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 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.

Keywords: multi storage, scalability, key-value storage, column family storage, 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 the number of users grows, the database becomes the bottleneck of the whole system. All other components 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 the database bottleneck is well-recognized by the industry. Although many fixes have been proposed, the general solution is still unknown. When a database workload increases, it is a common practice to split the database into smaller parts, and distribute it onto several servers. However, this is rather an ad-hoc fix and not a scalable architecture. 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 S3, Amazon SimpleDB, or others. Whatever solution has been chosen, the database is eventually split into several smaller instances running on different storage engines and servers. This however, makes the overall system architecture more complicated. Thus the application gets harder to maintain and the whole development process is more expensive. Sometimes the same data are 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 that 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 a data source occurs, our system modifies data in other storages, if it is needed. Figure 1 shows its architecture. The proper update propagation on underlying storages allows constructing a scalable joint storage with all advantages of underlying storages. The paper makes the following contributions. (1) We suggest a novel architecture to build several storages 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 a motivating example. Section 4 presents the data model. Section 5 defines the dependency graph used by the algorithm presented in Section 6. Section 7 summarises experimental results. Section 8 concludes.

2. Related work

Several publications address scalability and consistency. 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. Authors of [19] present their predictions about the future of multidatabase systems (MDBMS). According to them, further development and research will focus on strong heterogeneity and autonomy of data sources. Heterogeneity and autonomy introduce the consistency problem. On the other hand, they are critical for web data integration. The authors claim that some restrictions of functionality, like required simplicity of queries, will be needed to solve the problem. Such limitation are also assumed in our research. We precisely define restrictions to the data model. These restrictions allow constructing a scalable joint storage.

An interesting, ongoing research is a modular cloud storage system called Cloudy [10]. It is 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
The general problem can be described as keeping data consistent in different storages. We have examined possible solutions for maintaining materialized views which are a similar problem [20]. We focused on FlexViews [9] that implement materialized views within MySQL database based on the results described in [14, 16]. The application of materialized views is limited in our context as they can be maintained only within a single database. Additionally our system allows storing data in non-relational databases with limited (when compared to SQL) data access methods. The most important difference between solutions based on materialized views and our proposal is that in our system there is no master copy of data. FlexViews rely on applying changes that have been written to the change log. RDBMS change log is a single point of failure that is not acceptable in scalable solutions.

Another similar problem is consistent caching, i.e. 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 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 the number of data objects has increased and multiplied by the number of users. According to these observations, the graph size can grow rapidly and the method becomes impractical. In our approach the dependency graph has vertices that represent updates and retrievals. The complexity of our algorithm depends only on the number of columns.

3. Motivating example

Assume a simple bookstore platform that allows listing, searching, and buying books. Additionally each book has a list of readers’ opinions displayed on its info page. The database of the presented sample application needs to store: book information, users’ opinions on books, and information about sold items and users who bought them. Figure 2 depicts the data model of the bookstore. One can identify most common usage scenarios. Users search books using displace criteria. They browse through result pages and possibly full-text search items. When a book’s page is loaded, the system retrieves the information
When creating a scalable and efficient architecture, several different storages could be used in order to achieve better performance. Indexing engines like Sphinx [17] 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 [15] 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.

We have done tests in the bookstore scenario that explains why it is worth building a system with a number of different storages. We have used MySQL, PostgreSQL, Solr and Redis in our comparison. We started from full text searches. InnoDB of MySQL does not support text indexes, while PostgreSQL does. However the results are significantly worse than in Solr. For the duration of one minute the test query searched for books having comments with a given text phrase. PostgreSQL has finished 5 requests while at the same time Solr accomplished 232 requests. Figure 3 shows the results of two further scenarios. We tested a query that given the primary key of a book returns the number of sold copies. We compared relational databases that run a count selection against Redis that contained the counts in a key-value storage. This shows that, if common queries are known in advance, it is worth storing their results in specific storages. The update propagator allows denormalising and makes sure that derived data is up to date. As the result the caching layer can be managed by the update propagator since it assures consistency. The diagram on the right side of Figure 3 compares performance of a query which returns user data: the name and address. We can see a significant advantage of Redis.

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. Data model

Suppose our data consists of \( k \) relations: \( R_1, R_2, \ldots, R_k \). We assume that each relation has exactly one primary key element \( \text{id} \). This means that for each relation \( R_i(\text{id}, r_{i,1}, r_{i,2}, r_{i,3}, \ldots, r_{i,n_i}) \), the functional
dependency $id \rightarrow r_{i,1}, r_{i,2}, r_{i,3}, \ldots, r_{i,n_i}$ is satisfied: One-to-many associations between relations $R$ and $S$ are denoted by $R \prec_{r_i} S$. 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 to be 3NF. Additionally for any two relations $R$ and $S$, we say that they are associated, $R \bowtie S$, if there exist relations $S_1, S_2, \ldots, S_i$ and attributes $r_1, r_2, \ldots, r_{i+1}$ such that: $R \prec r_1 S_1 \prec r_2 S_2 \prec \cdots \prec r_i S_i \prec r_{i+1} S$. One-to-many associations between attributes of the same relation, $R \prec_{r_i} R$, are useful to represent hierarchical data.

4.1. Write operations

We put some restrictions on updates and retrievals. We assume that each update modifies a single tuple specified by $id$ parameter. We distinguish three types of write operations: adding a new tuple, editing a tuple attributes’ except for $id$ and deleting it. In general case, an update can be represented as $(R_U, type, value_id, \{(r_{i,1}, value_{r_{i,1}}), \ldots, (r_{j,1}, value_{r_{j,1}})\})$. Changes are applied to relation $R_U$. 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 the attributes to be changed, and their new values. The list of attributes and values remains empty for deletions. The tuple is determined by $value_id$ as for updates.

