University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Paweł Leszczyński

An update propagator for joint scalable storage

PhD dissertation

Supervisor

dr hab. Krzysztof Stencel

Institute of Mathematics
University of Warsaw

June 2012

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

June 12, 2012 . . . .

date *Paweł Leszczyński*

Supervisor's declaration:
the dissertation is ready to be reviewed

June 12, 2012 . . . .

date *dr hab. Krzysztof Stencel*

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. We also apply this algorithm for consistent object caching, which reduces database workload and makes web application perform significantly better. Next, we describe *PropScale*, i.e. a proof-of-concept implementation of the proposed algorithm. Using this system we have conducted experimental evaluation of our solution. The results prove its robustness.

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

ACM Computing Classification System: H.2.4. (Distributed databases)

Table of Contents

1 Introduction 5
2 Motivating Example 11
3 Alternatives 17
  3.1 Disadvantages of relational databases 17
  3.2 Classification of NoSQL storages 18
4 Academic proposals 23
5 The Update Propagator Algorithm 29
  5.1 Data architecture 29
  5.2 Dependency graph 32
  5.3 Data consistency problem – DCP 34
  5.4 Underlying storages' drivers 35
  5.5 The algorithm 36
  5.6 A Dependency Graph Example 41
  5.7 Correctness 43
  5.8 Complexity 50
6 Consistent Caching 53
  6.1 Motivating example—a community forum application . 53
  6.2 Existing caching solutions 54
  6.3 The dependency graph for consistent caching 58
  6.4 Experimental results 63
  6.5 Analysis 65
7 PropScale: an update propagator service for a joint storage 67
  7.1 System architecture 67
  7.2 Synchronization and Multithreading 71
8 The benefits of applying PropScale 73
  8.1 Introduced overhead 73
  8.2 Offset between updates in storages 73
  8.3 PropScale for cloud integration 74
  8.4 Custom statistics 75
9 Conclusion 81

1 Introduction

Modern web applications provide users with a significant number of interactive and personal features. This trend is called *Web 2.0* era of web applications. It has started around 2000. Before it, websites were used just to retrieve information. However human beings are not only consumers. They have also a need to produce and share data. Social network applications are a profound example. Furthermore, *Web 2.0* extends to a model, in which web applications are personalized and each user sees the content dedicated to him/her. For instance, in e–commerce every user sees its own basket, the history of bought products, etc. In contrast to the former model of web applications, *Web 2.0* introduces several performance issues. The size of stored data is multiplied by the number of users. When a website grows, it can reach several millions of users or even hundreds of millions, as some most successful applications do. Moreover, as the interaction level between a user and the application increases, numerous queries to a database are to be run. This makes applications data-intensive. This brief introduction explains that the technologies, that have been popular years ago, do not fit into current requirements and there is a need for a new architecture.

As the number of users grows, the database becomes a bottleneck of the whole system. All the other components of a system scale well, and can be easily distributed on several machines 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. Buying a new and better hardware does not solve the problem. As an example, having a medium class PC, one can buy a hardware that has twice as much computing power. However, when demanding ten times more computing power, the single machine with a better hardware costs much more than ten times the price of medium PC. Such a solution is also rather short-sighted since there are no machine being one million times faster. This approach is called *vertical scaling*. The opposite option is *horizontal scaling* where new nodes of approximately the same power are added to a system.

The most common architecture of a web application consists of application servers which receive requests from users and send responses according to the application logic and data from backend databases. Horizontal scaling of application servers is rather simple as they do not have to share any information with each other and a request can be operated by a single node. This can be easily done with a load balancer. However it is not so easy in case of the database, especially in the classic ACID transactional model. The problem of the database bottleneck is well-recognized in the industry, and many fixes have been proposed. However, the general solution is still unknown. When the database workload increases, it is a common practice to split the database into smaller parts, and distribute it onto more than one server. The most frequent approach to do so is partitioning or sharding. Partitioning corresponds to dividing the database schema into smaller schemata and store data on separate nodes. The first disadvantage of this approach is the performance of single queries that are run on different nodes, e.g. a join in a relational database where joined tables are stored by different nodes. However the main disadvantage is that, it is still not a horizontal scaling as a number of logical partitions of the schema is not flexible and rather limited.

Sharding splits tables and distributes them among servers. Sharding a relational database is also a non trivial problem. The benefits of sharding are clear when a query retrieves data from a single node. In order to do so, one has to determine which shard contains the requested data. Doing it on the database side adds a new bottleneck to the system. On the other hand, determining the shard on the client side, requires clients to change when new nodes are added. Sharding still does not scale well. When n shards contain too much data or serving too much queries, adding an empty shard does not improve the situation. When migrating data to a new shard, queries that performed well previously may suffer a performance loss. This can be mitigated for *key-value* storages by serving single values over a simple API. However, it is a hard problem for relational databases since they have rich query language and some queries perform badly for most shard distribution rules.

Partitioning and sharding relational databases does not build a scalable architecture and is 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 PaaS platforms (Platform as a Service) like Amazon S3, Amazon SimpleDB, or others. NoSQL storages are described in the next section. This term corresponds to a number of different non-relational databases that have proven to perform better that a RDBMS in specific scenarios. However they do not perform well in the general case and cannot be used as the single storage of the whole system. The external platforms' storages scale well but in most usage scenarios they are combined with local storages that store sensitive data.

According to this, whatever the solution has been chosen, the database is going to be 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 the application harder to maintain. As a result, the whole development process gets more expensive. Sometimes the same data is stored in several locations, and the application's logic needs to keep the replicated data in the consistent state. Thus developers must take care of all data writes and apply them carefully on several storages. When dealing with big applications, this can lead to errors which are hard to detect and repair. Assume an e–commerce platform with data distributed among several storages. One storage contains information about users while the other stores info about the bought products. The programmer A implements a module for listing users' bought products and needs to retrieve the users' names to display on the list. This cannot be done in an efficient manner by means of a single query in both storages. Thus, the programmer decides to store redundant users' names within the data on bought products. The module has been implemented and it works for several months. Then a programmer B is asked to implement an option to change a user's name on the platform. However B is unaware of redundancy of users' names introduced by A, and when a user's name is modified consistency issues between data stores arise. As this bug is detected, it is assigned to A since his functionality ceased to work. A is then wondering what could have happened, since it has worked for some time before the bug has been detected. This bug is difficult to avoid in the future, since A could not have known about module of B. Moreover it is hard to repair since A may have no idea about

changes of B. Such errors are very expensive. The research presented in this thesis focuses on integrating multiple storages into one joint storage while taking automatically care of data consistency issues. We aim at eliminating abovementioned errors. The system should handle proper data updates in different storages on its own.

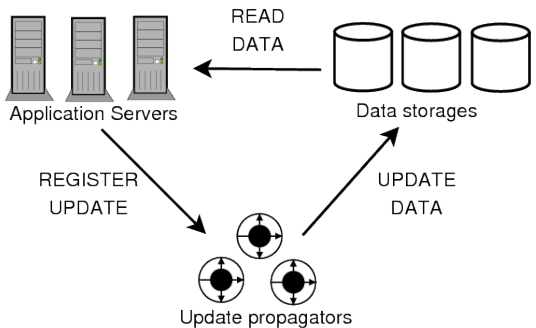

Fig. 1. The update propagator architecture

In this thesis, we describe a research on novel data propagation algorithm for joint storages that maintains replicated data in multiple sources in the consistent state. When an update on one source occurs, our system modifies data in other storages, if it is needed. Figure 1 shows the architecture of such a 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 thesis makes the following contributions.

- We suggest a novel architecture for building several storage systems into a system.
- We present the update propagation algorithm for keeping data in the consistent state.
- We prove the correctness of this algorithm.
- We describe how this algorithm can be customized and applied for consistent caching of objects.
- We present *Propscale*, i.e. a system built on the top of the update propagation algorithm, and benchmark it in real-life scenarios.

The rest of the thesis is organized as follows. Chapter 2 presents a motivating example for the research. In Chapter 3 we describe

potential drawbacks of relational databases and present a brief introduction to NoSQL storages and explain why they may be useful in our construction. Chapter 4 focuses on the existing academic proposals and the ongoing research. Chapter 5 describes the propagator algorithm. It presents data model assumptions, necessary auxiliary data structures and their role in proper update propagation. We show how the algorithm works in sample scenarios. In this chapter we also prove the correctness of the propagator algorithm. Chapter 6 provides information how the system can be used to maintain a consistent caching layer. Chapter 7 describes implementation details of the system, while Chapter 8 presents experimental results. First we focus on estimating the overhead introduced by the additional update layer of our system. Then we present the performance improvement achieved by applying our system and other benefits of using it. Chapter 9 concludes.

2 Motivating Example

Let us now consider a 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 its data model.

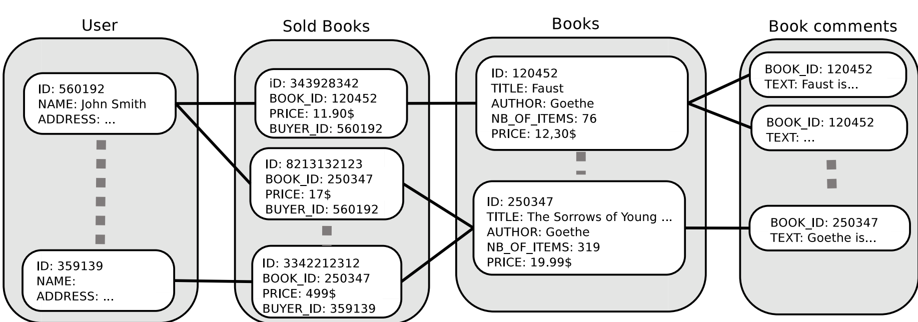

Fig. 2. The simple bookstore: book information, comments, sold items and users' data

The database schema consists of the following relations:

- *book*: It stores general book information like the title, the author and the primary key. Additionally it contains the current price and the number of items available.
- *user*: It stores user data like the first and the last name, the address and the primary key.
- *book_sold*: It associates users with books they have bought. Except for users' and books' primary keys, it also contains the price of the book used in this transaction.
- *book_comment*: It stores comments and opinions that relate to a specific book. Each row consists of the book's primary key and the comment text.

