Update Propagator for Joint Scalable Storage

Paweł Leszczyński and Krzysztof Stencel

Faculty of Mathematics and Computer Science
Nicolaus Copernicus University
Chopina 12/18, 87-100 Toruń
{pawel.leszczynski,stencel}@mat.umk.pl
http://www.mat.umk.pl

Abstract. In recent years, the scalability of web applications has become critical. Web sites get more dynamic and customized. This increases servers' workload. Furthermore, the future increase of load is difficult to predict. Thus, the industry seeks for solutions that scale well. With current technology, almost all items of system architectures can be multiplied when necessary. There are, however, problems with databases in this respect. The traditional approach with a single relational database storage has become insufficient. In order to achieve scalability, architects add a number of different kinds of storage facilities. This could be error prone because of inconsistencies in stored data. In this paper we present a novel method to assemble systems with multiple storages. We propose an algorithm for update propagation among different storages like multi-column, key-value, and relational databases.

Key words: multi storage, scalability, key-value storage, column family storage, scalability, data consistency, web applications

1 Introduction

Modern web applications provide users with a significant number of interactive and personal features. These require several queries to a database and make an application data-intensive. As a number of users grows, a database becomes a bottleneck of the whole system. All the other components of systems scale well and can be easily extended while scaling the database component is non-trivial. Scalability plays a noteworthy role in the web industry. At the beginning of the operation of a new application, only a few resources are needed. However, its owner has to be prepared for expansion. When the website suddenly gains popularity, the system architecture needs to be ready for a workload boost.

The problem of a database bottleneck is well-recognized in the industry, and many fixes have been proposed, however, the general solution is still unknown. When a database workload increases, it is a common practice to split a database into smaller parts, and distribute on more than one server. However, this is not a scalable architecture and rather a fix for current problems. The other option is to migrate some data into scalable storages. For this purpose one can apply a local NoSQL storage, or use SaaS platforms (Software as a Service) like Amazon

CONCURRENCY, SPECIFICATION AND PROGRAMMING
M. Szczuka et al. (eds.): Proceedings of the international workshop CS&P 2011
September 28-30, Pułtusk, Poland, pp. 334-345

S3, Amazon SimpleDB, or others. The general impression is that, whatever the solution has been chosen, database is split into several smaller instances running on different storage engines and servers. This however, makes the overall system architecture more complicated and, as a result, makes an application harder to maintain, and makes the whole development process more expensive. Sometimes the same data is stored in several locations, and the application's logic needs to keep the replicated data in a consistent state. This requires developers to take care of all data writes and apply them on several storages. When dealing with big applications, this can lead to errors which are hard to detect and repair.

In this paper, we propose a novel data propagation algorithm for joint storages that maintains replicated data in multiple sources in a consistent state. When an update on one source occurs, our system modifies data in other storages, if it is needed. Figure 1 shows an architecture of the constructed system. We believe that, the proper update propagation on underlying storages allows constructing a scalable joint storage while taking all advantages of the underlying systems. The paper makes the following contributions. (1) We suggest a novel architecture model for building several storage systems into a system. (2) We present an update propagation algorithm for keeping data in a consistent state.

The paper is organized as follows. In Section 2 we summarize the related work. Section 3 describes the motivating example of a bookstore application. In Section 4 we introduce our data model, and in Section 5 we define the problem in a formal way. Section 6 describes the propagation algorithm. Section 7 concludes.

2 Related Work

Several publications address scalability and consistency issues. The paper [1] describes design choices and principles for a scalable storage. The authors address a problem of filling the gap between key-value and relational storage, which is investigated in our research. The article [10] categorizes consistency levels into: serializable, session consistency, adaptive, and mixed. Serializable corresponds to a full transactional model while session consistency only assures to read own writes. In the adaptive level the system adjusts consistency to the current situation. This is done by comparing the cost per transaction. The system is hosted on an Amazon platform and is aware of a total hosting cost and a cost of a single failed transaction. When workload changes, it changes the consistency level to serve data at the lowest possible cost. The research addresses the same problem as ours, however, the cost comparison methodology restricts mainly to SaaS.