4.2. Views stored in underlying storages

Underlying storages maintain views. In this Section we describe types of views allowed in our system. Suppose $R$ is a relation and $S = (S_1, S_2, \ldots, S_i)$ is a sequence of relations associated with $R$, i.e. $R \bowtie S_1 \bowtie S_2 \bowtie \cdots \bowtie S_i$. 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 $R \prec_{r_{j,1}} S_{j,1} \prec_{r_{j,2}} S_{j,2} \prec_{r_{j,3}} \cdots \prec_{r_{j,k}} S_j$. Then we define $R_{S_j} = S_{j,1} \bowtie_{S_{j,1}.id=r_{j,1}} S_{j,2} \bowtie_{S_{j,2}.id=r_{j,2}} \cdots \bowtie_{S_{j,k}.id=r_{j,k}} S_j$. Suppose $r_1, r_2, \ldots, r_i$ are attributes of $R, R_{S_1}, R_{S_2}, \ldots, R_{S_i}$. We allow projections of the form $\pi_{r.id, r_1, \ldots, r_i} (R \bowtie_{R.id=r_1} R_{S_1} \bowtie_{R.id=r_2} R_{S_2} \bowtie_{R.id=r_3} \cdots \bowtie_{R.id=r_i} R_{S_i})$. 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 the primary relation of the projection.

Since we allow \( R \bowtie S \) and \( S \bowtie R \), it is possible that a safe projection outputs the same attribute of a relation several times. As an example, suppose an employee relation with the manager column which is a self join. When projecting an employee’s name and employee’s manager name, we project the same attribute twice. According to this, more than one projected attribute may correspond to the same relation attribute. We call them projection attributes in contrast to relation attributes. If a projection attribute \( r_p \) projects a relation attribute \( r \), then we say \( r_p \) is a projection attribute of \( r \) and we denote it as \( \kappa(r_p) = r \).

Given a safe projection \( U \) and a projected attribute \( r_p \), we introduce the trace of \( r_p \) in \( U \) which determines how \( r_p \) is projected. Suppose a projection attribute \( r_p \) is projected from a relation: \( R_{t_1} \bowtie R_{t_1}; id=t_2 \bowtie \cdots \bowtie R_{t_m}; id=t_m \), where \( t_1, t_2, \ldots, t_m \) are attributes of relations \( R_{t_1}, R_{t_2}, \ldots, R_{t_m} \) respectively. Additionally \( R = R_{t_1} \) is the primary relation of \( U \), and \( \kappa(r_p) \) equals \( t_m \). Then \( tr(U, r_p) \) is defined as:

\[
tr(U, r_p) = (R_{t_1}; id, R_{t_1}; t_1, R_{t_1}; id, R_{t_1}; t_2, \ldots, R_{t_m}; t_m)
\]  

(1)

Since we allow a single relation attribute to be projected several times, we need to take care of how it is projected. The function \( tr \) allows recovering the chain of joins that has been applied to project an attribute. For each projection, we introduce the set of attribute traces that contains all pairs of relation attribute and their traces:

\[
Trace(U) = \{(r_p, s) : tr(U, r_p) = s\}
\]  

(2)

The system also allows some simple aggregations on the views, like count or sum. For this purpose we define two types of selections: safely updatable and incrementally updatable. Let \( apply(U, k) \) denote the operation of applying changes specified in \( U \) into a tuple \( k \) in a safe projection. When \( U \) adds a new tuple to a projection, \( k \) is empty.

**Definition 4.1. (Safely updatable selection)**

Function \( f \) is safely updatable iff. for each update \( U \), \( f(apply(U, k)) \) can be computed from \( f(k) \) and \( U \).

An example of such operation is count. If we know the counter of a one to many relation, we can recompute it after data changes. If a row is added or deleted, we respectively increment or decrement it. Modifying other attributes does not change the counter. A sum is not such a selection, since given the sum and an update 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. There are not many safely updateable selections. Thus we introduce incrementally updatable selections:

**Definition 4.2. (Incrementally updatable selection)**

Function \( f \) is incrementally updatable iff. for each update \( U \), which adds a new tuple, \( f(apply(U, k)) \) can be computed from \( f(k) \) and \( U \).

Let us now reinvestigate the case of sum operation. As stated before, it is not safely updatable however it is incrementally updatable. As long as we only add new elements, a modification of a sum is a simple addition of a value stored in a new element. Another example, which is mentioned later, is incremental concatenation of strings. In our model we allow safely updatable and incrementally updatable selections when selected data is added in an incremental manner without any modifications nor removals. We have implemented count, sum and incremental concatenation operators. We also allow a simple selection of attributes which selects attributes’ values from associated relations.
When allowing different selection types, we have to ensure that at least one underlying storage contains original values of each attribute. For that purpose we denote \( \text{id\_f} \) as an identical function of attributes’ values, thus \( \text{id\_f} \) is safely updatable. Let \( \text{Attr}(\Pi) \) denote a set of projected attributes of a projection \( \Pi \) and let \( \text{Sel}(\Pi) \) denote a set of pairs of the form \( \{(r_1, f_1), (r_2, f_2), \ldots \} \) where the first element of each pair is an attribute and the latter is a safely updatable or incrementally updatable function. Based on a presented notation we introduce a complete projections’ set. We require projections in underlying storages to constitute a complete projections’ set:

**Definition 4.3. (Projections’ set completeness)**
A set of projections \( P \) is complete iff. for each relation \( R \) in a schema and for each projection attribute \( r_p \) it holds that \( \exists \Pi \in P \) (\( r_p \in \text{Attr}(\Pi) \land (r_p, \text{id\_f}) \in \text{Sel}(\Pi) \)).