Now we describe characteristics of data accesses in the bookstore application. According to [9], e-commerce web applications have high browse-to-buy ratios. This means that they are read dominant. People browse and search for products they are interested in. They read

products' reviews frequently but the frequency of buying a product or putting a comment is much lower.

One can identify the most common queries that are performed on the platform. Most of them can be detected before application deployment. 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 the database to adjust the number of available items.

The bookstore platform queries a backend data store to retrieve the number of available products. This information is shown on each book's site. Moreover, it is used when validating an order. It allows checking if there are enough items in the stock. Although retrieving the same data, these two separate actions require different consistency levels. When selling a product, we have to retrieve the exact value. Otherwise, the system may end up with selling an unavailable product which can be a serious problem. On the other hand, displaying a product page with an inaccurate number of stock items is not crucial. Therefore, the same data in different situations require different levels of precision.

The simple bookstore is expected to grow rapidly. Its architecture has to scale well and we focus on the database layer. Then the popular question arises: what storage would suit best our system's needs? In the era of *NoSQL* the natural answer is none. There is no single database that will fit best into all parts of the system. Instead, there are probably few different storages that fit best into specific parts of a system.

The search functionality is business critical in our bookstore. In order to achieve better performance, we can use indexing engines like Sphinx [3] or Solr [59] (enterprise search platform based on Lucene [37]). One of these can be used, since they outperform RDBMS in full-text searches. The other advantage of using Solr is that it allows a rich configuration of the ranking algorithm which results in more accurate results.

The number of book comments in the bookstore can grow rapidly, as it is a content generated by users. One may want to store it in a distributed column family storage like Cassandra [46]. Product information is accessed each time a product page is generated. This

requires a highly available storage with fast data access. Since a single request gets single product data, it can be stored in key-value storage. Its simple data structure offers best performance and highest availability.

The component for storing financial data is surely crucial, as it contains information on sold items and payments. In that case, it may be a business requirement to store it in RDBMS, as it provides different transaction levels that ensure necessary 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 rest of this thesis, we show methods how to architect such a system and most notable, how to preserve the required level of consistency among various storage components.

The problem of building a consistent storage on top of other storages is hard. In our research we are going to focus on storages used by web applications. The analysis of the bookstore has given us some hints that may be valuable in further research. First, web applications are read dominant. As an implication, if we construct a system with a better read performance, it will significantly increase the overall performance even if the writes take longer. The second hint is that the same data is required with different consistency levels in different scenarios. This allows replicating the same data over databases even if a replication is asynchronous. If accurate results are needed, the master database will be queried. The third hint is that data access patterns are known before the system deployment. Frequently accessed data can be put into highly available storages.

We have experimented on that. Some results for the bookstore scenario follows. In the tests we have used MySQL, PostgreSQL, Solr and Redis. We start from full-text searches and queries through book data and comments. InnoDB of MySQL does not support text indexes. 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 these counts in a key-

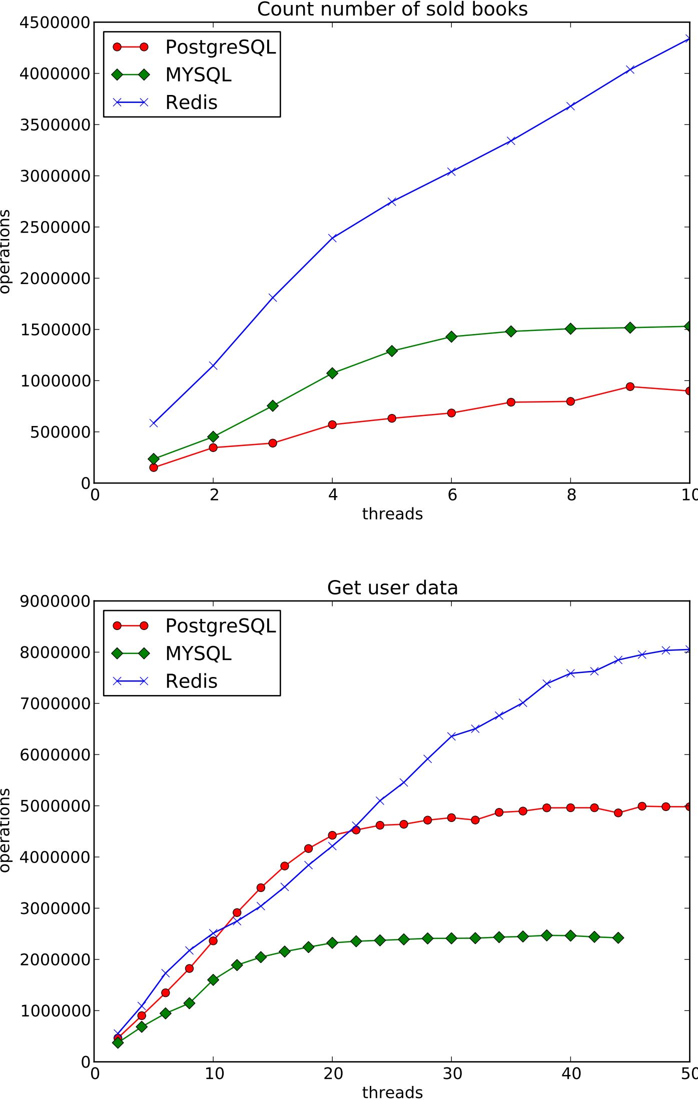

Fig. 3. The graphs show the number of accomplished operations with respect to the number of client threads. The first one concerns the query that given a book counts the number of sold copies. The second one relates to the query that retrieves user data. During these tests each 5ms another thread modified the queried data.

value storage. Obviously if common queries are known in advance, it is worth storing their results in specific storages. However, the update propagator allows automatic denormalisation and makes sure that derived data are up to date. As the result the caching layer can be managed by the update propagator since it assures consistency. The first diagram of Figure 3 compares performance of a query which returns user data: the name and address. We can see a significant advantage of Redis.

Storing data in multiple storages with different architectures is not a benefit of the research presented in this thesis. One can easily implement it manually. Our goal is bigger. We want a scheme to integrate storages into a system that *itself* takes care of storing data in the consistent state. A method to build such a system is the subject of this thesis.

3 Alternatives

3.1 Disadvantages of relational databases

In this section we describe drawbacks of existing relational databases management systems (RDBMS). Our motivation is based on [61, 62, 60]. Popular RDBMS reproduce the construction of System R from the 1970s. At that time hardware was completely different than today. At the end of 1970s a large machine had around a megabyte of main memory. Currently CPU's are thousands of times faster and memories are thousands of times larger. However bandwidth between disk and memory did not increase at the comparable pace. This have changed the overall system architecture. Although the hardware characteristics changed dramatically, RDBMS did not [61].

Michael Stonebraker predicted in those papers the end of *"One size fits all"* as a commercial relational DBMS paradigm. According to him, the last few decades of commercial DBMS can be summed up with this phrase. It is true that the same products were used in completely different environments with different characteristics of data accesses. In our research we restrict to web applications and we focus on such access patterns. As a web application has to react within milliseconds, its database schema has to be optimized for short response time. Web applications are read-dominant. Overwhelming majority of operations are read accesses. Additionally write operations are rather simple, as each user buys a limited number of products at once, writes a single post, etc. Most RDBMS vendors deliver a system with rich SQL syntax that allows abundance of features that are not used in the context of web applications. There is nothing wrong about not exploiting all possible features of a product. However it is reasonable to investigate if the unnecessary functionality lowers the overall performance.

In RDBMS data and indices are stored on disks. Generally there is nothing wrong about that, except for the case where it may be useful to put a whole storage into RAM for faster accesses and occasionally snapshot it to a disk. Of course, in this scenario, we can lose data when a machine crashes with unsaved writes stored in RAM. However, some scenarios accept this risk in order to perform significantly better. Traditional RDBMS cannot do that as they were invented at the time RAM was small and expensive. This has changed

rapidly and now most server machines operate on dozens gigabytes of memory.

Multithreading is a great power of RDBMS. It allows to fully utilize CPU and disk resources. These features are implemented in every part of RDBMS including resource governor, concurrent B-trees, etc. The problem is that they cannot be switched off when unnecessary. Assume an application that accesses a database in the single threaded execution mode. All the concurrency features are irrelevant and slow the system down. The other thing, related to multithreading and concurrency control, is the duration of transactions. Some mechanisms have been designed to protect transactions that take up to several minutes. However they may not be necessary in case of OLTP systems, where a transaction takes less than one millisecond on a low-end machine.

One of the greatest real life concerns with RDBMS is data replication. High availability requires distributing data and transactions over several nodes. A proper replication level is required for a system to work normally when one of its parts crashes. This requires replicated data writes, that are registered to multiple nodes. This is not true for RDBMS. Replication is assured with logs that are sent with some delay. When an instance crashes one cannot switch to another without an access to latest logs. This may be crucial in case of hardware/network errors when the logs are not reachable.

3.2 Classification of NoSQL storages

The term *NoSQL* corresponds to databases that do not fit to the traditional RDBMS model. However it has no formal definition. Surely, the term represents a new movement in IT society. Some believe, this term means *NO SQL* and encourages avoiding relational databases at all. Other believe it means *Not only SQL* and they figure out specific scenarios where traditional relational database do not perform well. In some cases *NoSQL* storages outperform RDBMS by an order of magnitude. In this section, we describe the classification of NoSQL storages and applications they are suited for.