An interesting, ongoing research is a modular cloud storage system called Cloudy [9]. It is going to be built on the top of different storage engines similarly to our system. Cloudy provides interfaces for read and write operations. This makes underlying storages invisible for an application server and is a clear design pattern. However, this concept tends to be complicated and hard to maintain. Updates are mainly simple, and mostly modify a single record while reads get more complicated. Additionally, there are plenty of NoSQL storages and they

change rapidly with new updates that makes storage internals difficult to maintain up to date. Furthermore, NoSQL storages provide plenty of API clients like JSON, XML, THRIFT [16] etc. and rewriting all of them is almost impossible. This issue is not present in the architecture from Figure 1 since we only care about a proper update propagation.

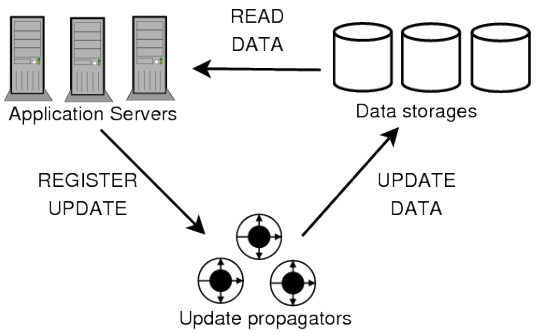

Fig. 1. The update propagator architecture

The general problem can be described as keeping data consistent in different storages. We have also examined possible solutions in problems that are similar, but not exactly the same. One of them, is a consistent caching which means an evaluation of invalidation clues of the cached data when an update on a data source occurs. Authors of [8, 7, 12] present a model that detects inconsistency based on statements' templates. However, their approach cannot handle join of attribute families or aggregation operators that are very common in web applications. Our approach is based on a graph with edges that determine the impact of the update operations on the cached data. The idea of the graph representation has been presented in [2, 5, 6]. The vertices of the graph represent instances of update statements and cached data objects. However, nowadays most web pages are personalized, and number of data objects has increased, and multiplied by a number of application users. According to these observations, the graph size can grow rapidly and the method becomes impractical. The graph size cannot depend on the size of data. In our approach the dependency graph has vertices that represent data modifications and read operations. We present a dependency graph algorithm whose efficiency depends only on a number of columns.

3 Motivating Example

Let us now consider a simple bookstore platform that allows listing, searching, and buying books. Additionally each book has a list of opinions displayed on its info page. The database of the presented application needs to store: book

information, users' opinions on books, and information about sold items and users who bought them. Figure 2 depicts an example data model.

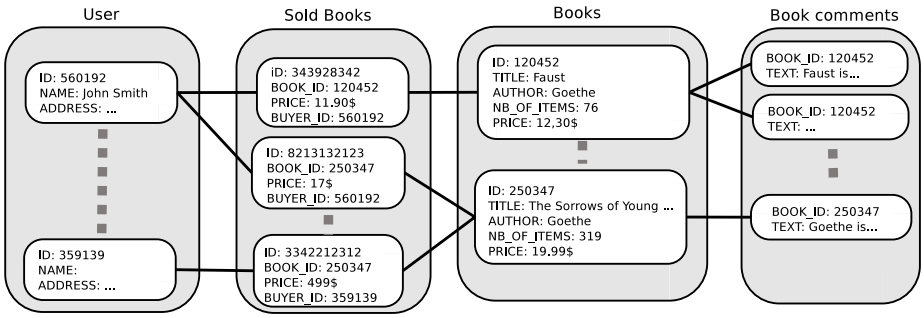

Fig. 2. The simple bookstore: book information, opinions, sold items and user data

One can identify the most common queries that performed on the platform. Users list books, view result pages and full-text search items. When a book's page is loaded, the system retrieves the information on this book together with the opinions. When a user decides to buy a product, the system updates database to adjust the number of available items.