## 5. Dependency graph

A dependency graph \( G \) is a triple \((V, E_{\text{strong}}, E_{\text{weak}})\) where \( V \) is the set of vertices, and \( E_{\text{strong}}, E_{\text{weak}} \) are sets of directed edges, which are called strong and weak edges respectively. Two vertices cannot be connected with both a strong and a weak edge at the same time.

Let \( A = \text{Attr}(R_1) \cup \text{Attr}(R_2) \cup \cdots \cup \text{Attr}(R_k) \) be the set of all attributes of all schema relations. \( \text{Attr}(R) \) is the set of attributes in relation \( R \). We distinguish attributes from different relations with the same name and consider them as separate elements of \( A \). Let \( P = \{P_1, P_2, P_3, \ldots \} \) be the set of all safe projections stored in the underlying storages.

For data modifications as defined in Section 4.1 we introduce the function \( \text{Map}(U) \):

\[
\text{Map}(U) = (R_U, \text{type}, \{r_i, \ldots, r_j\})
\]

It maps a write operation so that two updates that perform the same operation on the same attributes are treated as the same entity. Next we define \( M = \{\text{Map}(U_1), \text{Map}(U_2), \text{Map}(U_3), \ldots \} \) as the set of values of \( \text{Map} \) for all data modifications. Then the set of vertices \( V \) of the dependency graph is the union \( A \cup P \cup M \).

Next we define the edges of \( G \). For each \( R \prec_r S \), the foreign key \( S.r \) is connected by a strong edge with the primary key of its relation \( S.\text{id} \) and with the primary key of the foreign relation \( R.\text{id} \). Thus, \( \{(S.\text{id}, S.r), (S.r, R.\text{id})\} \in E_{\text{strong}} \).

Each projection vertex \( \Pi \) 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, i.e. \( (R.\text{id}, \Pi) \in E_{\text{strong}} \). Each projected attribute \( r \) is connected by weak edges with \( \Pi \): \( (r, \Pi) \in E_{\text{weak}} \).

Next we define edges connecting update vertices. Given a vertex \( \text{Map}(U) \), it is connected by a strong edge with \( R.\text{id} \), i.e. \( (\text{Map}(U), R_U.\text{id}) \in E_{\text{strong}} \) and by weak edges with all modified attributes: \( \forall_{i=1,\ldots,j}(\text{Map}(U), R_U.r_i) \in E_{\text{weak}} \). This ends the definition of the dependency graphs. An example \( G \) is shown on Figure 4.

## 6. The propagator algorithm

In this section we describe the problem in a formal way and we suggest an algorithm as a solution. We also prove the correctness of the algorithm. The problem can be specified as follows. Suppose
a data model as defined in section 4 where data is stored in different data storages. When an update request occurs, the system needs to apply it to the underlying storages. 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 6.1. (Data Consistency Problem–DCP)**

Suppose a system with projections $P_1, P_2, \ldots, P_j$ containing data in a state $T_1, T_2, \ldots, T_j$. An update $U$ changes a tuple in some relation and modifies a state of projections into $T'_1, T'_2, \ldots, T'_j$, 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 underlying storages, $n = |P_1| + |P_2| + \cdots + |P_j|$. Let $m = |V| + |E_{weak}| + |E_{strong}|$. Thus $m$ corresponds to the size of the dependency graph. It represents the complexity of the data schema. The complexity of algorithms realizing DCP is a function of $n$ and $m$. In this paper we present an algorithm whose complexity is independent on $n$. This assures the scalability since the complexity of the propagator does not depend on the data size.

### 6.1. Underlying storages’ drivers

First we describe functions implemented in underlying storages’ drivers that are used by the algorithm. Given an update $U$ and a safe projection $\Pi$, we distinguish two functions that add a new tuple:

$$\text{addPrimary}(\Pi, \{(r_{p_1}, value_{r_{p_1}}), \ldots, (r_{p_i}, value_{r_{p_i}})\})$$  \hspace{1cm} (4)

$$\text{add}(\Pi, value_{id}, \{(r_{p_1}, value_{r_{p_1}}), \ldots, (r_{p_i}, value_{r_{p_i}})\})$$  \hspace{1cm} (5)
The only difference is that \texttt{addPrimary} returns the primary key of the new tuple. On the other hand \texttt{add} inserts a tuple with a specified primary key. The \texttt{modify} function updates a single tuple in the projection \( \Pi \) with the primary key equal \( \text{value}_i \) with values specified in a form of a list, containing a projection attribute and its value: \( \text{modify}(\Pi, \text{value}_i, \{ (r_1, \text{value}_{r_1}), \ldots, (r_n, \text{value}_{r_n}) \}) \). Our algorithm uses the \texttt{retrieve}(\( \Pi \), \( r_p \), \( \text{value}_i \)) function that returns the value of a projection attribute \( r_p \) from a tuple in a projection \( \Pi \) with the primary key equal \( \text{value}_i \). The last function needed is \texttt{delete}(\( \Pi \), \( \text{value}_i \)). It removes a specified tuple from a projection. We assume those functions to be implemented in drivers of underlying storages’. We do not restrict to any database types nor vendors, and only assume a few basic functions that need to be implemented by a storage.

\section{Data identification function}

For a given relation attribute \( r \), the following set contains all projections that store values of \( r \):

\[
\{ (\Pi, r_p) : \Pi \in P \land r_p \in \text{Attr}(\Pi) \land \kappa(r_p) = r \land (r_p, \text{id}f) \in \text{Sel}(\Pi) \}
\]

\( \text{Sel}(\Pi) \) and \( \text{Attr}(\Pi) \) has been defined in Section 4.2. We need a function that indicates the projection to retrieve a relational attribute from. We introduce \( \text{Data} \) as such a function. \( \text{Data}(r) \) is defined as an arbitrary element of this set. In Section 4.2 we assumed that \( P \) is complete. Thus \( \text{Data}(r) \) is non-empty for each \( r \).

