Storing and Managing Context and Context History

Alaa Alsaig, Ammar Alsaig, Mubarak Mohammad(✉), and Vangalur Alagar

Concordia University, Montreal, Canada
ms_moham@cse.concordia.ca

Abstract. Bringing context into systems design has added a new dimension to modern technology. In service-centric and social-centric systems, the personalization of services to accommodate the preferences of each individual is essentially based on context information. Due to this importance, a significant amount of research work is being done on structuring and modeling contexts. However, no work has been done on storing these models using recent database technologies and techniques. Also, there is no reported work that considers a structure for context history, which is essential to maximize accessibility and scalability of context information in dynamic settings. Motivated by these issues, we have developed a general structure for storing context using three different database models. Additionally, we have compared the three models in terms of their performance and modeling ability. In this paper we present the data models for context, context history, and provide a summary of the experimental analysis conducted on them.

1 Introduction

As the technology evolves, the dependency of society on the technology becomes more intense. This increases the need for smarter systems that can provide specific services rather than general ones. Services in the Health Care sector is an example. As a result many service-oriented systems have become pervasive, requiring context for service provision. Context can be either a location of a subject or any environmental surrounding such as temperature or weather affecting the subject. There has been many studies on defining and modeling context [1]. However, there is no work yet in designing context databases. A context database is essential to manage a heterogeneous collection of contexts and their history in order that context information, both past and current, are made available in a time-critical manner for providing critical services. This is the motivation for us to provide a general implementation structure for context that could be

Alaa and Ammar are sincerely thankful to Saudi Arabia Government and the Saudi Cultural Bureau for their financial support.

Vangalur Alagar—The work reported in this paper is supported by a grant received by this author from Natural Sciences and Engineering Research Council (NSERC), Canada.

P.C. Vinh et al. (Eds.): ICCASA 2013, LNICST 128, pp. 35–46, 2014.
DOI: 10.1007/978-3-319-05939-6_4,

embedded in a Service-oriented Application (SOA) or any other ubiquitous computing system. Additionally, we propose a design for storing context history that maximizes data management and enhances accessibility.

1.1 Contributions and Organization of Paper

Our contributions include (1) a general model for storing and managing contexts, (2) an implementation for the proposed context model in three different database models, (3) a comparison of the three implementations based on experimental studies, and (4) a general database structure for storing and managing context history. These contributions are organized as follows. In Sect. 2, we introduce a definition of context, types of context, and a generic context model. In Sect. 3, we provide the three database structures for the generic context model and compare them. In Sect. 4, we provide a solution for handling the context history. In Sect. 5, we provide a brief literature survey of context modeling and implementation. In Sect. 6 we summarize our ongoing research work.

2 Context

There exists a large body of literature in context, as understood in different fields such as linguistics, AI, philosophy, and Human Computer Interface. In ubiquitous computing, *context* is a meta-information that qualifies either data or information or an entity of interest in the system. Within SOA we can regard context as any element that could affect the service provision and execution operations. In general, [11] mentions that context is any environment element of an entity that gives rise to meaningful interpretation of a function computation. As an example, location of a subject is the context which will decide whether or not mobile service could be provided to that subject. In [15], context is formalized and defined as a set of dimensions and tags. The set of dimensions "Who, Where, When, What, and why" are introduced to construct any general context. This definition has been considered in [10] for configuring a service. In the definition of "configured service", context is split into *ContextInfo* and *ContextRule*, where *ContextInfo* is the context representation introduced by [15] and *ContextRule* is the service qualifier rule that has to be met for getting the service. Below, we first provide an extension to the work done by [15] and introduce a detailed study on the context structure from an empirical point of view. Next, we introduce types of contexts, and provide a design for the context considering all contexts' types.