As mentioned before, *NoSQL* is not a single product or even a single technology. According to [63], the term represents the class of products and the collection of diverse concepts about data storage

and manipulation. The concept is not new, since the first storages for computing machines have also been non relational. The new thing is the reincarnation of the concept. There exists a strong need for solutions other than RDBMS to create scalable web applications.

The scalability problems has been first encountered by Google and several solutions have been applied. Google created a set of mechanisms to create an architecture which scales at every element of the system. This resulted in several publications. In order to create a fully scalable infrastructure Google has created a distributed file system [32], a column oriented family store [17], a MapReduce-based parallel algorithm execution environment [24] and a distributed coordination system Chubby [11]. In 2007 another great contributor to *NoSQL*, Amazon, presented ideas of its system Dynamo [25]. Dynamo is a distributed, highly available and eventually consistent data store.

As a web industry has been growing rapidly, several companies required fully scalable and highly available infrastructures. In turn, these required new methods to process large-scale data on a cluster of nodes. Here the MapReduce model has helped. It is a framework for processing distributive problems across huge data sets using a cluster of nodes. It consists of two steps: "Map" and "Reduce". During the "Map" step, the master node takes the input, partitions it up into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node. During the "Reduce" steps results are collected. The master node collects then the answers to the sub-problems and combines them to form the output - the answer to the problem it was originally trying to solve. As previously better performance can be achieved when reducing with several nodes with multiple reducers' levels. For instance having the sub-problems computed on 1000 nodes, we can use 100 nodes to reduce results from ten nodes each, then 10 nodes to reduce results of 100 nodes, and a single node which reduces results from 10 nodes at the end. The idea of MapReduce framework is to build a system that takes automatically care of data distribution across nodes, detecting absent nodes and other architectural issues. Within such

a system, writing a MapReduce program requires only to map the current problem into *Map* and *Reduce* functions.

These have been a great catalyst to the open source community. It created independent implementations of the concepts published by Google. One of the greatest contributor is the Hadoop project, which created a framework for building distributed applications on top of it. Several open source databases make use of them, for example Hive and H-Base data stores.

Column-Oriented Stores Data in RDBMS are stored with the row-oriented format. Each table is stored on a disk as a sequence of rows. All row data are located in the neighbourhood. The problems with this concept arise as we have a table with several columns whose value are mostly empty. If a database engine stores empty fields, then a lot of storage is wasted. If the value is marked to be empty then rows have different sizes and iterating over table rows gets slower.

In the column oriented model, a single row and its data can be thought as a set of key/value pairs. Each value is identified with the help of the primary identifier, often referred to as the primary key. Most column-oriented stores tend to call this primary key the row-key. Then the units of data are sorted and ordered on the basis of the row-key. Data is stored in a contiguous sequenced manner. As it grows and fills up one node, it is split into multiple nodes. It is sorted and ordered on each node, and also all nodes. This provides a large continuously sequenced set and makes read access by row-key very efficient, as the row-key is used to determine a node that contains data. When an update occurs, it is inserted at the end of the list. The model has been presented with publishing Google's BigTable concepts. An Apache open-source project HBase developed a database that makes use of the concept.

Key–Value storages Most key–value stores are based on a simple associative array structure. This simple data structure, when limited to accessing a single element of an array by its key, is very efficient. Single data access costs $O(1)$ running time. The key in an associative array is a unique identifier and it can be easily looked up to gain access to data.

The Oracle‘s Berkeley DB is considered to be a direct ancestor of most open source implementations as it has existed since the mid-1990s. There is a number of different key–value storage architecture scenarios. The greatest performance boost occurs when data is stored in memory and no disk operations are required to accomplish a request.

The frequent scenario to use key–value stores is a fast caching system that stores all data in memory. An example of such storages is Memcached [19], written by Brad Fitzpatrick for LiveJournal in 2003. It is an open-source, high-performance object caching system. The scenario is called as a *cache aside pattern*. It is an application's logic that writes data to a cache and invalidates it when necessary. Each time, data is needed, an application checks if it is present in a cache. If not, it queries it from other data sources and loads to a cache. When the cache fills up, some data is removed, e.g. using LRU (least recently used). This scenario has previously been a subject of our research and we elaborate on it in a chapter 6.2.

Several key–value storages implement persistency and store data on a disk. Redis [47] is a good example. The power of key–value storages is in keeping data in memory and avoiding slow disk accesses. In order to do so, Redis writes directly only to memory. Then the batch operation snapshots all data from memory to a disk. Of course, this makes possible losing data when server crashes without writing new data to the disk. However this is acceptable in some scenarios, where the stored data is not critical, e.g. the number of users who bought an item on an e–commerce platform. When such data is lost it can be simply recomputed. The other example is gathering statistics from user clicks on a page. If data are lost, we simply lose couple of seconds of user clicking stats. This can be sacrificed for performance. The configuration of Redis allows to set the frequency of snapshots to optimize a performance and minimize the possible data losses. The configuration allows setting the maximal number of seconds between snapshots or the number of writes needed to activate a batch script. This allows a conscious decision on how much data can be sacrificed.

Document storages In the context of *NoSQL* storages, the word *document* has nothing to do with business documents and document

management systems that everyone has on its own PC. According to [63], the term corresponds to loosely structured sets of key/value pairs in documents, which are typically JSON (JavaScript Object Notation). It has nothing in common with documents or spreadsheets. Although they, when properly structured, could be also put into such a storage.

The concept of document database relies on treating documents as a whole. This avoids configuring and defining their structure within a database, and can be done within an application logic. Moreover, the structure can change frequently which can be a serious problem for RDBMS. Schema changes on RDBMS may be painful. Document stores allow putting a diverse set of documents into a single collection. In addition to key–value stores, document databases index documents on its primary identifiers. They also allow defining indices on specified objects' properties. The most popular examples of such databases are MongoDB [18] and CouchDB [6].

Graph storages In the terms of Domain Driven Design, the previous three *NoSQL* groups are called Aggregate-Oriented databases. According to the definition from [26]: an aggregate is a cluster of associated objects that are treated as a unit for the purpose of data changes. The most illustrative example are document oriented storages. They store a whole document as a unit. It is different than most RDBMS where each column of a tuple can be retrieved or updated separately. With Aggregate–Oriented databases one can retrieve or update only a whole document. The great advantage of aggregates is that they create natural units for distribution strategies. On the other hand, Martin Fowler in [28] reveals some drawbacks of this approach. Although it gives a convenient data access when working with the aggregates, it can make serious issues when trying to look on the same data from different perspectives. For example, order entries in an e–commerce platform can be stored naturally as aggregates. However, it is not the best data structure for business analysis of product sales, etc. The interesting quotation of Martin Fowler fits here well: *The advantage of not using an aggregate structure in the database is that it allows you to slice and dice your data different ways for different audiences.*

Graph databases give a possibility to efficiently store and access data entities that are related. We describe an idea based on Neo4J [41], one of the most popular open source graph data stores. Data consists of nodes that can store properties and named relationships between them.

The interesting example is a database consisting of 1000 persons with each person having 50 friends on average. This is a simple graph. Assume a query that given two persons A and B, examines if there exists a path of length less than 4 between them. In the example, a query execution process can start with A. Then it simply traverses the graph until it reaches B or exceeds the limit of 4. The query execution becomes far more efficient than in RDBMS. Instead of the indices defined in RDBMS, the graph becomes an index on its own and allows fast data access. Such data structure facilitate queries that examine possible paths from a given node. In such cases, no matter how big the graph is, the problem becomes local and its performance depends on the number of edges per node instead of the total size of the graph as it would be in RDBMS.

4 Academic proposals

Several publications address scalability and consistency issues. The paper [1] describes design choices and principles for a scalable storage. According to the authors, main benefits of the cloud computing are elasticity, pay-as-you-go model of payment and the ability to run large scale environments on commodity hardware. It also reveals some potential drawbacks. The spectrum of data management systems has scalable key-value storages on one end, and transactional non-scalable relational databases on the other end. Providing efficient data management to the wide variety of applications in the cloud requires bridging this gap with systems that can provide various degrees of consistency and scalability. The authors address the problem of filling the gap between key-value and relational storage, which is investigated in our research. Authors of [65] 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. On one hand, 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 simplified queries, will be needed to solve the problem. This also happens in our research, as we precisely define assumptions and restrictions to the data model. This allows constructing scalable joint storages.

The trade-off between consistency and scalability is well known in the literature. It has been introduced for web services by Eric Brewer at PODC 2000 [10]. According to his CAP theorem, it is impossible for a web service to provide the following three guarantees: the consistency (C), the availability (A) and the partition-tolerance (P). Availability and partition-tolerance are the parameters which affect the system's scalability. On the other hand, consistency determines the expected behaviour of requests that modify data. The traditional approach is to expect a web service to behave in a transactional manner. A commit and an abort are atomic operations. Committed data are visible to all future transactions while uncommitted data are isolated from each other. These natural properties describe the expected behaviour of a web service. As proven in [33], these properties have to be relaxed in order to improve scalability. According to the authors of this paper, most real-world systems are forced to guarantee the proper behaviour *"most of the data, most of the time"*. This gives a possibility of unwanted application behaviour. Our construction gives the possibility to choose the best data store for different data in a system. This allows achieving transactional behaviour in components that require it (e.g. financial), while allowing other data to be kept in storages that relax the consistency for the high availability. We believe that a system operating on a joint storage is a better CAP trade-off.