In further sections we elaborate on the optimal choice of \( \text{Data}(r) \) element. The function allows identifying the storage where the given attribute is contained. Given a pair \( (\Pi, r_p) \), \texttt{retrieve}(\( \Pi \), \( r_p \), \( \text{value}_i \)) returns the value of attribute \( r \) in the tuple with the primary key equal \( \text{value}_i \).

In the previous section we have mentioned the \texttt{addPrimary} function. When a new tuple is added, changes are first applied via that function on the primary projection and we denote the projection as \( \text{Prim}(R) \). \( \text{Prim}(R) \) can be an arbitrary element of the set \( \{ \Pi \in P : (R, \text{id}, \Pi) \in E_{\text{str}} \} \).

\section{Detecting modified data}

Let \( A \) denote the set of attributes in a schema and \( P \) describe the set of projections. We define \( \text{Proj}(U) = \{ \Pi \in P : \exists r \in A(\text{Map}(U), r) \in E \land (r, \Pi) \in E \} \) as the function that returns all projections that make use of an attribute updated when applying changes of \( U \). In other words, it contains all projections that are affected by \( U \).

\section{Strong edges’ path}

Let \( R_U \) denote the relation modified by an update \( U \) and let \( R_{\Pi} \) denote the primary relation of the projection \( \Pi \). Here we describe a function \texttt{Path}(\( U \), \( \Pi \), \( r \)) which given an update \( U \), a projection \( \Pi \) and a relation attribute \( r \) from \( R_U \), finds strong edges’ paths from \( \text{Map}(U) \) to \( \Pi \). The returned path is based on the trace of \( \Pi \), and determines a chain of joins used to project an attribute \( r \).

According to the structure of the dependency graph, update and projection vertices can only connect attribute vertices. We investigate elements of \( \text{Trace}(\Pi) \) that has been defined in (2). Suppose a projection attribute \( r_p \) and a relation attribute \( r \) such that \( \kappa(r_p) = r \). Given \( r_p \), we investigate \( t = \text{tr}(\Pi, r_p) \) such that \( (r_p, t) \in \text{Trace}(\Pi) \). According to the definition (1), \( \text{tr}(\Pi, r_p) \) contains a sequence of primary
and foreign key attributes, from the primary key of the projection to the attribute $r$:

$$\{ (R_{t_1}.id, R_{t_1}.t_1, R_{t_1}.id, R_{t_1}.t_2, \ldots, R_{t_m}.t_m) \}$$

(7)

where $R_{t_1}$ equals the primary relation of $\Pi$, denoted by $R_{1\Pi}$, and $R_{t_m}$ is the relation containing an attribute $r$, and $R_{t_m}$ equals $R_U$. From the definition of the $tr$, we know that $t_1, t_2, \ldots, t_m$ are the foreign key attributes which implies:

$$\{ (R_{t_2}.t_2, R_{t_1}.id), (R_{t_3}.t_3, R_{t_2}.id), \ldots, (R_{t_m}.t_m, R_{t_{m-1}}.id) \} \subseteq E_{strong}$$

(8)

Additionally, from the construction of the graph, we know that:

$$\{ (R_{t_2}.id, R_{t_2}.t_2), \ldots, (R_{t_m}.id, R_{t_m}.t_m) \} \subseteq E_{strong}$$

(9)

since all foreign keys are connected by a strong edge with the primary key of the same relation. Since $R_{t_1} = R_{1\Pi}$ and $R_{t_m} = R_U$, the following statement follows:

$$\{ (Map(U), R_U.id), (R_{1\Pi}.id, \Pi) \} \subseteq E_{strong}$$

(10)

This leads to the sequence of vertices $SP(t) = (Map(U), R_{t_m}.id, R_{t_{m-1}}.t_m, \ldots, R_{t_1}.id, \Pi)$, which is a valid strong edge path between $Map(U)$ and $\Pi$. According to this, the $Path(U, \Pi, r)$ function can be defined as:

$$Path(U, \Pi, r) = \{ SP(t) : \exists r_p \in Attr(\Pi) \forall (r_p, t) \in Trace(\Pi) \}$$

(11)

We also introduce a function $A(U, \Pi, p)$, that gathers all attributes within a projection that are updated by $U$ such that their trace corresponds to the strong edge path $p$. Assume a trace $t$ such that $SP(t)$ equals $p$. Then $A(U, \Pi, p)$ can be defined as:

$$A(U, \Pi, p) = \{ r_p \in Attr(\Pi) : (Map(U), \kappa(r_p)) \in E \land (r_p, t) \in Trace(\Pi) \}$$

(12)

We introduce a function $Join(p)$ that, given a strong edge path $p$, is defined as:

$$Join(p) = \begin{cases} 
0, & p \text{ has exactly 3 vertices} \\
1, & p \text{ has exactly more than 3 vertices} 
\end{cases}$$

(13)

This simple function is quite useful. Suppose a projection attribute $r_p$ of $\Pi$ with the strong edge path, corresponding to $tr(\Pi, r_p)$, equal $p$. The $Join(p)$ function determines if $r_p$ has been projected in $\Pi$ via the chain of joins or not. If the path has 3 vertices, it can only contain: an update vertex, the primary key of the modified relation and a projection vertex. Thus there is no one-to-many association applied, which happens when the path is has more than 3 vertices.