When creating a scalable and efficient architecture, several different storages could be used in order to achieve better performance. Indexing engines like Sphinx [15] or Lucene [11] can be used for paging and searching products. If the number of products is large, the number of opinions can be expected to grow rapidly. Thus, it is worth storing them in a distributed column family storage like Cassandra [4]. Product information is accessed frequently and key-value storages like MemcacheDB [13] or Redis [14] may be applied. When selling products, it is frequently a business requirement to store the accountancy data in a relational database to ensure the transactional correctness.

As this analysis shows, in order to achieve better performance of our hypothetical system, it is reasonable to build different types of storage into it. In the following Sections, we show methods how to architect such a system and most notable, how to preserve the required level of consistency among various storage components.

4 Update Propagator

In this section, we present our data model and define the consistency problem in a formal way. Then we describe how a dependency graph is constructed and how it is used for proper data modifications.

4.1 Basic Schema Assumptions

Suppose our data consists of the k relations: R_1, R_2, ..., R_k. We assume that each relation has exactly one primary key element, and for the clarity we name it *id*.

$$R_i(id, r_{i,1}, r_{i,2}, r_{i,3}, \ldots, r_{i,n_i}) \tag{1}$$

This means that, for each relation R_i, the following functional dependency is satisfied:

$$id \rightarrow r_{i,1}, r_{i,2}, r_{i,3}, \ldots, r_{i,n_i} \tag{2}$$

In our schema, we allow *one-to-many* associations between the relations R and S and denote $R \prec_{r_i} S$. This means that, r_i is a foreign key in S and each tuple in S has a value of r_i equal to the primary key of some tuple in R. We also assume, our schema is in the third normal form (3NF). Any two relations R and S are associated, $R \lhd S$, if there exist relations $S_1, S_2, \ldots, S_i$ and attributes r_1, r_2, ..., r_{i+1} such that:

$$R \prec_{r_1} S_1 \prec_{r_2} S_2 \ldots S_{i-1} \prec_{r_i} S_i \prec_{r_{i+1}} S. \tag{3}$$

We also allow *one-to-many* associations between attributes of the same relation, $R \prec_r R$, that may be useful when having hierarchical data. As an opposite, we do not allow *one-to-one* associations between the relations. The schema is used to represent abstract data architecture rather than to define how data is stored. According to this, *one-to-one* associations can be replaced with a single relation since it does not influence concrete data storages.

4.2 Write Operations

Our data model adds also some restrictions to data accesses and modifications. We assume that each write operation modifies a single tuple in a projection relation and is specified by an *id* parameter. We distinguish three types of write operations: adding a new tuple, editing an existing tuple attributes' except for *id*, and deleting it. In general case, a write operation can be represented as:

$$(R, type, value_{id}, \{(r_i, value_{r_i}), \ldots, (r_j, value_{r_j})\}) \tag{4}$$

Changes are applied on a relation R. When adding a new tuple, we fill it with attributes' values from the fourth parameter. As a result of an operation in underlying storages, we retrieve $value_{id}$ that is the primary key of a new tuple. In case of updating an existing row, $value_{id}$ determines the tuple, and the last element contains attribute, that are going to be changed, and their new values. The list of attributes and values remains empty when deleting a tuple. The tuple is then determined by $value_{id}$ as in the update case.

4.3 Underlying Storages

Next we are going to define data stored in underlying storages. Suppose R is a relation and S is an arbitrary sequence of relations associated with R:

$$S = (S_1, S_2, \ldots, S_i) \wedge R \lhd S_1 \wedge R \lhd S_2 \wedge \ldots \wedge R \lhd S_i \tag{5}$$

For each $S_j \in S$ we take relations $S_{j,1}$, $S_{j,2}$, $\ldots S_{j,k}$ and attributes r_{j_1}, r_{j_2}, $\ldots$, r_{j_k} such that:

$$R \prec_{r_{j_1}} S_{j,1} \prec_{r_{j_2}} S_{j,2} \prec_{r_{j_3}} \ldots \prec_{r_{j_k}} S_j. \tag{6}$$

Then we define R_{S_j} as follows:

$$R_{S_j} = S_{j,1} \bowtie_{S_{j,1}.id=r_{j_2}} S_{j,2} \bowtie \ldots S_{j_k} \bowtie_{S_{j,k}.id=r_k} S_j \tag{7}$$