The fundamental definitions of data consistency and dependency have been presented in [58]. Authors distinguish three data replication types: derived data, primary/secondary copies and interdependent databases. In the third case databases are peers that store interdependent data and this category is not examined in our research. Our system deals with two other categories and it is capable of proper data consistency maintenance in that cases. The paper also describes consistency models including eventual consistency and lagging consistency. However there is no single definition of eventual consistency in the community. The authors of that paper claim the

eventual consistency requires redundant copies to become consistent at certain time although it allows inconsistencies meanwhile. Lagging consistency assumes that one copy is up to date while updates to other copies may be delayed. According to [58], this is a degenerate case of eventual consistency as it does not imply that redundant copies are consistent at some point in time. On the other hand Vogels [66] defines the eventual consistency in a different way. According to him, it guarantees that if there are no new updates made to the object, eventually all copies will store the consistent value of the last update. In our research we construct a system that matches Vogels definition of eventual consistency. When an update occurs, our system applies changes on primary storages and returns a response to the client. Then the secondary storages are modified. This allows a short time interval of inconsistency of data sources. However after that, if no new update to the same values is issued, each copy will store the same value.

The article [45] categorizes consistency levels into: serializable, session consistency, adaptive, and mixed. Serializable corresponds to the 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. Such system is hosted on the Amazon platform. It is aware of the total hosting cost and the cost of a single failed transaction. When workload changes, the system modifies the consistency level to serve data at the lowest possible cost. This research addresses the same problem as ours, however, the cost comparison methodology restricts mainly to PaaS.

Several publications address the problem of transaction processing in a distributed database. The article [54] presents an interesting motivating example. Suppose two users from Germany and US on a social web application. Additionally assume their personal data is stored in different data centres: Europe and America. They meet in the real world and decide to add each other as a friend on the website. When such an update occurs, friends' list of both users in both data centres need to be updated. Many interesting questions arise here: what if one data centre is unavailable? Should the transaction revert or should it use a retry queue in such cases. These are not just technical issues as they affect not only the performance but also the

end user experience. Is it better for a user to get an error information or allow some inconsistencies: A sees B as a friend, while B does not? The problem is valid for all possible storage types and there is no general solution. Authors of [38, 2] claim that transactions that run across arbitrary data of a system are inefficient and suggest some restrictions. This can be e.g. schema sharding and a limitation to allow transactions to run on a single shard. In [23] authors present the system G-STORE, which allows multi–key transactions on key-value stores. Authors of [49] present *Deuteronomy*. This system is divided into two parts: the data component (DC) and the transactional component (TC) that manages transactions including concurrency control and undo/redo recovery. TC does not know anything about the location of physical data and operates on an abstraction layer. This is similar to our approach: we do have backend storages which are managed by an external update propagator. The update propagator is unaware of storage types and their data architectures as it operates only on simple drivers implemented for each storage.

The research [34] focuses on the problem of how to assess *"good enough"* for the consistency in cloud databases. Authors present an interesting idea of extending SQL queries in order to allow specifying the maximal accepted delay boundaries. Although we doubt, SQL is a proper query language for a distributed environment, the clear definitions of data inaccuracy may fit well into the industry needs. Authors of [21] build a system on top of multiple RDBMS nodes and provide automatic partitioning. However, we are aware of possible drawbacks of the automatic partitioning in case of RDBMS. First, it is not clear for a developer, how physical data are stored. This can lead to performance issues. RDBMS provide a rich query language and some queries can get extremely slow when running in the distributed environment. Moreover, with automatic partitioning there may be queries performing well on a centralized storage, and significantly slowing down for partitioned storage.

An interesting ongoing research is the modular cloud storage system called Cloudy [44]. It is built on 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 sim-

ple, and most modify a single record, while reads are more complex. Additionally, there are many NoSQL storages that often offer new versions. This makes storage internals difficult to maintain up to date. Furthermore, NoSQL storages provide plenty of API clients like JSON, XML, THRIFT [55] etc. Thus rewriting all of them is almost impossible. This issue is not present in the architecture on Figure 1 since we only care about proper update propagation.

The general problem can be described as keeping data consistent in different storages. This can be similar to maintaining materialized views [8], especially views updated incrementally [35, 36]. Incremental maintenance is important in our case as there is no master copy of data, and storages need to be updated once a write request occurs in a system. Additionally our views have to be self updatable. When an update to a database occurs, views can be updated properly based on their current state and update values, without the need to query the source database. Most algorithms for view maintenance have been designed for OLAP database and provide some modification of summary delta method rather than for real time OLTP processing. However, these are SQL procedures that rewrite updates issued to a system based on a materialized view definition. They provide a framework for deriving incremental view maintenance expressions which is not the solution to our problem. Although we create an abstract database schema, the backend storages are not required to provide SQL and most does not. In following chapters we define requirements for a storage driver to fit our construction. The requirements are much simpler than SQL. The simplicity allows constructing a single system based on storages with different architectures and data access methods

We examined *FlexViews* [27] that implement materialized views within MySQL database based on the results described in [53, 57]. The application of materialized views is limited in our context as they can be maintained only within a single database. 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 category of problems, that are similar, but not exactly the same, is consistent caching, i.e. the evaluation of invalidation clues of the cached data when an update on a data source occurs. Authors of [31, 30, 52] 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 [40, 16, 15]. 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 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 the dependency graph algorithm whose efficiency depends only on the number of columns.

Authors of [12] present a system that integrates real-time transactions (OLTP) and analytical operations (OLAP) within a single storage. Our system allows integrating OLAP and OLTP storages where each write is synchronously applied in OLTP. Asynchronous writes to OLAP storage assure the system does not slow down while keeping analytical data storages up to date.

The focus on the related work strongly motivates our research. Several publications state the gap between transactional relational database and scalable non-relational storages, as the important research problem. They also claim some restrictions to query languages is needed, when compared to SQL, to achieve the scalability. This will be presented in following chapters. We compared our research problem to maintaining materialized views and consistent caching, arguing why the problem requires a different approach. We have also described some other joint storages and compared it to our idea.

5 The Update Propagator Algorithm

5.1 Data architecture

Data model Suppose our data consists of k relations: R_1, R_2, ..., R_k. We assume that each relation has exactly one primary key element and we denote it as *id*. This means that for each relation

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

the functional dependency

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

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 R is associated with S, denoted by $R \lhd S$, if there exist relations $S_1, S_2, \ldots, S_i$ and attributes r_1, r_2, ..., r_{i+1} such that:

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

One-to-many associations between attributes of the same relation, $R \prec_r R$, are useful to represent e.g. hierarchical data.

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

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

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.

Underlying storages Next we are going to define data stored in underlying storages. Suppose R is a relation and $S = (S_1, S_2, \ldots, S_i)$ is a sequence of relations associated with R, i.e.

$$R \lhd S_1 \land R \lhd S_2 \land \cdots \land R \lhd S_i. \qquad (5)$$

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

$$R \prec_{r_{j_1}} S_{j,1} \prec_{r_{j_2}} S_{j,2} \prec_{r_{j_3}} \cdots \prec_{r_{j_k}} S_j. \qquad (6)$$

Then we define

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

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

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

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 *the primary relation* of the projection.

Since we allow $R \lhd S$ and $S \lhd R$, 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* Π and a projected attribute r_p, we introduce the trace of r_p in Π which determines how r_p is projected. Suppose a projection attribute r_p is projected from a join:

$$R_{t_1} \bowtie_{R_{t_1}.id=t_2} R_{t_2} \cdots \bowtie_{R_{t_{m-1}}.id=t_m} R_{t_m} \qquad (9)$$

where t_1, t_2, ..., t_m are attributes of relations R_{t_1}, R_{t_2}, ..., R_{t_m} respectively. Additionally $R = R_{t_1}$ is the primary relation of Π, and $\kappa(r_p)$ equals t_m. Then $tr(\Pi, r_p)$ is defined as:

$$tr(\Pi, r_p) = (R_{t_1}.id, R_{t_1}.t_1, R_{t_1}.id, R_{t_1}.t_2, \ldots, R_{t_m}.t_m) \tag{10}$$

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 Π, the function $Trace$ returns the set of all pairs composed of a relation attribute and its traces:

$$Trace(\Pi) = \{(r_p, s) : tr(\Pi, r_p) = s\} \tag{11}$$

The underlying storages can also contain processed results of *safe projections*. This can be a simple operation 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 an empty set.

Definition 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 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 the sum is the simple addition of the 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 id_f as the identity function on attributes' values. Obviously, id_f is *safely updatable*. Let $Attr(\Pi)$ denote the set of projected attributes of a projection Π and let $Sel(\Pi)$ denote the set of pairs of the form $\{(r_1, f_1), (r_2, f_2), \dots\}$ where the first element of each pair is an attribute and the second is a *safely updatable* or *incrementally updatable* function. Based on the presented notation we introduce complete projections' sets. We require projections in underlying storages to constitute a complete projections' set:

Definition 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*

$$\exists_{\Pi \in P}(r_p \in Attr(\Pi) \wedge (r_p, id_f) \in Sel(\Pi)). \tag{12}$$

5.2 Dependency graph

A dependency graph G is a triple $(V, E_{strong}, E_{weak})$ where V is the set of vertices, and E_{strong}, E_{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

$$A = Attr(R_1) \cup Attr(R_i) \cup \dots \cup Attr(R_k) \tag{13}$$

be the set of all attributes of all schema relations. $Attr(R)$ 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:

$$P = \{P_1, P_2, P_3, \dots\} \tag{14}$$

be the set of all safe projections stored in the underlying storages.