### 6.5. Tuple identification

We define a function $Find(U, \Pi)$ that identifies modified tuples in a projection. Let us assume $U = (S, type, value_id, Val)$ is an update where $Val = \{(r_i, value_{r_i}), \ldots, (r_j, value_{r_j})\}$ as previously defined. Suppose $r$ is an attribute from $\{r_1, \ldots, r_j\}$ and let us focus on a single element of $Path(U, \Pi, r)$
from (11). Given values of \( U \), \( \text{Find}(U, \Pi) \) returns primary keys of modified tuples in \( \Pi \). Let \( AV_t \) denote a value of an attribute \( t_i \) in the modified tuple and suppose \( AV_{t_0} = AV_{R_i.id} = value_{id} \). Suppose \( t_i \) and \( t_{i+1} \) are attributes of relations \( R_i \) and \( R_{i+1} \), respectively. Then \( AV_{t_{i+1}} \) is determined as follows:

\[
AV_{t_{i+1}} = \begin{cases} 
\text{retrieve}(\text{Data}(t_{i+1}), AV_t), & R_t = R_{t_{i+1}} \\
AV_t, & R_t \neq R_{t_{i+1}} 
\end{cases}
\]  

(14)

We simply iterate through the attributes of path from \( Path(U, \Pi) \), and evaluate values of the joined tuple attributes until we retrieve the attribute \( R_i.id \). This is possible since we iterate through strong edges, and in case of connecting attributes, strong edge connects either an attribute with the primary key attribute of the same relation or foreign key attribute with the primary key of the associated relation. As a result \( AV_{R_i.id} = AV_{t_0} \).

We have constructed a function, that given an element \( p \) from \( Path(U, \Pi, r) \) and the primary key of the updated tuple, returns the primary key of the modified tuple corresponding to a path of strong edges between \( \text{Map}(U) \) and \( \Pi \). We denote that function \( g(U, p) \). When finding all modified tuples, an algorithm examines all paths \( Path(U, \Pi, r) \) of all modified attributes \( r \). As a simple remark from the dependency graph construction, \( U \) modifies an attribute \( r \) when \( (\text{Map}(U), r) \in E \). In general \( \text{Find}(U, \Pi) \) returns the maximal set that contains pairs of strong edge paths between update and projection vertices associated with the primary key of the column that has to be modified:

\[
\text{Find}(U, \Pi) = \{(g(U, p), p) : \exists_r (\text{Map}(U), r) \in E \land p \in Path(U, \Pi, r)\}
\]  

(15)

### 6.6. Data modifications

Assume an update \( U = (R_U, \text{type}, value_{id}, Val) \). Then we construct a set \( Val(\Pi, U, p) \) as:

\[
Val(\Pi, U, p) = \{(r_i, value_{r_i}) : r_i \in A(\Pi, U, p) \land (\kappa(r_i), value_{r_i}) \in Val\}
\]  

(16)

The \( Val(\Pi, U, p) \) contains pairs of projection attributes and values. Projection attributes correspond to the attributes from \( Val \) that affect the projection \( \pi \). The last function we present is a \( \text{Mod} \) function:

\[
\text{Mod}(Val(\Pi, U, p), \Pi, p, value_{id})
\]  

(17)

which modifies data in underlying storages. The function modifies a tuple in a projection \( \Pi \) with the primary key equal \( value_{id} \). The tuple is modified according to \( Val(\Pi, U, p) \) which consists of pairs containing attribute and value. These pairs define attributes that are going to be modified and values that have been given in \( U \). The parameter \( p \) is a strong edge path, which is required for proper data modification. As an example, suppose employee table with the self-join on manager’s column and a projection which projects employee name and the manager’s name. Let \( U \) modifies a manager’s name. Then the \( \text{Mod} \) function is applied on several tuples in the projection: on the manager’s tuple and on subordinates. When applying changes of \( U \) on a tuple, a strong path is required to modify proper column: employee name or managers name.

We have assumed in Section 4.2 that each projected attribute can store values which are processed by \textit{safely updateable} or \textit{incrementally updateable} selections. Thus we can easily evaluate a new value of a tuple in \( \Pi \) and this is done in \( \text{Mod} \). The direct implementation depends on a selection type. As an example, in the case of \textit{count} we increment the value when \( U \) adds a tuple, decrement in case of tuple removal or leave it unchanged if modified.
6.7. The algorithm

Having all necessary functions presented we show the whole algorithm based on the predefined functions. Assume an update $U$ equals $(R_U, \text{type}, \text{value}_{id}, \text{Val})$. Then:

1. If $U$ is add, let $\Pi = \text{Prim}(U)$ and let $p$ be the strong edge path $p$ containing $(\text{Map}(U), R_U.id, \Pi)$. Apply $\text{addPrimary}(\Pi, \text{Val}(\Pi, U, p))$ and append its result to values of $U$ as the 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 $T = \text{Proj}(U) \setminus \text{Prim}(U)$, in other case let $T = \text{Proj}(U)$.

3. For each $\Pi \in T$:
   
   3.1 For each $(\text{value}_{\Pi.id}, p) \in \text{Find}(U, \Pi)$:
      
      3.1.1 If $\text{Join}(p) = 0$, then apply $\text{add}(\Pi, \text{value}_{\Pi.id}, \text{Val}(\Pi, U, p))$, $\text{delete}(\Pi, \text{value}_{\Pi.id})$ or $\text{modify}(\Pi, \text{value}_{\Pi.id}, \text{Val}(\Pi, U, p))$ according to the type value.
      
      3.1.2 If $\text{Join}(p) = 1$, then apply $\text{Mod}(\text{Val}(\Pi, U, p), \Pi, p, \text{value}_{\Pi.id})$.

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 tuples modified in each projection. Modified tuples are represented as elements of $\text{Find}(U, \Pi)$ and contain the primary key of the modified tuple and a strong edge path corresponding to the trace of modified attributes.

Step 3.1.1 describes the simple case when the algorithm fills underlying storages with modified values. This happens when $\Pi$ contains a subset of attributes of relation $R$ which is modified by $U$. The algorithm runs the requested operation on a tuple in underlying storage Step 3.1.2 applies data changes on tuples that via one-to-many joins contain data that is affected by $U$. The changes are applied by the $\text{Mod}$ function.