Let $r_1, r_2, \ldots, r_i$ denote attributes of $R, R_{S_1}, R_{S_2}, \ldots, R_{S_i}$. In our data model we allow projections of the form:

$$\pi_{R.id,r_1,r_2,\ldots}(R \bowtie_{R.id=r_{1_1}} R_{S_1} \bowtie_{R.id=r_{2_1}} R_{S_2} \ldots \bowtie_{R.id=r_{i_1}} R_{S_i}) \tag{8}$$

In other words, we allow joins between R and arbitrary number of relations associated with R. We require that the primary key of R is projected and allow arbitrary attributes from $R, R_{S_1}, \ldots, R_{S_i}$ to be projected. We call such projection *a safe projection* and R is denoted as *the primary relation* of the projection.

Since we allow the same relation attribute to be projected several times, we distinguish between *relation attributes* and *projection attributes*. For each projection attribute, we collect a sequence of foreign key relation attributes used to project it, and we call it *a trace*.

The underlying storages can also contain processed results of *safe projections*. This can be simple operation of *count* or *sum*. For this purpose we define two types of selections that are allowed in our model: *safely updatable* and *incrementally updatable*. The first type applies, when a tuple can be modified based on its current data and an update U. An example of such operation is a *count* aggregation, while a *sum* is not such a selection, since given a sum value and modification of one row, we cannot recompute it. We need to know the former value of an element to compute the difference between former and current state. The *sum* is *incrementally updatable* selection, i.e. when an update U adds a new tuple, a tuple can be modified based on its current content and U.

5 Consistency Problem

The problem can be specified as follows. Suppose a data model constructed as defined in section 4.1, where data is stored in different data storages. When an update request occurs, the system needs to apply it to the underlying storages. Generally speaking the problem may be understood as finding a function that applies data changes of a given update to the underlying storages. This can lead to several problems. First, updating storages has to be an atomic operation

and cannot partially modify storages leaving some data unchanged. Second, an updating function needs to handle associations between relations: for instance adding a new tuple into a storage may cause invalidation of tuples in other storages. This intuitive description of the problem leads us to a formal definition.

Definition 1 (Data Consistency Problem–DCP). *Suppose a system with projections $P_1, P_2, \ldots, P_j$ containing data in a state T_1, T_2, ..., T_j. An update U changes a tuple in some relation and modifies a state of projections into T'_1, T'_2, ..., T'_j, where $T_i = T'_i$ if P_i has not been changed. A consistent data propagator is a computable function F, such that $F(U, T_1, \ldots, T_j) = (T'_1, \ldots, T'_j)$.*

Suppose n is a total number of tuples stored in underlying storages, $n = |P_1| + |P_2| + \ldots + |P_j|$. Additionally let us define m as a sum of the number of projections stored in underlying storages and the number of projected attributes. This means that m depends on a data schema complexity, in contrast to n, which is dependent on a size of stored data. The complexity of algorithms solving a DCP problem is a function of n and m. The purpose of the presented research is to provide a construction which is independent of n. This will assure a scalability of the constructed system, since its complexity will not depend on a data size.

6 The Propagator Algorithm

6.1 The Dependency Graph

This section describes how a dependency graph is constructed. Let G denote a dependency graph with $G = (V, E_{strong}, E_{weak})$, where V stands for a set of vertices and E_{strong}, E_{weak} stand for sets of directed edges, which are called strong and weak edges respectively.

First let us assume A is the smallest set that contains all attributes of all relations:

$$A = Attr(R_1) \cup Attr(R_i) \cup \ldots \cup Attr(R_k) \tag{9}$$

where $Attr(R)$ stands for the set of attributes in relation R. At this stage, we distinguish attributes from different relations with the same name and consider them as separate elements of A. Then P is a set of all safe projections that are stored in the underlying storages:

$$P = \{P_1, P_2, P_3, \ldots\} \tag{10}$$

In the data model, we allow modifications of single tuples. Suppose U is an update operation, as specified in the equation (4). Let us now define a function $Map(U)$ as follows:

$$Map(U) = (R, type, \{r_i, \ldots, r_j\}) \tag{11}$$

The method maps a write operation so that, two updates that perform the same operation on the same attributes, are mapped into the same value. Next we define M as a set of all values of Map:

$$M = \{Map(U_1), Map(U_2), Map(U_3), \ldots\} \tag{12}$$

Then a set of vertices V is defined as a sum of A, P, and M:

$$V = A \cup P \cup M \tag{13}$$

Next we define strong and weak edges contained in G. Suppose relations R and S such that $R \prec_r S$. Each primary key is connected by strong edges with all foreign keys in the relation:

$$(S.id, S.r) \in E_{strong} \tag{14}$$

and each foreign key is connected by a strong edge with the primary key of the foreign relation:

$$(S.r, R.id) \in E_{strong} \tag{15}$$

Each projection vertex π is connected by a strong edge with the primary key of its primary relation. The edge goes from the primary key to the projection vertex:

$$(R.id, \pi.id) \in E_{strong} \tag{16}$$

and each projected attribute r is connected by weak edges with π:

$$(r, \pi) \in E_{weak} \tag{17}$$

Next we define edges connecting update vertices. A vertex $Map(U)$ is connected by a strong edge with $R.id$:

$$(Map(U), R.id) \in E_{strong} \tag{18}$$

and by weak edges with all modified attributes:

$$\forall_{i=1,\ldots,j}(Map(U), R.r_i) \in E_{weak} \tag{19}$$

This ends the definition of the dependency graphs and an example of such graph is shown on Figure 3.

6.2 Helpful Methods

The presented construction relies on drivers of underlying storages, and we require them to implement $addPrimary(U, \pi)$ and $add(U, \pi)$ methods, which add a new tuple to π. The $addPrimary$ method generates and returns the primary key of a new tuple, while add takes the primary key as an argument in values of U. We also make use of $modify$ and $delete$ methods, which change a single tuple. These methods can be easily implemented for every storage engine.

Next we define $Proj(U)$ function that returns projections affected by U. These are projections π, such that there exists a relation attribute r such that:

$$\{(Map(U), r), (r, \pi)\} \subset E_{strong} \cup E_{weak} \tag{20}$$

Additionally, for each update U we define a $Prim(U)$, which specifies a projection where data changes are at first applied.

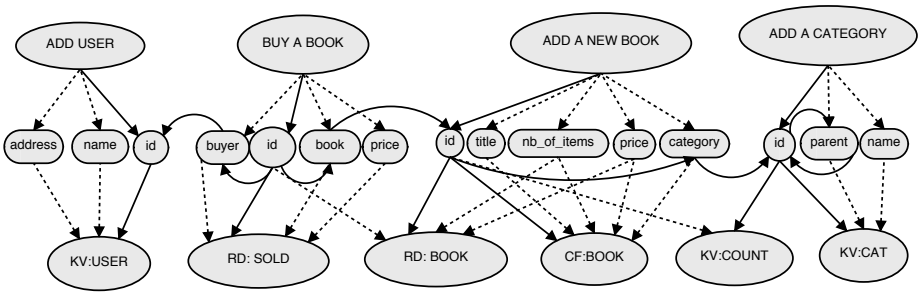

Fig. 3. Graph example from a bookstore application. The upper vertices represent update operations. The middle ones correspond to relation attributes of relations: user, sold items, book and category. The lower ones are vertices of the underlying storages, which will be described in section 6.4.

Let $Find(U, \pi)$ denote a function that identifies tuples modified by an update U in a projection π. For each projected attribute in π, we focus on its trace, which is a sequence of foreign key attributes and determines how the attribute has been projected. We examine the set of traces of projection attributes in π. Each trace corresponds to a single tuple modified by U and the algorithm constructs a strong path (i.e. a path composed of strong edges) from an update vertex to a projection one. When traversing the path, it collects visited attributes' values and, as a result, retrieves the primary key of the modified tuple at the end. In general $Find(U, \pi)$ returns pairs containing the primary key of a modified tuple and a strong path used to determine it. Additionally we define $A(U, \pi, p)$ method which given an update U, a projection π and a strong path p, evaluates a trace corresponding to p and returns projection attributes that are projected according to the evaluated trace.