2.1 Context Types

The three important entities in any SOA are *service*, *service requester* (SR), and *service provider* (SP). Each entity will be influenced by their own set of contexts. Thus, we define the three categories *Service Context* (SC) *Service Requester Context* (SRC), and *Service provider Context* (SPC). A context of type SC is to

describe the service status. For example, a service may be "temporarily unavailable" in some contexts or is available only in "certain contexts". A context of SRC qualifies the status while requesting or receiving the service. For example, the location and time parameters characterize the context of a client while requesting or receiving a service. A context of type SPC is to qualify service availability and service quality for a service provided by a SP. As an example, a SP may have license to provide service within 10 km of the location where SP is registered. So, his location and the authorized zonal information for his service contribute to constructing SPC contexts. Contexts from these categories regulate and restrict service provisioning in SOA. Contexts of SC and SPC types must be pre-defined in the system, although contexts of type SRC may vary dynamically due to the mobility of SRs. In general, a context type can be put into one of the three subtypes *permanent*, *temporal*, and *transient*. A permanent context needs to be saved. Contexts that arise in Health Care service domain are examples of this type. A temporal context may undergo changes. Many contexts that arise in business applications are of this type. As an example, a business rule of a multinational corporation might change depending upon the government imposed legalities. A transient context arises dynamically, and after its use it may never arise again. Contexts that arise in many game playing systems are of this type.

2.2 Generic Context Model

The generic context model has the three main parts *ContextInfo*, *ContextRule* and *ContextValue*. Based on the context representation introduced in [15] we have structured *ContextInfo* and *ContextRule* as shown in Fig. 1. However, *ContextValue* requires a more sophisticated structure in order to capture the change of values. We decided to include information such as the identifier of the context's collector and a date and time of collection.

The information included in the *ContextValue* is included in two different nodes. A dimension node information is specific to each dimension separately. This information includes source ID which is the context collector's identifier. Since information for each dimension can be collected by several collectors, it is

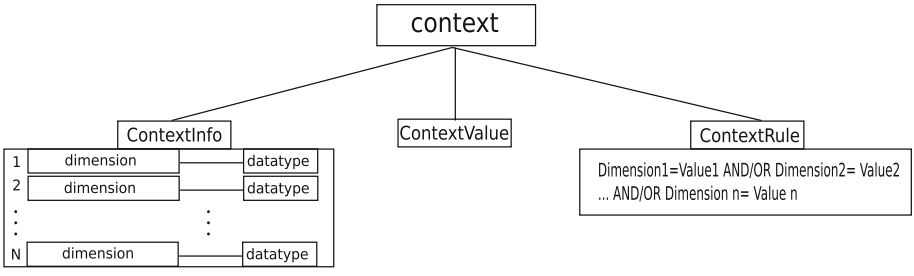

Fig. 1. The main structure of the context

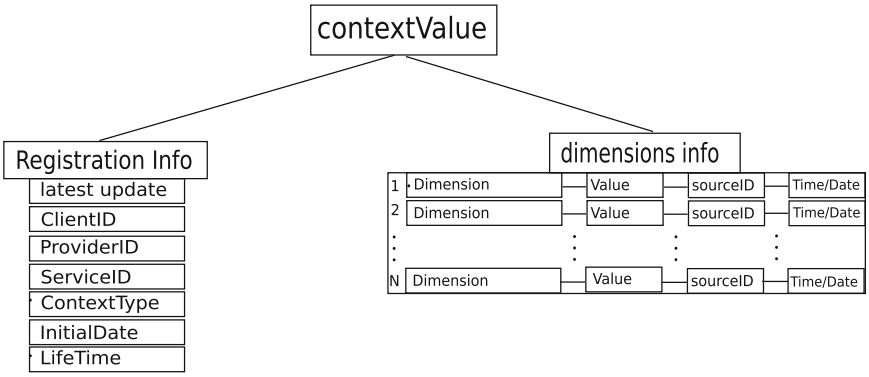

Fig. 2. The structure of *contextValue*

important to know which collector has collected the information to track it in case of a failure. Also, date and time of collection are made part of *ContextValue* in order to record the history of change. The second node of *ContextValue* is the registration node. This node includes information that is general for all dimensions such as context's type, requester ID, provider ID, service ID and date/time of the last update. This information except date/time of last update, is not updated frequently. Rather, they are set when the service is executed and will remain the same for other updates (Fig. 2). The fields in this node are defined below.