For data modifications, as defined in Chapter 5.1, we introduce the function $Map(U)$:

$$Map(U) = (R_U, type, \{r_i, \dots, r_j\}) \tag{15}$$

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

$$M = \{Map(U_1), Map(U_2), Map(U_3), \dots\} \tag{16}$$

as the set of values of 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 $S.id$, and with the primary key of the foreign relation $R.id$. Thus,

$$\{(S.id, S.r), (S.r, R.id)\} \in E_{strong}. \tag{17}$$

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

Next we define edges connecting update vertices. Given a vertex $Map(U)$, it is connected by a strong edge with $R.id$, i.e.

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

and by weak edges with all modified attributes:

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

This ends the definition of the dependency graphs. An example G is shown in Figure 4.

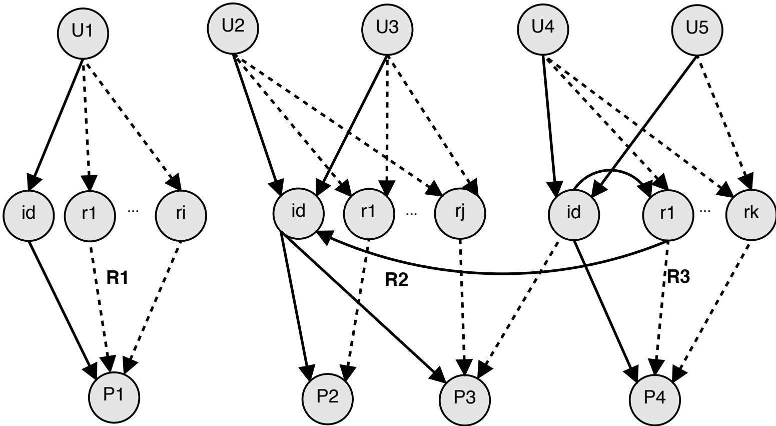

Fig. 4. The figure shows an example of the dependency graph. $U_1, \ldots, U_5$ denote update vertices, $R_1, \ldots, R_3$ denote schema relations and $P_1, \ldots, P_4$ projections stored in underlying systems. We assume $R_2 \prec_{R_3.r_1} R_3$.

5.3 Data consistency problem – DCP

In this section we define the problem in a formal way. Suppose a data model as defined in chapter 5.1, 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, e.g. adding a new tuple into a storage may cause invalidation of tuples in other storages. This intuitive description of the problem leads us to the following formal definition.

Definition 4 (Data Consistency Problem–DCP). *Suppose a system with projections $P_1, P_2, \ldots, P_j$ containing data in the states T_1, T_2, $\ldots$, T_j. An update U changes a tuple in some relation and modifies the states 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*

$$F(U, T_1, \ldots, T_j) = (T'_1, \ldots, T'_j). \tag{20}$$

Suppose n is a total number of tuples stored underlying storages,

$$n = |P_1| + |P_2| + \cdots + |P_j|, \tag{21}$$

and l is a total number of all traces' lengths of all projected attributes. Formally, we can define that as:

$$l = \sum_{\pi \in \{P_1, P2, \ldots, P_j\}} \sum_{(r_p, s) \in Trace(\pi)} |s| \tag{22}$$

Let m be

$$m = |V| + |E_{weak}| + |E_{strong}| + l, \tag{23}$$

Thus m corresponds to the size of the dependency graph and the structure of projections: number of projected attributes and chain of joins used to project them. In general, it represents the complexity of the data schema. The complexity of algorithms realizing DCP is a function of n and m. In this thesis 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.

5.4 Underlying storages' drivers

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

$$addPrimary(\Pi, \{(r_{p_1}, value_{r_{p_1}}), \ldots, (r_{p_i}, value_{r_{p_i}})\}) \tag{24}$$

$$add(\Pi, value_{id}, \{(r_{p_1}, value_{r_{p_1}}), \ldots, (r_{p_i}, value_{r_{p_i}})\}) \tag{25}$$

The only difference is that *addPrimary* returns the primary key of the new tuple. On the other hand *add* inserts a tuple with a specified primary key. The $modify$ function updates a single tuple in the projection Π with the primary key equal $value_{id}$ with values specified in a form of a list, containing a projection attribute and its value:

$$modify(\Pi, value_{id}, \{(r_{p_1}, value_{r_{p_1}}), \ldots, (r_{p_i}, value_{r_{p_i}})\}). \tag{26}$$

Our algorithm uses the

$$retrieve(\Pi, r_p, value_{id}) \tag{27}$$

function that returns the value of a projection attribute r_p from a tuple in a projection Π with the primary key equal $value_{id}$. The last function needed is

$$delete(\Pi, value_{id}). \tag{28}$$

It removes the specified tuple from a projection. We assume those five functions to be implemented in drivers of underlying storages. We do not restrict to any database types nor vendors. We only assume a few basic functions that need to be implemented by each storage system.

5.5 The algorithm

Data identification function The propagator algorithm sometimes has to retrieve the value of a given relation attribute. For an attribute r, the following set contains all projections that store values of r:

$$\{(\Pi, r_p) : \Pi \in P \wedge r_p \in Attr(\Pi) \wedge \kappa(r_p) = r \wedge (r_p, id_f) \in Sel(\Pi)\} \tag{29}$$

$Sel(\Pi)$ and $Attr(\Pi)$ has been defined in Chapter 5.1. We have to nominate one of the members of (29) to be used by our algorithm to retrieve the value of the attribute r. This choice is implemented by a function $Data(r)$ that returns an arbitrary element of (29). In Chapter 5.1 we assumed that P is complete. Thus $Data(r)$ is well defined for each r.

The function allows identifying the storage where the given attribute is contained. Given a pair (Π, r_p), $retrieve(\Pi, r_p, value_{id})$ returns the value of attribute r in the tuple with the primary key equal $value_{id}$.

We have already mentioned the *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 $Prim(R)$. $Prim(R)$ can be an arbitrary element of the set:

$$\{\Pi \in P : (R.id, \Pi) \in E_{strong}\} \tag{30}$$

Detecting modified data Let A denote the set of attributes in a schema and P describe the set of projections. We define $Proj(U)$ as

$$Proj(U) = \{\Pi \in P : \exists_{r \in A}(Map(U), r) \in E \wedge (r, \Pi) \in E\} \tag{31}$$

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.

Strong paths Let R_U denote the relation modified by an update U and let R_Π denote the primary relation of the projection Π. *A strong path* is composed solely of strong edges. Here we describe the function $Path(U, \Pi, r)$ which given an update U, a projection Π and a relation attribute r from R_U, finds strong paths from $Map(U)$ to Π. The returned path is based on the trace of Π, and determines the 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 $Trace(\Pi)$ that has been defined in (11). Suppose a projection attribute r_p and a relation attribute r such that $\kappa(r_p) = r$. Given r_p, we investigate $t = tr(\Pi, r_p)$ such that $(r_p, t) \in Trace(\Pi)$. According to the definition (10), $tr(\Pi, r_p)$ contains a sequence of primary and foreign key attributes, from the primary key of the projection to the attribute r:

$$(R_{t_1}.id, R_{t_1}.t_1, R_{t_1}.id, R_{t_1}.t_2, \ldots, R_{t_m}.t_m) \tag{32}$$

where R_{t_1} equals the primary relation of Π, denoted by R_Π, and R_{t_m} is the relation containing an attribute r. Additionally 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:

$$\{(R_{t_2}.t_2, R_{t_1}.id), (R_{t_3}.t_3, R_{t_2}.id), \ldots, (R_{t_m}.t_m, R_{t_{m-1}}.id)\} \subset E_{strong} \tag{33}$$

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

$$\{(R_{t_2}.id, R_{t_2}.t_2), \ldots, (R_{t_m}.id, R_{t_m}.t_m)\} \subset E_{strong} \tag{34}$$

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

$$\{(Map(U), R_U.id), (R_\Pi.id, \Pi)\} \subset E_{strong} \tag{35}$$

This leads to the sequence of vertices

$$SP(t) = (Map(U), R_{t_m}.id, R_{t_m}.t_m, \ldots, R_{t_1}.id, \mathit{\Pi}). \quad (36)$$

which is a valid strong path between $Map(U)$ and $\mathit{\Pi}$. According to this, the $Path(U, \mathit{\Pi}, r)$ function can be defined as:

$$Path(U, \mathit{\Pi}, r) = \{SP(t) : \exists_{r_p \in Attr(\mathit{\Pi})} \kappa(r_p) = r \wedge (r_p, t) \in Trace(\mathit{\Pi})\} \quad (37)$$

We also introduce a function $A(U, \mathit{\Pi}, p)$, 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 $A(U, \mathit{\Pi}, p)$ can be defined as:

$$\{r_p \in Attr(\mathit{\Pi}) : (Map(U), \kappa(r_p)) \in E \wedge (r_p, t) \in Trace(\mathit{\Pi})\} \quad (38)$$

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

$$Join(p) = \begin{cases} 0, & p \text{ has exactly 3 vertices} \\ 1, & p \text{ has exactly has more than 3 vertices} \end{cases} \quad (39)$$

This simple function is quite useful. Suppose a projection attribute r_p of $\mathit{\Pi}$ with the strong edge path, corresponding to $tr(\mathit{\Pi}, r_p)$, equal p. The $Join(p)$ function determines if r_p has been projected in $\mathit{\Pi}$ 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 has more than 3 vertices.