The algorithm also uses a $Join(p)$ function that, given a strong path between the update and projection vertex, returns 0 if the path corresponds to a simple projection without join statements or 1 in the opposite case. As the last, we will make use of Mod method. The method, given a tuple in the projection, modifies it according to the values of an update. It uses a strong path to determine how an update changes the values.

6.3 The Main Algorithm

In this section, we describe a construction of the algorithm:

1. If type of U is *add*, then apply $addPrimary(U, Prim(U))$ and append its result to values of U as a primary key of the new tuple.
2. Let us define a set of projections T that are going to be updated. If type of U is *add* then $T = Proj(U) \setminus Prim(U)$, in other case let $T = Proj(U)$.
3. For each $\pi \in T$:
   3.1 For each $(tupleId, p) \in Find(U, \pi)$:
      3.1.1 Let $Val_p = \{(r_i, value_{r_i}) : r_i \in A(U, \pi, p) \wedge (r_i, value_{r_i}) \in Val\}$

3.1.2 If $Join(p) = 0$, then apply $add(U, value_{id})$, $delete(U, value_{id})$ or $modify(\pi, value_{id}, Val_p)$ according to the given update type of U.
3.1.3 If $Join(p) = 1$, then apply $Mod(Val_p, \pi, p, tupleId)$.

First, the algorithm checks a type of a given update U. If a new tuple is going to be inserted, it is first added to the primary projection of the updated relation where a new tuple gets the primary key. In step 2, the algorithm finds all projections that have been affected by U. When type of U is *add*, then the primary projection of the modified relation is excluded since the new tuple has already been added there in step 1. In steps 3 and 3.1, we iterate through all modified projections and all modified tuples in each projection. Modified tuples are represented as elements of $Find(U, \pi)$ which contain the primary key of the modified tuple and a strong path corresponding to the trace of modified attributes.

Step 3.1.2 describes the simple case when the algorithm fills underlying storages with modified values. This happens when π contains a subset of attributes of relation R which is modified by U, and the algorithm runs the requested operation on a tuple in underlying storage. Step 3.1.3 applies data changes on tuples, that via *one-to-many* joins, contain data affected by U. The changes are applied by the Mod method.

6.4 A Dependency Graph Example

Figure 3 shows the dependency graph for the presented bookstore example. Suppose *KV:USER* represents a key-value storage where user data is stored. Then *RD:SOLD* and *RD:BOOK* vertices represent relational databases. In the first one, financial data is stored, while the second contains some book information including number of items in stock, which has to be modified in a transactional way. *CF:BOOK* is a column-family storage containing additional book data and *CF:CAT* contains category tuples. Suppose *KV:COUNT* is a key value storage that contains: the number of books that have a *category* attribute equal the primary key of this category, and an amount of books assigned to its children nodes. These are not the counters of items in the subtree, and rather a direct number of occurrences of a given *category* attribute in book relation. We assume each value contains the category's primary key, a number of books in this category and in its children nodes.

As an usage example, let us suppose, someone buys a book. At first, a new tuple is added into *RD:SOLD* in a transactional manner. Then the data propagator algorithm is run to update data in other data sources. Additionally, a tuple in *RD:BOOK* needs to be modified, since there exist an attribute such that an update vertex connects it and it connects a *RD:BOOK* vertex. The algorithm evaluates the tuple that needs to be modified: it is identified by the primary key equal *book* attribute from an update. Then, it increments the number of sold books in a tuple.