- lastupdate: includes the date and time of last update of the *ContextValue*
- requesterID: includes the ID of the requester to whom the service is provided
- providerID: includes the ID of the service provider
- serviceID: includes the ID of the service
- ContextType: can be permanent, temporal or transient
- intialdate: includes the date and time when the context was initialized
- lifetime: includes the time window for the life of the context

3 NoSql Implementation for Generic Context Structure

NoSql technology is selected to implement the generic context structure for the following reasons: (1) it supports semi or free schema [14] which makes it suitable for managing dynamic data, (2) it supports hierarchical structures, (3) it is highly scalable which makes it suitable for distributed databases, (4) it manages attributes with multiple values, and (5) it provides efficient query processing mechanisms [3]. There are three main categories of NoSql database, classified based on their storing techniques: *Document-Oriented*, *Key-Value*, and *Column-Oriented*. Each of these NoSql technologies has many tools to support its operations. Thus, we decided to use these three database technologies for managing contexts. We choose one implementation from each class: *MongoDB* for Document-Oriented, *Redis* for

Key-Value, and *Hbase* for Column-oriented. The following describes how each of these technologies can be used to implement context.

3.1 Service Context Model in MongoDB

MongoDB is an open source *document-oriented database*. Each record in this style is called a *document*. A document is made up of a group of *fields* and their associated values. It can contain *embedded documents* with an overall size that does not exceed 16 MB. The number of fields need not be the same in all documents. That is, each record can have different structure. Each document has a unique key by default. A secondary key can be assigned. A *collection* is a pool of documents, which is equivalent to a table in SQL. The database supports all primitive types (Integer, String, Float), and arrays.

Figure 3 shows our proposed MongoDB model for the generic context structure. In this figure, the *ContextInfo* node is modeled as an embedded document that contains all dimensions as fields with their types as values. The *ContextRule* is modeled as an embedded document with one field of string type. The *ContextValue* is modeled as an embedded document that contains fields and arrays as follows. The *datetime, clientID, providerID,* and *serviceID* are modeled as regular fields. Each dimension of the context is modeled as an array structure, which wraps the information specific to each dimension in one memory block. Thus, all information regarding one dimension including *sourceID, date/time of collection,* and *value* of the dimension can be retrieved by the name of the dimension. The rational for representing dimensions as arrays instead of embedded documents is to reduce the levels of document embedding. Increasing the levels of document embedding makes MongoDB's operations resource intensive and causes complex query processing and retrieval. Thus, with the current structure, when an update operation is performed, only *lastupdate* field and dimensions' values are updated with a single query.

3.2 Service Context Model in Redis

Redis is an open source advanced *key-value* database. A record in the key-value database consists of a key mapped to its corresponding value. Redis is considered