Tuple identification We define a function $Find(U, \mathit{\Pi})$ that identifies modified tuples in a projection. Let us assume

$$U = (S, type, value_{id}, Val) \quad (40)$$

is an update where

$$Val = \{(r_i, value_{r_i}), \ldots, (r_j, value_{r_j})\}) \quad (41)$$

as previously defined. Suppose r is an attribute from $\{r_i, \ldots, r_j\}$ and let us focus on a single element of $Path(U, \mathit{\Pi}, r)$ from (37). Given

values of U, $Find(U, \Pi)$ returns primary keys of modified tuples in Π. Let AV_{t_i} denote a value of an attribute t_i in the modified tuple and suppose

$$AV_{t_0} = AV_{R_U.id} = value_{id}. \tag{42}$$

Suppose t_i and t_{i+1} are attributes of relations R_{t_i} and $R_{t_{i+1}}$ respectively. Then $AV_{t_{i+1}}$ is determined as follows:

$$AV_{t_{i+1}} = \begin{cases} retrieve(Data(t_{i+1}), AV_{t_i}), & R_{t_i} = R_{t_{i+1}} \\ AV_{t_i}, & R_{t_i} \neq R_{t_{i+1}} \end{cases} \tag{43}$$

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 $R_\Pi.id$. This is possible since we iterate through strong edges, and in case of connecting attributes, a strong edge connects either an attribute with the primary key of the same relation or foreign key with the primary key of the associated relation. As a result $AV_{R_\Pi.id} = AV_{t_m}$.

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 $Map(U)$ and Π. 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 $(Map(U), r) \in E$. In general $Find(U, \Pi)$ returns the maximal set that contains pairs of strong paths between update and projection vertices, associated with the primary key of the tuple that has to be modified:

$$Find(U,\Pi) = \{(g(U, p), p) : \exists_r (Map(U), r) \in E \wedge p \in Path(U, \Pi, r)\} \tag{44}$$

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

$$Val(\Pi,U,p) = \{(r_i, value_{r_i}) : r_i \in A(U,\Pi, p) \wedge (\kappa(r_i), value_{r_i}) \in Val\} \tag{45}$$

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

$$Mod(Val(\Pi, U, p), \Pi, p, value_{id}) \tag{46}$$

which modifies data in underlying storages. The function modifies a tuple in a projection Π 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 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. Suppose U modifies a manager's name. Then the 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 Chapter 5.1 that each projected attribute can store values which are processed by *safely updateable* or *incrementally updateable* selections. Thus we can easily evaluate a new value of a tuple in Π and this is done in Mod. The direct implementation depends on a selection type. As an example, in the case of *count* we increment the value when U adds a tuple, decrement in case of tuple removal or leave it unchanged if modified.

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

1. If U is *add* then:
   - 1.1 Let $\Pi = Prim(U)$.
   - 1.2 Let p denote a strong path containing $(Map(U), R_U.id, \Pi)$.
   - 1.3 Apply $addPrimary(\Pi, Val(\Pi, U, p))$ 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 $T = Proj(U) \setminus \Pi$, in other case let $T = Proj(U)$.

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

First the algorithm checks the type of the submitted update U. If a new tuple is going to be inserted, it is first added to the primary projection of the updated relation where the 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 $Find(U, \Pi)$ and contain the primary key of the modified tuple and a strong edge path corresponding to the trace of modified attributes.

Step 3.1.2 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. 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 that is affected by U. The changes are applied by the Mod function.

5.6 A Dependency Graph Example

Figure 5 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

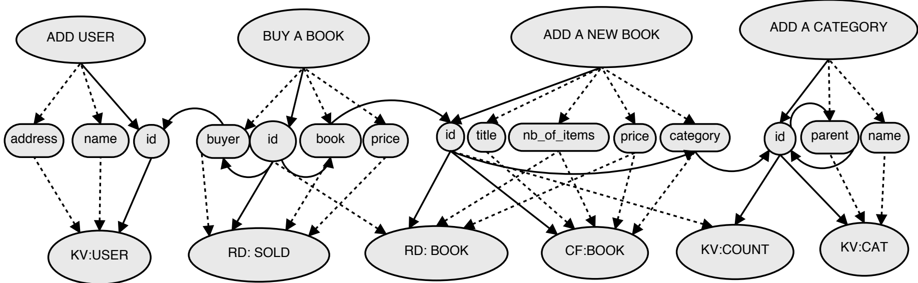

Fig. 5. Graph example from the bookstore application. The upper vertices represent update operations. The middle ones correspond to relation attributes of relations: user, sold items, book and category. The lower ones are vertices representing projections stored in backend storages.

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, the 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:

$$(Map(U), book.id, book.category, category.id, \pi) \tag{47}$$

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:

$$(Map(U), book.id, book.category, category.id, \\ category.parent, category.id, \pi) \quad (48)$$

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.

5.7 Correctness

In this section, we prove the correctness of the presented algorithm. Let Π denote an arbitrary projection, and U an update that occurs. Additionally assume T denotes the state of Π before the update, and T' the 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 proof 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.

Detecting data modification Now we focus on data modifications. We start with the following lemma:

Lemma 1. *An update U modifies data in a projection Π iff. $\Pi \in Proj(U)$*

Proof. Assume $\Pi \in Proj(U)$. According to the definition of $Proj$ (31), there exists an attribute r such that (r, Π) and $(Map(U), r)$ are contained within E. The existence of the edge $(r, \Pi) \in E$ implies that Π projects the attribute r, while the edge $(Map(U), r)$ implies

that U modifies a value of the attribute r, and as a conclusion U modifies Π.

Now suppose U modifies Π. Then U has to modify some attribute r that is projected by Π. U modifies r in some tuple, then $(Map(U), r) \in E$. Additionally, since Π projects r, $(r, \Pi) \in E$. As a result $\Pi \in Proj(U)$, which ends the proof. □

Now we are going to prove that *Find* returns modified tuples or nothing, when the given projection has not been updated. We start with the evaluation of $g(U, p)$. The strong path p determines the sequence of joins used in the projection. Assume $t_1, t_2, \ldots, t_m$ are the foreign key attributes in p and a relation R' is constructed as:

$$R_{t_1} \bowtie_{R_{t_1}.id=t_2} R_{t_2} \cdots \bowtie_{R_{t_{m-1}}.id=t_m} R_{t_m} \quad (49)$$

where R_{t_1} equals R_Π, and R_{t_m} equals R_U.

Lemma 2. *Let U be an update equal $(R_U, type, value_{id}, Val)$ and let p denote a strong path between $Map(U)$ and Π. Then:*

$$g(U, p) = \pi_{R_\Pi.id}(\sigma_{R_U.id=value_{id}}(R')). \quad (50)$$

Proof. We have defined $g(U, p)$ as the function which traverses the strong path p and gathers the primary keys of the traversed relations. We start the traversal with the primary key $value_{id}$ 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:

$$AV_{p_{i+1}} = AV_{p_i} \quad (51)$$

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:

$$\pi_{p_{i+1}}(\sigma_{p_i=AV_{p_i}}(R')) \quad (52)$$

In our data model we allow only *one-to-many* joins, between relations. According to this:

$$\sigma_{p_{i+1}=AV_{p_{i+1}}}(R') \subset \sigma_{p_i=AV_{p_i}}(R') \quad (53)$$

This leads us to:

$$g(U,p) \in \pi_{p_{i+1}}(\sigma_{R'.t_m=value_{id}}(R')) \tag{54}$$

The last thing, we have to proof, is that $\sigma_{R'.t_m=value_{id}}(R')$ selects a single tuple from R'. R' is constructed from relations R_{t_1}, R_{t_2}, ..., R_{t_m} such that:

$$R_{t_1} \lhd R_{t_2} \lhd R_{t_3} \lhd \cdots \lhd R_{t_m} \tag{55}$$

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 Π, modified by U, is contained in pairs of $Find(U,\Pi)$. Then, we prove the opposite direction: each pair of $Find(U,\Pi)$ contains the primary key of some tuple in Π, that is modified by U.

Lemma 3. *Suppose k is a tuple in Π with the primary key equal $value_k$. If k is modified by an update $U = (R_U, type, value_{id}, Val)$, then $value_k$ is contained in some pair of $Find(U,\Pi)$.*

Proof. We show the construction of a strong path p such that:

$$(value_k, p) \in Find(U,\Pi) \tag{56}$$

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:

$$R' = R_{t_1} \bowtie_{R_{t_1}.id=t_2} R_{t_2} \cdots \bowtie_{R_{t_{m-1}}.id=t_m} R_{t_m} \tag{57}$$

is a relation with an attribute r, while $R_{t_1} = R_\Pi$ and $R_{t_m} = R_U$. From Definition (11):

$$(r_p, t) \in Trace(\Pi) \tag{58}$$

and Definition (37) it follows that:

$$t \in Path(U,\Pi,r) \tag{59}$$

Based on the definition (44) of $Find$, we know that there exist a strong path p such that:

$$(g(U,p),p) \in Find(U,\Pi) \tag{60}$$

which, according to the lemma 2, is equivalent to:

$$(\pi_{R_{t_1}.id}(\sigma_{R_{t_m}.id=value_{id}}(R')),p) \in Find(U,\Pi) \tag{61}$$

Assume the value of an attribute r_p from R' is modified in k by U. According to this:

$$value_k \in \Pi_{R_{t_1}.id}(\sigma_{R_{t_m}.id=value_{id}}(R')) \tag{62}$$

From the proof of Lemma 2 we know that $\Pi_{R_{t_1}.id}(\sigma_{R_{t_m}.id=value_{id}}(R'))$ selects just a single row from R' and, as a result, it equals $value_k$. Eventually, this implies:

$$(value_k,p) \in Find(U,\Pi) \tag{63}$$

□

Lemma 4. *Suppose an update* $U = (R_U, type, value_{id}, Val)$*, a strong path* p *and a projection* Π*. If* $(value_k,p) \in Find(U,\Pi)$*, then a tuple* k *in* Π *with the primary key equal* $value_k$ *has been modified by* U*.*

Proof. According to Definition (44), there exists an attribute r such that:

$$(Map(U),r) \in E \tag{64}$$

and:

$$p \in Path(U,\Pi,r) \tag{65}$$

This ensures, from the definition (37), 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, ..., t_m are the foreign key attributes in t. We construct a sequence of relations R_{t_1}, R_{t_2}, ..., R_{t_m} such that:

$$R' = R_{t_1} \bowtie_{R_{t_1}.id=t_2} R_{t_2} \cdots \bowtie_{R_{t_{m-1}}.id=t_m} R_{t_m} \tag{66}$$

Again, $R_{t_1} = R_\Pi$ is the primary relation of Π, and $R_{t_m} = R_U$ is an updated relation. Due to definition (11) of $Trace$, r_p is a projection attribute from R' in Π. 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:

$$\pi_{R_{\Pi}.id}(\sigma_{R_U.id=value_{id}}(R')), \tag{67}$$

since R_{t_m} constitutes the primary key of R'. Based on Lemma 2, this assures that it is equal $g(U, p)$ 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. □

Applying data changes: We prove that data modifications, applied by the algorithm, are correct. We assume that an update U modifies a tuple k in a projection Π. We denote by T the state of Π before the update, and by $S_{\Pi}(T, U)$ the state of the projection after performing U. Additionally assume F_{Π} is the function of U and T, which returns the state of Π after applying U by the presented algorithm.

