AND
      u2.fk_subject <> u1.fk_subject AND
      u2.fk_subject= args[0] AND
      s.type = args[1];
```

Query 7:

```
SELECT DISTINCT o.type
FROM subject s JOIN uses u ON s.pk_subject = u.fk_subject
     JOIN contextobject o ON o.pk_contextobject = u.fk_contextobject
WHERE s.pk_subject = args[0]
ORDER BY o.type asc;
```

Query 8:

```
SELECT u.fk_subject, o.pk_contextobject, o.unit, o.contextvalue,
       o1.pk_contextobject, o1.unit, o1.contextvalue
FROM logicalcombination l, contextobject o, contextobject o1, uses u
WHERE l.fk_contextobjectfull= o.pk_contextobject AND
      l.fk_contextobjectpart= o1.pk_contextobject AND
      o.pk_contextobject = u.fk_contextobject AND
      u.fk_subject = args[0];
```

Query 9:

```
SELECT COUNT(distinct g.fk_datagenerator), pk_subject
FROM subject s, uses u, contextobject o, generates g
WHERE s.pk_subject = u.fk_subject AND
      o.pk_contextobject = u.fk_contextobject AND
      g.fk_contextobject = o.pk_contextobject
GROUP BY pk_subject;
```

Query 10:

```
INSERT INTO contextobject (pk_contextobject, contextvalue, type, unit)
VALUES (args[0], args[1], args[2], '');
```

Query 11:

```
CREATE TABLE tmp AS
SELECT o1.pk_contextobject
FROM contextobject o, contextobject o1, logicalcombination l
WHERE l.fk_contextobjectfull = o.pk_contextobject AND
      l.fk_contextobjectpart = o1.pk_contextobject AND
      l.fk_contextobjectfull = args[0];

UPDATE contextobject
SET contextvalue = args[1]
WHERE pk_contextobject IN
      (SELECT pk_contextobject
       FROM tmp);
```

Query 12:

```
DELETE FROM contextobject
WHERE pk_contextobject = args[0];
```

C Appendix - Queries in SPARQL

This section shows the implemention of the benchmark queries in SPARQL. In the queries shown here the parameters are marked by args[i]. We took the same parameters for each query independent of the used query language.

Namespace Prefixes:

```
PREFIX rdf:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdfs:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX context:<http://localhost/owl/Context.owl#>
PREFIX dsi:<http://localhost/owl/DataSetInstances.xml#>
PREFIX dsc:<http://localhost/owl/DataSetClasses.owl#>
```

Query 1:

```
SELECT ?type
WHERE {
  dsi:args[0] rdf:type ?type .
}
```

Query 2:

```
SELECT ?contextObject ?contextType ?contextUnit
       ?contextValue ?accuracy ?timestamp ?datagenerator
WHERE {
  ?contextObject rdf:type             ?contextType .
  ?contextType   rdfs:subClassOf      context:ContextObject .
  ?contextObject context:generatedBy  ?datagenerator .
  ?contextObject context:about        dsi:args[1] .
  ?contextObject rdf:type             dsi:args[0] .
  ?contextObject context:accuracy     ?accuracy .
  ?contextObject context:timestamp    ?timestamp .
  ?contextObject context:unit         ?contextUnit .
  ?contextObject context:contextValue ?contextValue .
}

ORDER BY DESC(?timestamp)
LIMIT 1
```

Query 3:

```
SELECT ?contextObject ?contextType ?contextUnit
       ?contextValue ?accuracy ?timestamp ?datagenerator
WHERE {
  ?contextObject rdf:type             ?contextType .
  ?contextType   rdfs:subClassOf      context:ContextObject .
  ?contextObject context:generatedBy  ?datagenerator .
  ?contextObject context:about        dsi:args[1] .
  ?contextObject rdf:type             dsi:args[0] .
  ?contextObject context:accuracy     ?accuracy .
  ?contextObject context:timestamp    ?timestamp .
```

```
  ?contextObject context:unit         ?contextUnit .
  ?contextObject context:contextValue ?contextValue .