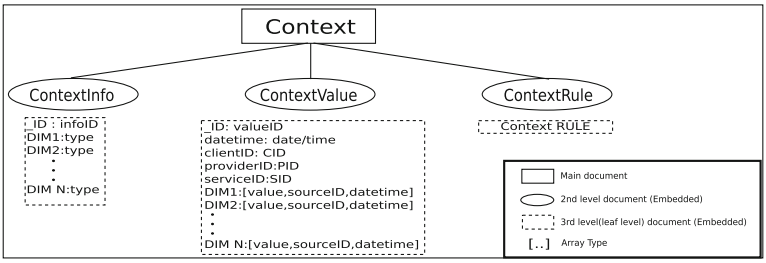

Fig. 3. Service context model in MongoDB

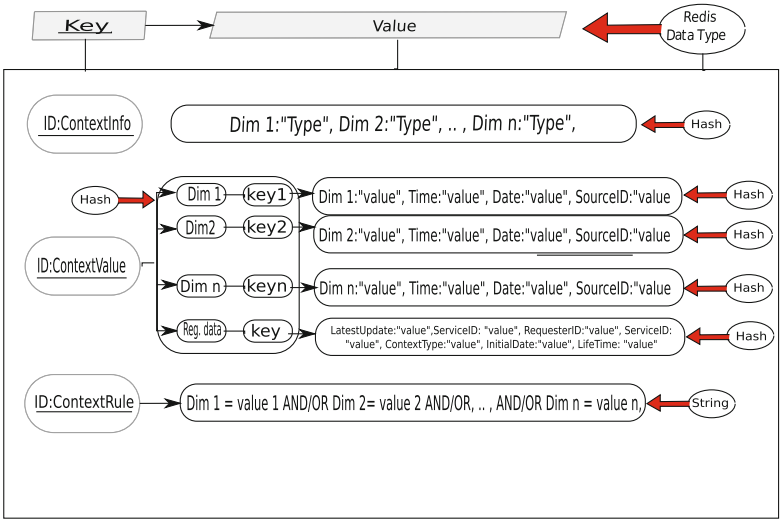

Fig. 4. Service context model in Redis

advanced because it provides five possible data structures for the value type. These data structures are *String*, *Hash*, *Set*, *List*, and *Sorted Set*. A String is a single value with a maximum size of 512 MB. A Hashes is a set of pairs where each pair consists of a name field and its corresponding value. A single Hashes record could have up to $2^{32} - 1$ pairs. A Set is an unsorted and not duplicated group of elements connected to a single key. In a Set, the maximum number of elements is $2^{32} - 1$. A List is simply a list of string values that are ordered as they are entered. A List could have a maximum size of $2^{32} - 1$ values. A Sorted Set is similar to Set, but each value is attached with a score. A score is an integer number attached to each value of a Sorted set. The values of a Sorted set are sorted in ascending order based on their score. The maximum number of values in a Sorted Set is similar to a Set.

Figure 4 depicts our Redis, key-value model, for context structure. The model uses strings and Hashes to model elements. Because *ContextInfo* consists of many pairs of dimension names and their types, Hash is a good data type to use. Similarly, *ContextValue* contains pairs. However, it has two levels of hierarchy. The first level is used to map the names of dimensions to their value keys. The second level is used to map the value keys to nested Hashes that include dimensions' information. The *ContextRule* attribute is modeled as a String data type because it contains only one value which is the *ContextRule* statement.

3.3 Service Context Model in Hbase

Hbase is an open source *column-oriented* database. It supports key-value techniques, and it is based on the *BigTable* technology [7], which is designed by Google. It provides flexible table structure. An Hbase table contains a bunch