6.8. Correctness

In this section we prove the correctness of the presented algorithm. Let $\Pi$ denote an arbitrary projection and $U$ an update that occurs. Additionally assume $T$ denotes a state of $\Pi$ before an update, and $T'$ a state after $U$ is performed. As previously, let $F(U, T)$ denote the propagator algorithm function which applies $U$. The purpose of this section is to prove $F(U, T) = T'$. We divide the section into two parts. Firstly we show that the algorithm properly detects tuples that require modification, secondly we elaborate on applying changes on a single tuple. We start with the following observation:

**Lemma 6.2.** An update $U$ modifies data in a projection $\Pi$ iff. $\Pi \in \text{Proj}(U)$

**Proof:**

Assume $\Pi \in \text{Proj}(U)$. According to the definition of $\text{Proj}$ (cf. Section 6.3), there exists an attribute $r$ such that $(r, \Pi) \in E$ and $(\text{Map}(U), r) \in E$. The existence of the edge $(r, \Pi) \in E$ implies that $\Pi$ projects the attribute $r$, while the edge $(\text{Map}(U), r)$ implies that $U$ modifies a value of the attribute $r$, and as a conclusion $U$ modifies $\Pi$. 
Now suppose $U$ modifies $\Pi$. Then $U$ has to modify some attribute $r$ that is projected by $\Pi$. $U$ modifies $r$ in some tuple, then $(\text{Map}(U), r) \in E$. Additionally, since $\Pi$ projects $r$, $(r, \Pi) \in E$. As a result $\Pi \in \text{Proj}(U)$, which ends the proof.

We are going to prove that $\text{Find}$ returns modified tuples or nothing, when given projection has not been updated. We start with an evaluation of the $g(U, p)$ function. The strong edge path $p$ determines the sequence of joins used in a projection. Assume $t_1, t_2, \ldots, t_m$ are the foreign key attributes in $p$ and a relation $R'$ is constructed as: $R_{t_1} \times R_{t_1}.id=t_2 \times R_{t_2}.id=t_2 \times \cdots \times R_{t_{m-1}}.id=t_{m-1} \times R_{t_m}$ where $R_{t_1} = R_{\Pi}$, and $R_{t_m} = R_U$.

**Lemma 6.3.** Let $U = (R_U, \text{type}, \text{value}_{id}, \text{Val})$ be an update and $p$ be a strong edge path between $\text{Map}(U)$ and $\Pi$. Then $g(U, p) = \pi_{R_{t_1}.id}(\sigma_{R_U.id=\text{value}_{id}}(R'))$.

**Proof:**
We have defined a $g(U, p)$ as a function which traverses a strong edge path $p$ and gathers the primary keys of the traversed relations. We start the traversal with the primary key $\text{value}_{id}$ of the updated relation. Given an attribute $p_i$ and its value $AV_{p_i}$, the function evaluates $AV_{p_i+1}$. When $p_i$ and $p_{i+1}$ belong to different relations $AV_{p_i} = AV_{p_{i+1}}$, since those vertices represent two attributes from join between the relations. On the other hand, if $p_i$ and $p_{i+1}$ belong to the same relation, then $p_i$ is the primary key and the algorithm reads the value of $p_{i+1}$ from some projection. In general, in terms of relation $R'$, evaluating following $AV_{p_{i+1}}$ values can be described as: $\pi_{p_{i+1}}(\sigma_{p_i=AV_p(R')})$. In our data model we allow only one-to-many joins, between relations. According to this $\sigma_{p_{i+1}=AV_{p_{i+1}}}(R') \subset \sigma_{p_i=AV_{p_i}}(R')$. This leads us to: $g(U, p) \in \pi_{p_{i+1}}(\sigma_{R_{\Pi}.t_m=\text{value}_{id}}(R'))$. The last thing, we have to prove, is that $\sigma_{R_{\Pi}.t_m=\text{value}_{id}}(R')$ selects a single tuple from $R'$. $R'$ is constructed from relations $R_{t_1}, R_{t_2}, \ldots, R_{t_m}$ such that: $R_{t_1} \land R_{t_2} \land R_{t_3} \land \cdots \land R_{t_m}$. According to this $R_{t_m}.id$ constitutes a valid primary key of $R'$, and the examined selection returns a tuple, which is identified by the primary key. This ends the whole proof.

We split the proof into two parts: first we show that each tuple from $\Pi$, modified by $U$, is contained in pairs of $\text{Find}(U, \Pi)$. Then, we prove the opposite direction: each pair of $\text{Find}(U, \Pi)$ contains the primary key of some tuple in $\Pi$, that is modified by $U$.

**Lemma 6.4.** Suppose $k$ is a tuple in $\Pi$ with the primary key equal $\text{value}_k$. If $k$ is modified by an update $U$ of the form $(R_U, \text{type}, \text{value}_{id}, \text{Val})$, then $\text{value}_k$ is contained in some pair of $\text{Find}(U, \Pi)$.