Let us focus now on the $Val(\Pi, U, p)$ function defined in (45). Assume an update U that affects an attribute r_p of a tuple k from a projection Π. We start with the following lemma:

Lemma 5. *Let p denote a strong path corresponding to the $tr(\Pi, r_p)$ and let $value_{\kappa(r_p)}$ denote a value of U used to modify r_p in k. Then:*

$$(\kappa(r_p), value_{\kappa(r_p)}) \in Val \Leftrightarrow (r_p, value_{\kappa(r_p)}) \in Val(\Pi, U, p) \tag{68}$$

Proof. First, we assume $(r_p, value_{\kappa(r_p)}) \in Val(\Pi, U, p)$. The leftward implication follows from Definition (45) 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 (45), we need to prove that $\kappa(r_p) \in A(U, \Pi, p)$, which has been defined in (38). 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 path between $Map(U)$ and Π. This assures that for a trace t, corresponding to p, $(r_p, t) \in Trace(\Pi)$ which ends the proof. □

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

Lemma 6. *If* $Join(U, \Pi)$ *equals* 0*, then:*

$$F_{\Pi}(U, T) = T'. \tag{69}$$

Proof. When $Join(U, \Pi)$ equals 0, then U modifies the primary relation of Π. According to the type of U, the algorithm does the following operation:

1. In case of adding a new tuple, the tuple is added to Π. A tuple k is filled with values from $Val(\Pi, U, p)$. The attributes of Π, that are missing values in U, are filled with default values.
2. In case of deleting a tuple, k is deleted from Π.
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 second case is self-explanatory, the correctness of the other steps relies on the lemma (5). The lemma assures that updating tuple with values of $Val(\Pi, U, p)$, which contains a subset of values from U, applies the changes of U to the tuple in Π. 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 Π. □

Now we elaborate on the second case when $Join(U, \Pi)$ equals 1:

Lemma 7. *If* $Join(U, \Pi)$ *equals* 1*, then:*

$$F_{\Pi}(U, T) = T' \tag{70}$$

Proof. Let R_{Π} be the primary relation of the projection Π and R_U is the relation modified by U. $Join(U, \Pi) = 1$ implies that $R_{\Pi} \neq R_U$. The update U modifies attributes that are projected in Π via some chain of *one-to-many* joins. In this case a tuple in Π is always modified, even if U adds or deletes some tuple in S. The data modification depends on the implementation of the Mod function from (46),

$$Mod(Val(\Pi, U, p), \Pi, p, value_{id}) \tag{71}$$

Parameters Π and $value_{id}$ identify the projection and the primary key of the 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 Π, while U contains the type of operation and new attributes' values.

The projection Π 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 Definition 1 and Definition 2, the selection function that selects the projection attribute r_p from the relation attribute r has to fulfil the following criteria: if an update occurs the new value of r_p can be recomputed from the former value of r_p and the 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. □

The correctness theorem: We have already defined and proven all needed lemmata and we introduce the main theorem:

Theorem 1. *Assume U is an update of the form*

$$U = (R_U, type, value_{id}, Val). \tag{72}$$

For each projection Π:

$$F_{\Pi}(U, T) = T'. \tag{73}$$

Proof. The proof of the theorem relies on the former lemmata. Suppose a projection Π. Using Lemma 1 we know that the algorithm updates data in Π iff. $\Pi \in Proj(U)$. According to Lemmata 3. and 4., $value_k$ 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 the trace. The trace allows determining how the update affects the tuples. The algorithm modifies found tuples according to two different cases: $Join(U, \Pi)$ equal 0 or 1. Lemmata 6. and 7. 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 according to the

selection type. We have proven the correctness of both steps, thus proving the correctness of the whole algorithm. □

5.8 Complexity

In this section, we consider the complexity of the algorithm. As described in Section 5.3, the complexity of the DCP problem is a function of the schema size and the size of stored data. We prove that the complexity of our algorithm does not depend on the size of data. Furthermore, we show that some methods used by the algorithm can be precomputed before the deployment.

Lemma 8. *The complexity of the algorithm does not depend on the size of stored data.*

Proof. This can be proven without going into details of the methods used in the algorithm. Each method operates on the dependency graph and the graph's size depends only on the schema's complexity. The main algorithm checks the projections modified by an update. For each projection it iterates through the elements of $Find(U, \pi)$. Thus, it only depends on the schema's complexity.

The key point of this lemma, is the $Map(U)$ function defined in (15), which maps update operations to graph vertices. This gives us abstraction classes of each update. Two updates U_1 and U_2, such that $Map(U_1)$ equals $Map(U_2)$, are applied the same way by the algorithm. Computations of the algorithm are run on the graph structure composed of $Map(U)$ vertices rather than U vertices. This assures that the algorithm depends only on the schema complexity. □

Now we are going to assess the complexity of the algorithm as a function of the size of the database schema. Although the size of the schema is significantly smaller than the size of data, the number of update vertexes in the graph can still grow exponentially. Suppose a system with k relations and l attributes in each of them. The number of possible update vertices equals $k \cdot (l-1)!$, since each update vertex has to connect the primary key of the modified relation. This could break our assumption about the simplicity of the graph. In order to avoid such situation, when implementing the system, we can use the following trick in the algorithm. Suppose an update U of the form:

$$(R, type, value_{id}, \{(r_i, value_{r_i}), \ldots, (r_j, value_{r_j})\}) \quad (74)$$

The same changes can be done by applying the sequence of single attribute's updates. For each modified attribute r, we create an update:

$$(R, type_r, value_{id}, \{(r, value_r)\}) \tag{75}$$

We denote the new update type by $type_r$, and it is evaluated as follows. In case of deleting a tuple, there are no values in U, and the sequence contains a single update which equals U. If *type* represents editing operation, then $type_r$ equals *type*. The difference appears in case of adding a tuple. When adding new tuple, the type of the first update in the sequence is *add*, while the others simply edit the new tuple.

Splitting each update into the sequence of single attribute's updates solves the problem of too many update vertices. At this stage, the number of update vertices is limited by the number of relation attributes. However, the algorithm in this form introduces a new problem: updating each column separately will increase the overall number of operations run on underlying storages. To avoid this, we modify steps of the algorithm where it writes to underlying storages. Instead of running immediately each requested update, when methods $addPrimary$, add, $modify$ or $delete$ are called, the algorithm puts them on a stack. They are applied, when the dependency algorithm gathers all write operations that need to be run. Running together all updates on underlying storages allows grouping them, so that an update on a single tuple in a projection is run only once. Of course these are merely implementation details of the algorithm. However, they influence the complexity analysis. Thus they had to be clearly stated.

So far we know the complexity of the algorithm in terms of the number of calls to auxiliary functions. Now we evaluate their complexities. In our analysis we skip the complexity of methods which can be precomputed at the applications deployment, as they are called only once and do not influence the complexity of an update request execution.

First, we show the methods that can be precomputed. We start with $Proj(U)$, that needs to be computed only once for each vertex $Map(U)$. Therefore, it does not affect the complexity. The same happens with $Path(U, \pi, r)$ and $A(U, \pi, r)$. The evaluation of both

methods depends on $Map(U)$ and when the methods are precomputed, they can be stored in a lookup table.

On the other hand the method $Find(U, \pi)$ has to be computed for each actual update U, since it uses the update values to retrieve the primary key of the modified projection. Thus, it cannot be precomputed. The $Find$ method traverses strong paths, from an update vertex to π, that correspond to the traces of attrubutes in π. The number of traces is limited by the number of projected attributes, which is in fact limited by l from an equation (22). From equation (23), we know that it is limited by m, as $m = |V| + |E| + l$. For each found trace, the $Find$ method traverses the path to identify the primary key of a modified tuple. Since we allow self-joins, a strong paths may contain graph cycles. According to this, we cannot limit a path length with a number of graph vertexes. However, it is still limited by l and m, as l is a sum of all traces' lengths. To sum up, $Find(U, \pi)$ traverses at most m paths of a size at most m each. Thus its complexity equals $O(m^2)$.

The main algorithm contains a loop and the $Find$ method is called within the loop. These are the only methods that affect the complexity and can be estimated as $O(m^3)$. This has lead us to a lemma.

Lemma 9. *Given a dependency graph and a database schema, suppose m as defined in (23). Then, the complexity of the algorithm equals $O(m^3)$.*

The complexity of the algorithm is thus limited with the size of the data schema. The overall complexity of a system based on our algorithm is thus hardly influenced with the complexity of the propagator algorithm. However, the overall system performance does depend on the time spent when applying changes in backend storages and synchronizing propagator threads. This will be examined in details in further chapters. At this stage we have proven that the complexity of the propagator algorithm does not depend on the size of data. This has been our aim, as stated in Section 5.3. Moreover, we want to build a fully scalable update layer. In this chapter, we have proven that in theory the overhead of the propagator algorithm is not significant.