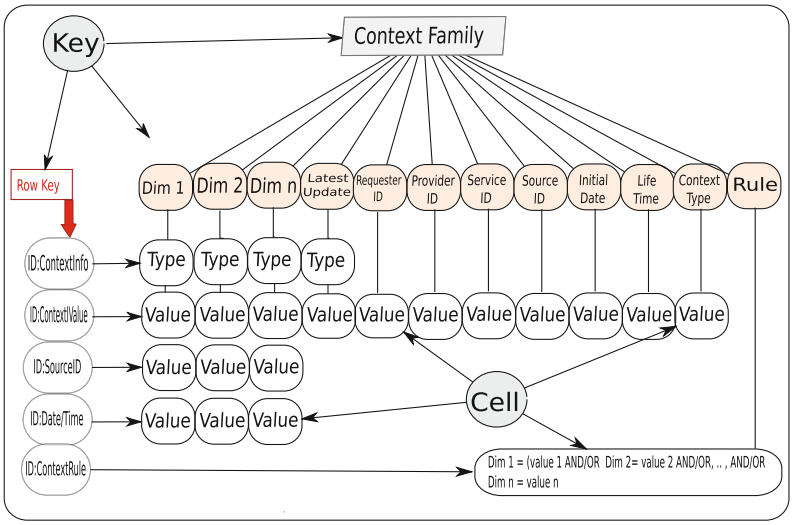

Fig. 5. Service context model in Hbase

of Key-Values wrapped together under one name. This name is called *Column Family* (CF). *Column Qualifier* (CQ) is a field of data. *Row Key* is a unique key that differentiates a row from another. *Cell* stores an atomic value. To store this value or retrieve it, three keys are needed. These keys are row key, column family, and column qualifier. The size of a cell could be from 10 to 50 MB [6]. *Version* is characterized by a time stamp. Every time data is inserted/updated in a cell, the system stores a time stamp for this action. If time stamp is not specified when retrieving data, the system automatically returns the latest one.

Hbase does not have a fixed pre-defined schema which makes it very flexible to structure Context Model. Figure 5 shows the Service Context Model in Hbase, in which we have named the column family as Context Family. The Context Family is mapped to the set of dimension names. It is also mapped to some columns that provide information to the *ContextValue*. Actually, columns are the data that are related to one or more rows. Rows are the data that are related to the dimensions. As illustrated by Fig. 5, only the *ContextValue* needs all the fields represented by columns. This results in rows with different length. The third row key is *ContextRule* which does not need any data of column defined in the structure. Thus, a new column qualifier is added and named Rule.

3.4 A Comparison of the Three Models

We compare the three models in terms of the features afforded by the underlying database models and in terms of their performances. Table 1 compares their structural features and Table 2 compares their performance.

The *CAP Theorem* [2], which studied *consistency*, *availability*, and *partition tolerance* of NoSql databases, states that any NoSql database should have two

Table 1. A comparison of the structural properties of three context database models

	Redis	MongoDB	Hbase
CAP theorem	CPT	CPT	CPT
Strengths	High speed	Flexibility, simplicity	Versions support, compressions
Weaknesses	Durability problem	Difficult to update	Can't work alone, or scale down
Maximum size	Section 3.2	16 MB	Cell 10-50 MB
Indexing	One index	Allows secondary index	Cell queried by (row key, CF, CQ)
Modeling ability	One structure, No hierarchy	Supports embedding	Tables and embedding

strong features out of three. In [9], it is stated that MongoDB, Redis, and Hbase have the two strong features *consistency* and *partition tolerance* (CPT). Redis is a flexible database but has some limitations, compared to the other two. The constraints on data type, indexing system, and key value structure make it more difficult to use with complex rich data. On the other hand, both MongoDB and Hbase can handle complex data. MongoDB supports hierarchical structures by permitting nested documents and allowing secondary indexing [4]. In Hbase, hierarchical structures are supported by nested columns with multiple indexing [7]. These features help developers to structure rich context data. MongoDB is easier than Hbase in configuring and coding Table 1.

We tested the performance of the three databases using YCSB benchmarking tool [5]. We used different workloads, defined by YCSB, where each workload differs from the other by the number and types of operations performed. The numbers shown in Table 2 represent the average result of each workload examined on different number of records that range from 10,000 to 1,000,000 records. We tested the performance based on two factors: runtime and throughput[1]. In our results, Redis occupies the first place in terms of runtime and throughput followed by MongoDB and, finally, Hbase. However, YCSB does not consider complexity of structure. Therefore, the results could change dramatically with more complex structures. Specifically, because Redis does not have pre-defined data structures, it consumes more operations to perform a single query. As a result, Redis performance decreases, whereas MongoDB and Hbase seem to perform better with complex structures.

4 Database for Managing Context History

An analysis of historical information of contexts will provide valuable lessons to service providers in modifying their business practices in future. Historical data regarding clients is very valuable for improving businesses and capturing

[1] Throughput is the number of operations performed per millisecond.