**Proof:**
We show the construction of a strong edge path $p$ such that: $(\text{value}_k, p) \in \text{Find}(U, \Pi)$. Suppose $r_p$ is a projection attribute, which projects an attribute $r$, modified in $k$ by $U$. Such a $r_p$ exists, since $k$ is modified by $U$. This also implies the existence of a sequence of foreign key attributes $t = (t_1, t_2, \ldots, t_m)$, such that: $R' = R_{t_1} \times R_{t_1}.id=t_2 \times R_{t_2} \times R_{t_{m-1}}.id=t_{m-1} \times R_{t_m}$ is a relation with an attribute $r$, while $R_{t_1} = R_{\Pi}$ and $R_{t_m} = R_U$. According to the definition (2): $(r_p, t) \in \text{Trace}(\Pi)$ and the definition (11) follows: $t \in \text{Path}(U, \Pi, r)$. Based on the definition (15) of $\text{Find}$, we know that: $(g(U, p), p) \in \text{Find}(U, \Pi)$ which, according to the lemma 6.3, is equivalent to: $(\sigma_{R_{t_1}.id}(\sigma_{R_{t_m}.id=\text{value}_{id}}(R')), p) \in \text{Find}(U, \Pi)$. A value of an attribute $r_p$ from $R'$ is modified in $k$ by $U$. According to this: $\text{value}_k \in \Pi_{R_{t_1}.id}(\sigma_{R_{t_m}.id=\text{value}_{id}}(R'))$. From the proof of the lemma 6.3, we know that $\Pi_{R_{t_1}.id}(\sigma_{R_{t_m}.id=\text{value}_{id}}(R'))$
Lemma 6.5. Assume a strong edge path \( p \), an update \( U = (R_U, type, value_{id}, Val) \), and a projection \( \Pi \). If \((value_k, p) \in Find(U, \Pi)\), then a tuple \( k \) in \( \Pi \), with the primary key \( value_k \), has been modified by \( U \).

Proof:
According to the definition (15), there exists an attribute \( r \) such that \((Map(U), r) \in E\) and \( p \in Path(U, \Pi, r)\). This ensures, from the definition (11), the existence of the projection attribute \( r_p \) such that \((r_p, t)\) is contained in \(Trace(\Pi)\), for some trace \( t\). Assume \( t_1, t_2, \ldots, t_m \) are the foreign key attributes in \( t\). We construct a sequence of relations \( R_{t_1}, R_{t_2}, \ldots, R_{t_m} \) such that: \( R' = R_{t_1} \bowtie_{R_{t_1}.id = t_2} R_{t_2} \boweq \cdot \cdot \cdot \boweq_{R_{t_{m-1}}.id = t_m} R_{t_m}\). Again, \( R_{t_1} = R_{\Pi} \) is the primary relation of \( \Pi \), and \( R_{t_m} = R_{U} \) is an updated relation. Due to definition (2) of \( Trace\), \( r_p \) is a projection attribute from \( R'\) in \( \Pi \). Additionally the existence of an edge \((Map(U), r)\) implies that an attribute \( r_p \) is modified in some tuple of \( R'\). The primary key of that tuple equals \( \pi_{R_{t_1}.id}(\sigma_{R_{t_1}.id=value_{id}}(R'))\), since \( R_{t_m} \) constitutes the primary key of \( R'\). Based on the lemma 6.3, this assures that it is equal \( g(p) \) and, as a consequence, it equals \( value_k \). This ends the proof, since \( r_p \) is the attribute modified in the tuple \( k \) by the update \( U \). \( \square \)

Next we are going to prove that data modifications, applied by the algorithm, are correct. We assume that an update \( U \) modifies a tuple \( k \) in a projection \( \Pi \). We denote by \( T \) a state of \( \Pi \) before an update, and by \( S_{\Pi}(T, U) \) a state of the projection after performing \( U \) on an abstract data model. Additionally assume \( F_{\Pi} \) is a function of \( U \) and \( T \), which returns a state of \( \Pi \) after applying \( U \) by the presented algorithm.

Let us focus now on the \( Val(\Pi, U, p) \) function defined in (16). Assume an update \( U \) that affects an attribute \( r_p \) of a tuple \( k \) from a projection \( \Pi \). We start with a remark:

Lemma 6.6. Let \( p \) be a strong edge path corresponding to the \( tr(\Pi, r_p) \) and let \( value_{\kappa(r_p)} \) be a value of \( U \) used to modify \( r_p \) in \( k \). Then: \((\kappa(r_p), value_{\kappa(r_p)}) \in Val(\Pi, U, p) \iff (r_p, value_{\kappa(r_p)}) \in Val(\Pi, U, p)\)

Proof:
First, we assume \((r_p, value_{\kappa(r_p)}) \in Val(\Pi, U, p)\). The leftward implication follows from Definition 16 of \( Val(\Pi, U, p)\). Second, we investigate the opposite direction. Assume then \((\kappa(r_p), value_{\kappa(r_p)}) \) is contained in \( Val \). Based on (16), we need to prove that \( \kappa(r_p) \in A(U, \Pi, p) \), which has been defined in (12). The update \( U \) contains an attribute \( \kappa(r_p) \) in its values, thus \((Map(U), \kappa(r_p)) \in E\). We have also assumed that \( p \) is a valid strong edge path between \( Map(U) \) and \( \Pi \). This assures that for a trace \( t \), corresponding to \( p \), \((r_p, t) \in Trace(\Pi)\) which ends the proof. \( \square \)

We are going to distinguish two possible cases: the length of \( p \) is 3, or more. This is equivalent to \( Join(U, \Pi) \) equal 0 and 1 respectively. The further sections elaborate on those cases and the correctness in each of them is proven separately.

Lemma 6.7. If \( Join(U, \Pi) \) equals 0, then \( F_{\Pi}(U, T) = T' \).

Proof:
When \( Join(U, \Pi) \) is equal to 0, then \( U \) modifies the primary relation of \( \Pi \). According to the type of \( U \), an algorithm does the following operation:
1. In case of adding a new tuple, the tuple is added to $\Pi$. A tuple $k$ is filled with values from $Val(\Pi, U, p)$. The attributes of $\Pi$, that are missing values in $U$, are filled with default values.

2. In case of deleting a tuple, $k$ is deleted from $\Pi$.

3. In case of updating an existing tuple, values of $Val(\Pi, U, p)$ overwrite some values in $k$.

In the first and the last case we update a tuple with a subset of values from $U$. In the second case we simply delete the tuple. While the seconds case is self-explanatory, the correctness of the other steps relies on the lemma (6.6). The lemma assures that updating tuple with values of $Val(\Pi, U, p)$, which contains a subset of values from $U$, applies the same changes. When compared to values of $U$, $Val(\Pi, U, p)$ maps attribute values to projection attributes and contains a subset of values from $U$ for attributes that are present in $\Pi$.