  FILTER(xsd:dateTime(?timestamp) > xsd:dateTime( args[2] )
      && xsd:dateTime(?timestamp) < xsd:dateTime( args[3] )) .
}
ORDER BY DESC(?timestamp)
```

Query 4:

```
SELECT ?contextObject ?contextType ?contextUnit
       ?contextValue ?accuracy ?timestamp ?datagenerator
WHERE {
  ?contextObject rdf:type             ?contextType .
  ?contextType   rdfs:subClassOf      context:ContextObject .
  ?contextObject context:generatedBy  dsi:args[2]  .
  ?contextObject context:about        dsi:args[1] .
  ?contextObject rdf:type             dsi:args[0] .
  ?contextObject context:accuracy     ?accuracy .
  ?contextObject context:timestamp    ?timestamp .
  ?contextObject context:unit         ?contextUnit .
  ?contextObject context:contextValue ?contextValue .
}
ORDER BY DESC(?timestamp)
LIMIT 1
```

Query 5:

```
SELECT DISTINCT ?datagenerator ?precision ?frequency ?usageMode ?dataMode ?type
WHERE {
  ?datagenerator context:measures    dsi:args[1].
  _:tmp_co       context:generatedBy ?datagenerator .
  _:tmp_co       rdf:type            dsi:args[0].
  ?datagenerator context:precision   ?precision .
  ?datagenerator context:frequency   ?frequency .
  ?datagenerator context:usageMode   ?usageMode .
  ?datagenerator context:dataMode    ?dataMode .
  ?datagenerator rdf:type            ?type .

  FILTER(?type != context:Sensor && ?type != context:DataGenerator ) .
}
```

Query 6:

```
SELECT DISTINCT ?subject
WHERE {
  ?co      context:about dsi:args[0] .
  ?co      context:about ?subject .
  ?subject rdf:type      dsi:args[1] .
}
```

Query 7:

```
SELECT DISTINCT ?type
WHERE {
  _:tmp_co context:about dsi:args[0] .
  _:tmp_co rdf:type      ?type .
}
ORDER BY (?type)
```

Query 8:

```
SELECT ?contextObject1 ?unit1 ?value1
       ?contextObject2 ?unit2 ?value2
WHERE {
  ?contextObject1 context:contains     ?contextObject2 .
  FILTER(?contextObject1 != ?contextObject2) .
  ?contextObject1 context:unit         ?unit1 .
  ?contextObject1 context:contextValue ?value1 .
  ?contextObject2 context:unit         ?unit2 .
  ?contextObject2 context:contextValue ?value2 .
  ?contextObject1 context:about        dsi:args[0] .
}
```

Query 9:

```
SELECT ?subject
WHERE {
  ?datagenerator context:measures ?subject
}
ORDER BY (?subject)
LIMIT 10000
OFFSET args[1]
(Aggregation was calculated in client-side logic.)
```

Query 10 (RDF/XML):

```
<contextType rdf:ID = args[0]
             context:accuracy  = args[4]
             context:timestamp = args[5]>
   <context:generatedBy>
             <args[7] rdf:about = args[6]/>
   </context:generatedBy>
   <context:contextValue>args[2]</context:contextValue>
   <context:unit>args[3]</context:unit>
</contextType>
</rdf:RDF>
```

Query 11:

```
This query is not yet implemented!
```

Query 12 (Sesame Transaction):

```
<transaction>
    <remove>
        <uri>
            http://localhost/owl/DataSetInstances.xml#args[0]
```

```
            </uri>
            <null/>
            <null/>
        </remove>
        <remove>
            <null/>
            <null/>
            <uri>
               http://localhost/owl/DataSetInstances.xml#args[0]
            </uri>
        </remove>
    </transaction>
```

D Appendix - Queries in SODA

This section shows the implemention of the benchmark queries in SODA. In the queries shown here the parameters are marked by args[i]. We took the same parameters for each query independent of the used query language.

Query 1:

```
Query query=db.query();
      query.constrain(Subject.class);
      query.descend("id").constrain(args[0]);
```

Query 2:

```
Query query=db.query();
      query.constrain(ContextObject.class);
      query.descend("type").constrain(args[0]);
      Query meta = query.descend("metadata");
            meta.constrain(ContextMetaData.class);
            meta.descend("time").orderDescending();
      Query subs = query.descend("subjects");
            subs.constrain(Subject.class);
            subs.descend("id").constrain(args[1]);
```

Query 3:

```
Query query=db.query();
      query.constrain(ContextObject.class);
      query.descend("type").constrain(args[0]);
      Query meta = query.descend("metadata");
            meta.constrain(ContextMetaData.class);
            meta.descend("time").constrain(args[2]).greater();
            meta.descend("time").constrain(args[3]).smaller();
      Query subs = query.descend("subjects");
            subs.constrain(Subject.class);
            subs.descend("id").constrain(args[1]);
```

Query 4:

```
Query query=db.query();
      query.constrain(ContextObject.class);
      query.descend("type").constrain(args[0]);
      Query dg = query.descend("dataGenerator");
            dg.constrain(DataGenerator.class);
            dg.descend("id").constrain(args[2]);
      Query meta = query.descend("metadata");
            meta.constrain(ContextMetaData.class);
            meta.descend("time").orderDescending();
      Query subs = query.descend("subjects");
            subs.constrain(Subject.class);
            subs.descend("id").constrain(args[1]);
```

Query 5:

```
Query query=db.query();
      query.constrain(ContextObject.class);
      query.descend("id").constrain(args[0]);
      Query sub = query.descend("subjects");
            sub.constrain(Subject.class);
            sub.descend("id").constrain(args[1]);
      Query dg = query.descend("dataGenerator");
            dg.constrain(DataGenerator.class);
```

Query 6:

```
Query query=db.query();
      query.constrain(ContextObject.class);
      Query subs1=db.query();
            subs1.constrain(Subject.class)
                  .and(query.constrain(ContextObject.class).not())
                  .and(query.constrain(ComposedContext.class).not())
                  .and(query.constrain(DataGenerator.class).not());
            subs1.descend("id").constrain(args[0]);
      Query subs=db.query();
            subs.constrain(Subject.class)
                .and(query.constrain(ContextObject.class).not())
                .and(query.constrain(ComposedContext.class).not())
                .and(query.constrain(DataGenerator.class).not());
            subs.descend("type").constrain(args[1]);
            subs.descend("id").constrain(args[0]).not();
```

Query 7:

```
Query query=db.query();
      query.constrain(ContextObject.class);
      Query subs = query.descend("subjects");
            subs.constrain(Subject.class);
            subs.descend("id").constrain(args[0]);

ObjectSet result= query.execute();

TreeMap<String, ContextObject> types = new TreeMap<String, ContextObject>();
      while(result.hasNext()) {
          ContextObject co = (ContextObject) result.next();
          types.put(co.getType(), co);
}

Set set = types.keySet();
Iterator i = set.iterator();
while (i.hasNext()) {
      i.next();
}
```

Query 8:

```
Query query=db.query();
      query.constrain(ComposedContext.class);
      Query subs = query.descend("subjects");
```

```
            subs.constrain(Subject.class);
            subs.descend("id").constrain(args[0]);

ObjectSet result= query.execute();
while(result.hasNext()) {
    ComposedContext cc = (ComposedContext) result.next();
}
```

Query 9:

```
Query query=db.query();
      query.constrain(ContextObject.class);

ObjectSet result= query.execute();
Hashtable<Subject, HashSet<DataGenerator>> resQuery
       = new Hashtable<Subject, HashSet<DataGenerator>>();
while(result.hasNext()) {
      ContextObject co  = (ContextObject) result.next();
      DataGenerator dg = co.getDataGenerator();
      if(dg != null){
         List subs = co.getSubjects();
         Iterator i = subs.iterator();
         while(i.hasNext()){
            Subject s = (Subject)i.next();
            if (!resQuery.containsKey(s)){
               HashSet<DataGenerator> dgs = new HashSet<DataGenerator>();
               dgs.add(dg);
               resQuery.put(s, dgs);
            }else{
               HashSet<DataGenerator> dgs = resQuery.get(s);
               dgs.add(dg);
            }
        }
     }
}

Enumeration e = resQuery.keys();
while(e.hasMoreElements()){
      Subject s = (Subject) e.nextElement();
}
```

Query 10:

```
ContextObject tw = new ContextObject(args[0], args[1], args[2]);
db.store(tw);
```

Query 11:

```
Query query=db.query();
      query.constrain(ComposedContext.class);
      Query subs = query.descend("subjectList");
            subs.constrain(Subject.class);
            subs.descend("id").constrain(args[0]);
```

```
ObjectSet result=query.execute();