Table 2. A comparison of the performances of three context database models

Workloads	Redis		MongoDB		HBase	
-	Runtime (ms)	Throughput	Runtime (ms)	Throughput	Runtime (ms)	Throughput
Workload (a)	459.25	2279.725	759	1322.43	11477	8682.03
Workload (b)	391.75	2589.860	737.25	1363.452	12508.5	425.7
Workload (c)	356.00	2899.95	660.25	1514.95	8136.74	443.93
Workload (d)	368.50	2785.15	700	1431.6	5331.75	485.44
Workload (e)	5196.25	192.65	3922.25	424.91	7351.5	299.07
Workload (f)	471.50	2132.19	1124.5	933.87	4295	506.41

the market needs and business trends. Through the accumulated contexts, service providers can observe and evaluate the services provided in the past and re-evaluate their business policies. In particular, service providers can perform some data mining and discover the contexts in which the frequency of service requests peaked. When some of these contexts occur in future, providers can be better prepared to serve the clients. Also, historical information can be critical in health-related applications where there is an essential need to access the history of patients. For example, in providing health care for mental illness, it is very useful to investigate a patient's reactions in different situations for understanding and identifying the problem. The volume of data involved in historical evolution of contexts is rather immense. Consequently, we need a structure in which information is allowed to grow in an orderly manner, data access time is optimized, and insertion and deletion of information are done efficiently.

We propose a hierarchical structure that categorizes the historical contexts based on services associated with providers of the services. Figure 6 shows the hierarchy, where the subtree rooted at a service provider contains the services and the contexts of providing these services. Thus, with the help of information included in the data registration node, reaching the contexts of a specific service for a specific client can be an easy process. Also, the hierarchical classification helps in keeping the growth manageable by narrowing it down to a specific provider, and service. Thus, the data related to one provider to one service is clustered together. Therefore, when providers are to access services' contexts they only need to surf their own contexts between their own clients. This classification can also be furthered by clustering the contexts for providers based on contexts' types. Thus, permanent contexts are clustered together and remain untouched, whereas temporal contexts are visited periodically for cleaning.

Additionally, to keep the history manageable we introduced *lifetime*, *intialdate* and *contextType* fields in Sect. 2.2. Based on *contextType* a context is either to be deleted or retained. In case the type of context is *permanent*, context is persisted. If the context type is *transient*, it is not saved at all. If the type is *temporal*, the lifetime field is added to the field *intialdate* which will define the expiry date of the context. This expiry date is calculated whenever the clean-up process is activated and the record is deleted if either the current date information in it is equal or past the expiry date information.

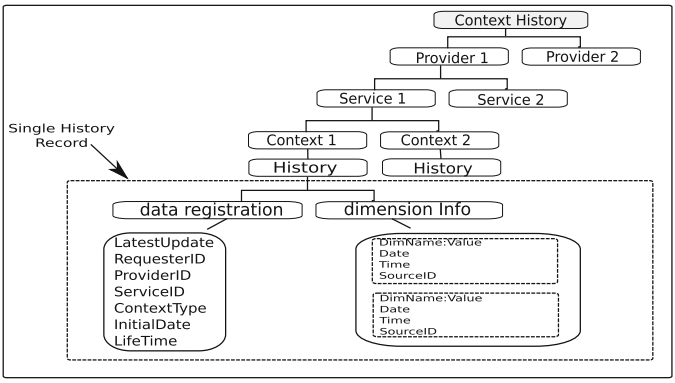

Fig. 6. Context history hierarchical structure

5 Related Work

There exists a large body of literature in the study of context. In this section we have chosen some recent published work on context modeling to compare our work. In general, to our knowledge there is no work done yet on database models for managing context and context history.