Lemma 6.8. If $Join(U, \Pi)$ equals 1, then $F_{\Pi}(U, T) = T'$.

Proof:
Let $R_{\Pi}$ be the primary relation of a projection $\Pi$ and $R_U$ is a relation modified by $U$. The $Join(U, \Pi)$ implies $R_{\Pi} \neq R_U$. The update $U$ modifies attributes that are projected in $\Pi$ via some chain of one-to-many joins. In this case a tuple in $\Pi$ is always modified, even if $U$ adds or deletes some tuple in $S$. The data modification depends on the implementation of the $Mod$ function, $Mod(Val(\Pi, U, p), \Pi, p, value_id)$. Parameters $\Pi$ and $value_id$ identify a projection and the primary key of a tuple that is going to be changed. The other two arguments provide data for the proper modification. The sequence $p$ determines how the update $U$ influences the projection $\Pi$, while $U$ contains the type of operation and new attributes’ values.

The projection $\Pi$ can store data transformed by a safely updatable selection, or by a incrementally updatable selection, while the second case only allows adding a new tuple. According to the definition 4.1. and the definition 4.2., a selection function that selects a projection attribute $r_p$ from a relation attribute $r$ has to fulfill the following criteria: if an update occurs a new value of $r_p$ can be recomputed from a former value of $r_p$ and a value of $r$ in $U$. Although the implementation of $Mod$ method depends on the selection function, its correctness is proven and relies on the definitions of safely updatable and incrementally updatable selections.

Theorem 6.9. Assume an update $U = (R_U, type, value_id, Val)$. For each projection $\Pi$, the equation $F_{\Pi}(U, T) = T'$ holds.

Proof:
The proof of the theorem relies on the former lemmata. Suppose a projection $\Pi$. The algorithm updates data in $\Pi$ iff. $\Pi \in Proj(U)$. This is correct, due to the lemma 6.2. According to lemmata 6.4. and 6.5., $value_k$ is the primary key of some tuple and $value_k$ is contained in a pair of $Find(U, \Pi)$ iff. the tuple is modified by $U$. As a result, if the algorithm evaluates $Find(U, \Pi)$, then the primary keys of all modified tuples are identified.

At this stage, we have found all tuples that need to be updated. Each tuple is identified by the primary key, and the algorithm is aware of a trace. The trace allows us to determine how the update affects the tuples. The algorithm modifies found tuples according to two different cases: $Join(U, \Pi)$ equal 0 or 1. The lemmata 6.7. and 6.8. prove the correctness of applied update changes in both those cases.

The algorithm starts with identifying all tuples that require invalidation. Then the changes on tuples are applied due to a selection type. We have proven the correctness of both steps, thus proving the correctness of the whole algorithm.
6.9. A Dependency graph example

Figure 5 shows the dependency graph for the bookstore example (cf. Section 6.9). 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 \((Map(U), book.id, book.category, category.id, \pi)\) where \(Map(U)\) represents the update vertex, that adds a new book, and \(\pi\) 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: \((Map(U), book.id, book.category, category.id, category.parent, category.id, \pi)\). 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. Experimental results

We have implemented the update propagator as a web service using Java and Thrift [18]. Thrift is a software framework for scalable cross-language services development.

We tested several scenarios in the bookstore example (cf. Section 3). First, we assumed there was only one relation \textit{book} in the schema and only one its projection in PostgreSQL. We add new tuples to the relation and test the overhead introduced by propagator web service layer. Figure 6 presents the results. It shows the total time of client request to the propagator compared to the time consumed by PostgreSQL. This allows assessing the overhead for different workloads. The result is satisfactory. The overhead of the propagator seems independent of the workload and remains at the acceptable level.

The second test focuses on the offset between updating the primary storage and secondary storages. We added new books into the bookstore. We assumed that financial data, like prices and stock were placed in PostgreSQL. That was the primary projection of the relation. Additionally these data were also

![The offset of the propagator](image1)

**Figure 6.** The results of the experiment on the overhead of the propagator. The gap between the curves is the additional time above the PostgreSQL operation needed when using the propagator.

![The offset of data updates in Redis](image2)

**Figure 7.** The offset between updating data in the primary storage and in secondary storages. The bottom curve represents time spent in the primary storage PostgreSQL. The middle curve includes the offset introduced by the propagator. The upper curve presents the total time till all changes appeared in secondary storage.
stored in MongoDB, while Redis kept the count of books for each category. Figure 7 shows the results. Again we can observe that the tested offset did not grow when the workload increased. Note that on the functional level this is the same test as the first one. Although storing the same data in different storages introduced extra workload, the system run more than 12000 operations per second, i.e. more than in the first test where only the PostgreSQL database was used.

8. Conclusion

According to the CAP theorem [3], there exists a trade-off between consistency and availability. Some storages like relational databases provide ACID but do not scale well. Possibly inconsistent NoSQL storages offer high availability. We believe that our system allows tuning this trade-off in a better way. Within our model, data can be cheaply split into smaller chunks put into custom storages for better performance.

Creating a scalable database storage is a valid research problem. We have focused on web applications with additional assumptions: (1) read and update operations are known in advance, and (2) retrievals are dominant. Several consistency levels are needed in different contexts.

We have presented the scalable joint storage system based on several underlying storages that propagates updates to keep all data copies consistent with each other. We have shown the architecture and described basic implementation assumptions. The update propagator algorithm has been described in details and proven to be correct. The idea of the joint storage allows taking advantage of different architectures that suit best specific data.

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

References