ComposedContext toUp = null;
      while(result.hasNext()) {
            toUp = (ComposedContext)result.next();
            ArrayList subjects = toUp.getSubjectList();
            for (int i = 0; i < subjects.size(); i++){
                ((ContextObject)subjects.get(i)).setValue(args[1]);
            }
      }

db.store(toUp);
```

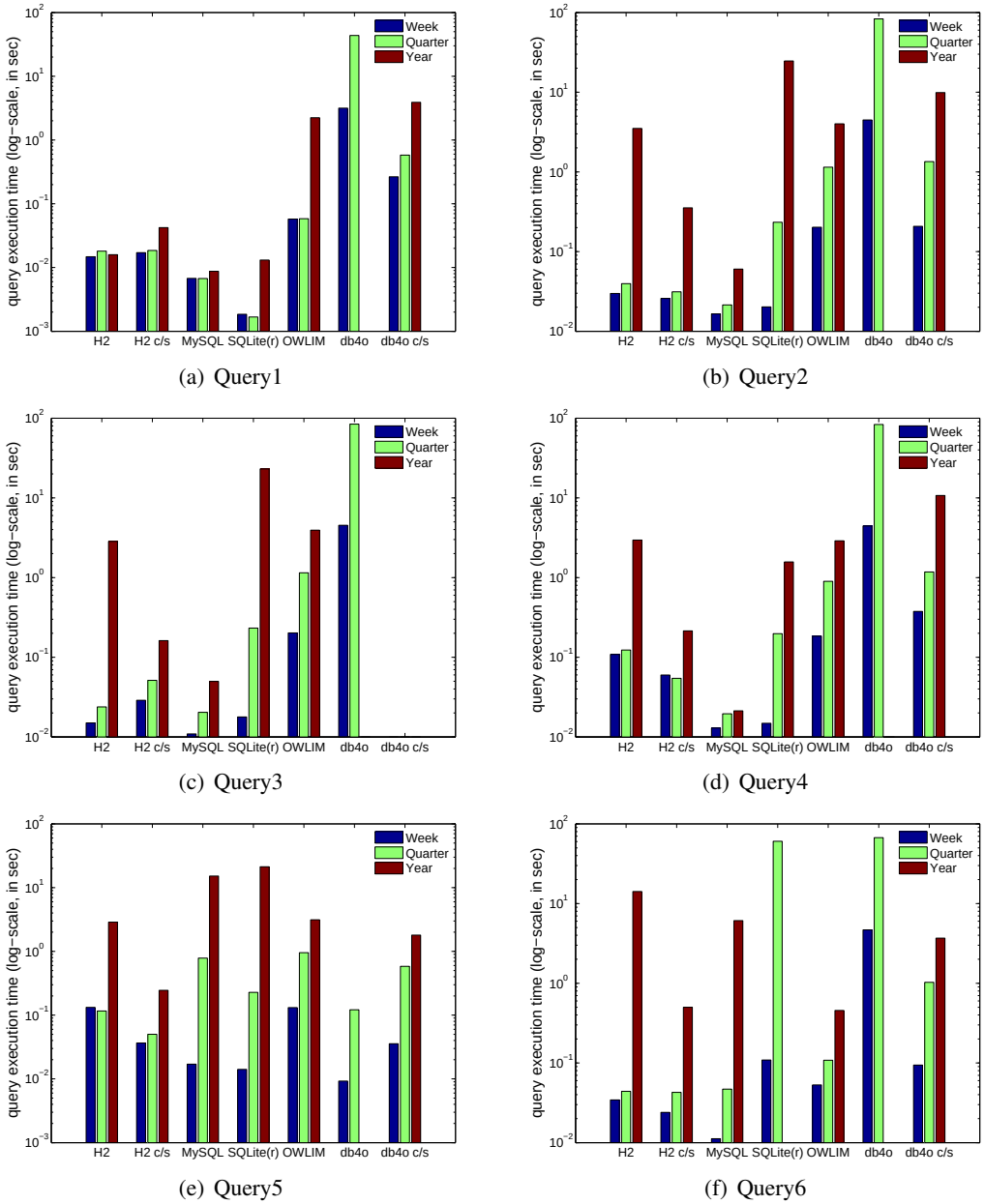
Query 12:

```
Query query=db.query();
      query.constrain(ContextObject.class);
      query.descend("id").constrain(args[0]);