The UML context model proposed in [12] considers atomic and complex contexts. An atomic context is modeled as a class in which the two attributes are the name of the context and the source name of context. The only attribute of the complex context is the aggregation of its different contexts, with some logical operations. The two context models are independent of service. However, there is a class, called context-awareness, which is a component of the service. The proposed context model is both abstract and incomplete. It is abstract in the sense that the authors did not provide any language or database support that are necessary for implementing the model. It is incomplete in the sense that the type information necessary to capture the heterogeneity of information, the nature of context (permanent or temporal), and rules for using it in services are not modeled. Although the authors [8] claim to have put forth a context-aware service application, the work does not provide any view of the context structure and how it is defined. Actually, the work is an extension to [12] that they considered as state and event based context. On one hand, a state-based context includes data of attributes that could be entity, device or user related. On the other hand, the event-based context encompasses a bunch of entity events. These events could be related to an application or a user with consideration to events' history. However, there is no elaboration for how the context is structured and where the data is stored. Also, there is no specific structure for the history and what data could be included. In [13], the authors have introduced a context structure mainly for *Mashup* application requirements. They have considered the dimensions *when*, *where*, *what* and *who* to construct contexts. However, the structure assigns several entities for each dimension. This

makes context structure complex. In general, not all applications require the same context information. Consequently, their model could result in aggregating useless information. The model does not provide any mechanism to add another dimension. In addition, although the context change history was mentioned in the paper, there was not any information regarding history structure, model or attributes and data of the history.

6 Conclusion

The significant virtue of the context structure that we have proposed in this paper is its ability to handle the richness of context information. It can fit the needs of service definition, service provider characterization, and service requester preferences. The context model is independent from service models of service providers, yet the context structure can be adapted to fit in service models. The three database structures that we have investigated seem to adequately handle the management requirements of a large collection of contexts and their histories. We have compared the three database organization from both structural and performance characteristics. We are currently embedding the context databases in service registries, the central publishing house in a service-oriented architecture.

References

1. Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., Riboni, D.: A survey of context modelling and reasoning techniques. Pervasive Mob. Comput. **6**(2), 161–180 (2010)
2. Brewer, E.A.: Towards robust distributed systems. In: PODC, p. 7 (2000)
3. Chakraborty, S., Sarkar, M., Mukherjee, N.: Implementation of execution history in non-relational databases for feedback-guided job modeling. In: Proceedings of the CUBE International Information Technology Conference, pp. 476–482. ACM (2012)
4. Chodorow, K.: MongoDB: The Definitive Guide. O'Reilly, Sebastopol (2013)
5. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. ACM (2010)
6. Dimiduk, N., Khurana, A., Ryan, M.H.: HBase in Action. Manning, Shelter Island (2013)
7. George, L.: HBase: The Definitive Guide. O'Reilly Media Inc., Sebastopol (2011)
8. Grassi, V., Sindico, A.: Towards model driven design of service-based context-aware applications. In: International Workshop on Engineering of Software Services for Pervasive Environments: in Conjunction with the 6th ESEC/FSE Joint Meeting, pp. 69–74. ACM (2007)
9. Han, J., Haihong, E., Le, G., Du, J.: Survey on NoSQL database. In: 2011 6th International Conference on Pervasive Computing and Applications (ICPCA), pp. 363–366. IEEE (2011)
10. Ibrahim, N.: Specification, composition and provision of trustworthy context-dependent services. Technical report, Concordia University (2012)

11. Keith, J.: Building a contextaware service architecture. http://www.ibm.com/developerworks/architecture/library/ar-conawserv/index.html
12. Sheng, Q.Z., Benatallah, B.: ContextUML: a UML-based modeling language for model-driven development of context-aware web services. In: International Conference on Mobile Business, 2005, ICMB 2005, pp. 206–212. IEEE (2005)
13. Treiber, M., Kritikos, K., Schall, D., Dustdar, S., Plexousakis, D.: Modeling context-aware and socially-enriched mashups. In: Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups, p. 2. ACM (2010)
14. Tweed, R., James, G.: A universal NoSQL engine, using a tried and tested technology (2010)
15. Wan, K.: Lucx: lucid enriched with context. Ph.D. thesis, Concordia University (2006)