The interesting example is when a new book is added, and the category counter has to be recomputed. New tuples in *RD:BOOK* and *CF:BOOK* are

added. The *KV:COUNT* projection also needs to be modified. The algorithm encounters two traces and constructs the strong paths corresponding to them. The first one is the following sequence of vertices:

$$(Map(U), book.id, book.category, category.id, \pi) \tag{21}$$

where $Map(U)$ represents the update vertex, that adds a new book, and π is the *KV:COUNT* vertex. The second one is very similar, however it goes once through the cycle between *id* and *parent* attribute in a category relation:

$$(Map(U), book.id, book.category, category.id, category.parent, category.id, \pi) \tag{22}$$

Having two paths, the algorithm travels through them and collects the attribute values and, as a result, recovers two primary keys of the tuples in *KV:COUNT*. In the first case, the primary key equals value of a *category* attribute. In the latter, when reaching *parent* attribute in a *category* relation, the algorithm queries the *KV:CAT* to retrieve a *parent* value, which is the primary key. Having the primary keys, the algorithm increments counters in both tuples: in the first case we increment the number of books in the given category, while in the latter we increment the number of books assigned to children nodes. This can be done due to traces of projection attributes.

7 Conclusion

The performance of the presented system relies on mapping different update operations into the same update vertex, thus we can precompute the steps of the algorithm for each update type. This is especially efficient for web applications, since users perform the same actions on the site, and similar database queries are run. As the number of different update types can grow exponentially, with the number of attributes, the implementation of the algorithm can be optimized to precompute an update of each attribute separately. The update propagation mechanism is scalable since the graph size does not depend on data size.

We have recently finished an implementation of the algorithm and we are preparing benchmarks to assess the overhead introduced by the propagation system. The presented mechanism is stateless. It can be set up on several servers and it does not introduce a new bottleneck in the system.

According to the CAP theorem [3], there exists a trade-off between consistency and availability. Some storages like relational databases provide ACID properties but do not scale well. The other example are possibly inconsistent NoSQL storages that provide high availability of the storage. We believe that our system allows tuning the trade-off in a better way. With our model, a storage can be easily split into smaller modules and each of them can be provided with a custom solution.

Creating a scalable database storage is a valid research problem. We have focused on web applications which gave us additional assumptions about data model. The read and update operations are known in advance, and data accesses

are read dominant. Several consistency levels are needed in different contexts. We believe that these assumptions lead to a better trade-off.

We have presented the scalable joint storage system which is based on several underlying storages, and propagates updates to keep all data copies consistent with each other. We have shown the architecture and described basic implementation assumptions of the constructed model. The update propagator algorithm has been described in details. The idea of the joint storage based on the underlying storages allows to take advantages of different architecture that suit best a specific data.

We believe that it allows building a scalable web application at the lower cost, because it eliminates the risk of programming faults affecting the data consistency that are difficult to fix and detect.

References

1. D. Agrawal, A. E. Abbadi, S. Antony, and S. Das. Data management challenges in cloud computing infrastructures. In S. Kikuchi, S. Sachdeva, and S. Bhalla, editors, *DNIS*, volume 5999 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2010.
2. D. D. Arun Iyengar, James R. Challenger and P. Dantzig. High-performance web site design techniques. *IEEE Internet Computing*, 4:17–26, 2000.
3. E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.
4. Cassandra. http://cassandra.apache.org/, Jan. 2011.
5. J. R. Challenger, P. Dantzig, A. Iyengar, M. S. Squillante, and L. Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Trans. Netw.*, 12:233–246, April 2004.
6. J. R. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. 1999.
7. C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable consistency management for web database caches. computer science. Technical report, 2006.
8. C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable query result caching for web applications. *Proc. VLDB Endow.*, 1:550–561, August 2008.
9. D. Kossmann, T. Kraska, S. Loesing, S. Merkli, R. Mittal, and F. Pfaffhauser. Cloudy: A modular cloud storage system. *PVLDB*, 3(2):1533–1536, 2010.
10. T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.
11. Lucene. http://lucene.apache.org, Jan. 2011.
12. A. Manjhi, P. B. Gibbons, A. Ailamaki, C. Garrod, B. M. Maggs, T. Mowry, C. Olston, A. Tomasic, and H. Yu. Invalidation clues for database scalability services. Technical report, In Proceedings of the 23 rd International Conference on Data Engineering, 2006.
13. MemcacheDB. http://memcachedb.org/, Jan. 2011.
14. Redis. http://code.google.com/p/redis/, Jan. 2011.
15. Sphinx. http://sphinxsearch.com, Jan. 2011.
16. Thrift. http://thrift.apache.org/, Jan. 2011.