ObjectSet result=query.execute();
while(result.hasNext()) {
      Object toDel = result.next();
db.delete(toDel);
}
```

E Appendix

The following figures (Figure 9 and 10) show the results of the single query runs in detail.

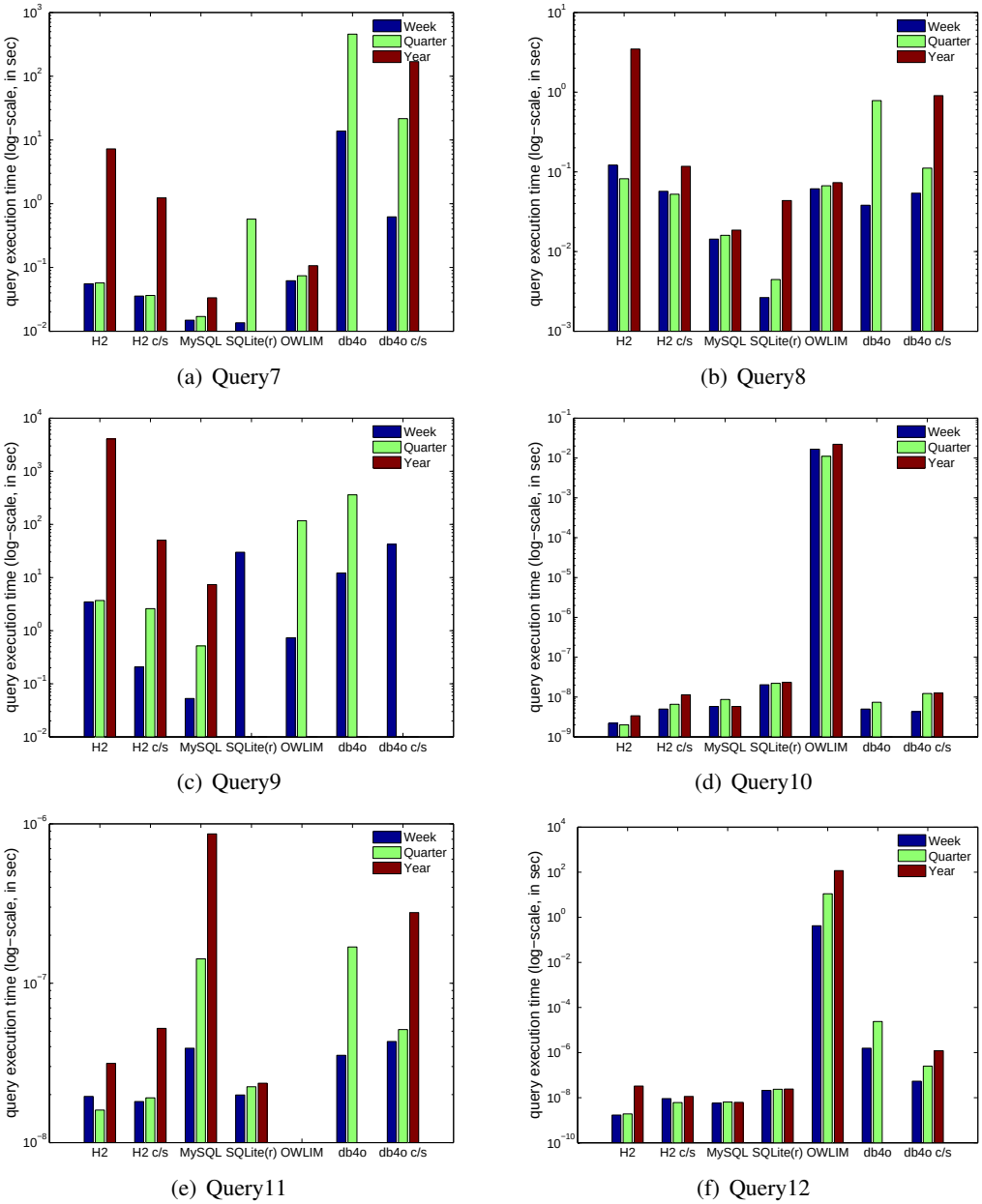


(a) Query1 (b) Query2

(c) Query3 (d) Query4

(e) Query5 (f) Query6

Figure 9: Single Query Evaluation for Queries 1 to 6



(a) Query7 (b) Query8

(c) Query9 (d) Query10

(e) Query11 (f) Query12

Figure 10: Single Query Evaluation for Queries 7 to 12