6 Consistent Caching

In this chapter we describe the problem of consistent caching. It has been the subject of our initial research. The results of this research have been published in [48]. They have lead us to the idea of the joint consistent storage, i.e. the main topic of this thesis. The update propagator algorithm described in the previous chapter extends the ideas presented here and applies well to the problem of consistent caching presented in this chapter.

6.1 Motivating example—a community forum application

We start with an example. Let us consider a community forum application. Suppose we have four tables: *user*, *forum*, *topic*, *post*. The table *user* contains user data. Fora are stored in the *forum* table. Each forum consists of topics, which contain lists of posts. Let us assume the database schema consists of the following tables:

```
user:  id, nick
forum: id, name, desc
topic: id, forum_id
post:  id, topic_id, title, text, user_id, created_at
```

When a new topic is added, its title and other data are stored in the first post. From the database's point of view, the website consist of three database intensive views: listing fora, listing topics and listing posts. Let us now focus on listing topics. Figure 6 contains a snapshot of a real forum application. It shows which data are needed when displaying the list of topics.

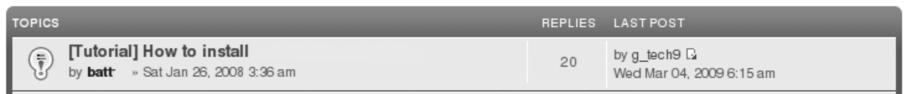

Fig. 6. The figure shows a single line from the list of visible topics. Each line contains: a topic's name which is the first post's name, the owner of the topic, the date it was created, the post count, and the information about the last post: its author and the date it was added.

When the database schema is in 1NF, performing a query each time the website is loaded is too expensive and harms the database.

Thus such an architecture is not used in practice. Instead, modern systems modify the database schema by adding redundant data. In particular, one can add *first_post_id*, *last_post_id* and *post_count* fields to the tables *forum* and *topic* and also *topic_count* to the table *forum*.

Such a solution resolves the efficiency problem stated before. However, it also introduces new difficulties. It recedes the database schema from 1NF which is strongly desired in OLTP applications. Consider an addition of a new post as an example. At such an event, the *post* table is not the only one to be modified. In order to maintain the post count, we also have to update *topic* and *forum*. This also does not solve the main problem, since the database is still the bottleneck of the system. Adding redundant data means also adding several logical constrains that have to be maintained and are error prone. It would also introduce problems when trying to replicate the whole database.

The obvious solution is to use a cache to keep all these views in memory. However, the data are updated by the users who add posts. Whenever this happens, many post counters have to be recomputed. The desired property of a cache is to recalculate only those counters which have to be recomputed and possibly nothing else. In this chapter we show a method how to reduce the invalidations as much as it is practically possible.

6.2 Existing caching solutions

Existing caching models can be divided into caching single queries, materialized views and tables. When caching single queries it may be hard to discover similarities and differences between queries and their results. Let us suppose a query and its result returned with descending and ascending sorting criteria. If no paging (limit/offset) is defined, it is still the same result but in a different order. Apparently, it does not need to be stored twice. Caching based on hash based algorithms does not work well in the presented application and applies more to caching files than to the OLTP database schemata. The idea of caching tables' fragments has been first shown in [22] where the authors propose a model with dividing tables into smaller fragments. It can be understood as storing data sets in caches and allowing for them to be queried [5, 4, 51]. However this does not solve the whole

problem, since it lacks count operations. In the forum example, and in most Web 2.0 applications, the website contains several counters which cannot be evaluated each time the website is loaded. Performing count operations on the cached data sets is difficult. It is hard to detect if all data to be counted is loaded into the cache. There is also no need to store whole data when only counters are needed. Another parameter becomes an issue. Data can be loaded ad hoc or loaded dynamically, each time it is needed. When all data are loaded to the cache at once, one can easily discover which count queries can be performed but it also means caching data that may never be used. When an update occurs, all the cache needs to be reloaded. This may cause severe problems because updates occur frequently in OLTP. This also means performing updates to maintain the consistency of the cached data which is never used. Invalidation can occur each time the update occurs or in the specified time intervals. The second case would be efficient but would also allow storing and serving data that is not up to date. Loading data statically is more like database replication than a caching technique. An interesting recent work on replicating data sources and reducing communication load between backend database and cache servers has been described in [64]. It presents an approach based on hash functions that divide query result into data chunks to preserve the consistency. However this also does not solve the problem of aggregation queries. Their results are difficult to be kept consistent via hash similarity.

Described schemata show that granularity of the cached data is strongly desired. In that case only atoms which are not up to date would be invalidated thus improving the efficiency. However these atoms cannot be understood as table rows, since count would be difficult to define, and they should be more like tuples containing data specified by the application logic. This is what many schemata shown before cannot afford because of persistent proxy between the application server and the database. On one hand, this feature aids software programmers because they do not need to dig into caching techniques. On the other hand, it is the software programmer who has to specify what data has to be cached because it is strongly related to the application's specific logic.

Caching Objects vs. Caching queries Most of previously described caching techniques include caching queries. This solution needs a specification of queries that need to be cached because they are frequently performed. If the query used for listing topics of a community forum is taken into consideration, one can argue if it makes sense to cache its result. On one hand, the query is performed each time the user enters a specific forum. On the other hand one should be aware of user conditions. If the user searches for topics with posts containing a specific string, it may be useless to cache them because of the low probability they will ever be reused.

Instead of caching queries, one should take into consideration caching objects. Suppose objects of class *topic* and *forum* are created and each of them contains the following fields:

```
FORUM: id, name, post_count, topic_count,
       last_post_author_nick, last_post_created_at
TOPIC: id, first_post_title, first_post_created_at,
       first_post_author_nick, last_post_author_nick,
       last_post_created_at
```

Having such objects stored in a cache, the query listing topics could look like the SQL query below:

```
SELECT id FROM topic WHERE forum_id = $forum_id
AND ## user condition LIMIT 20;
```

With the list of topics' ids, we simply get those objects from the cache. Each time the object does not exist in the cache it is loaded from the database and created. This means significant reduction of the query complexity and the performance improvement. *Memcached* [19] is an example of the mechanism widely used in practice. It is a high-performance, distributed memory object caching system and is used by *Wikipedia*, *Facebook* and *LiveJournal*. In December 2008 *Facebook* considered itself as the largest *memcached* user storing more than 28 terabytes of user data on over 800 servers [56].

From the theoretical point of view, the idea can be seen as using a dictionary for storing data objects created from the specified queries. One can argue if storing relational data inside the dictionary is sensible. Here the performance becomes an issue. Since all

the cached data is stored in RAM (it makes no sense to cache data on a disk) a cache server only needs to hash the name of the object and return data which are stored under the hashed location. The caching mechanism is outside the database. This means a significant performance gain due to the reduction of the database workload.

Data consistency problem The data consistency problem arises when caching techniques are applied. It is similar to the one stated in previous chapters: how to maintain data in caches consistent with data from a relational database. Several improvements to the consistent caching problem have been applied by the industry. According to [67] *Wikipedia* uses a global file with the description of all classes stored in the cache [1]. However this is not a general solution to the problem but only an improvement which helps programmers to manually make a safe guard from creating objects in an inconsistent state. In our research, we propose the model of fully automatic system which maintains the consistency of cached data.

The presented solution brings some SQL restrictions to queries that feed caches. These restrictions are needed, since mapping the relational data to the dictionary is performed. As we restrict to the domain of web applications, select statements are significantly more frequent than inserts and updates. We assume that the database contains k relations and each of them has the primary key which consists of a single attribute. We identify the set of select statements S which are used for creating cached objects:

$$S = \{S_1, S_2, ..., S_r\} \tag{76}$$

For each class of objects we can identify the subset of S used to create its objects. The set U is the set of statements which modify the database.

$$U = \{U_1, U_2, ..., U_m\} \tag{77}$$

Each of members of U modifies only a single row selected by a value of the primary key. Additionally select statement can have other conditions in the WHERE clause but they can only involve columns

[1] This file is available at `http://svn.wikimedia.org/viewvc/mediawiki/trunk/phase3/docs/memcached.txt?view=markup`.

of the parameterised table. Cached objects and queries from S and U are parameterised by the primary key of some relation.

In some cases, it is convenient to know cached data to optimize invalidation clues. However the model assumes data inside objects to be persistent because sometimes, instead of caching data, developers decide to cache whole HTML fragments corresponding to the cached objects. Other database caching systems could not allow this because of being persistent to the application server. Once again the persistence of caching models reveals its drawback. The presented model can be also seen as a combination of caching static HTML pages [42] and data from database. The similar construction of mapping database content to Web pages is presented in [50], but it maps queries to URL addresses. However this approach loses efficiency in Web 2.0 applications since the same HTML code can used on many pages and it does not make sense to invalidate whole pages.

The aim of the following construction is to create a graph which allows identifying objects to invalidate when an update statement performed. The construction is similar to one presented in the previous chapter. However, there are some differences. We do not want to use the update propagator algorithm to manage all system storages. Here, we assume there exists a backend RDBMS with a known schema and there exists an extra layer where cached objects are stored. Each time an update occurs, it is registered to the cache manager. The cache manager algorithm checks if there are some cached objects that require invalidation, and updates them if necessary. We present the cache manager which assures that each time data is changed, all the cached data remains consistent with the primary storage.

6.3 The dependency graph for consistent caching

Query identification Let us first identify queries used by the application when creating objects in our forum example. The list of queries used when creating topic objects follows.

```
S1:   SELECT * FROM topic WHERE id = $topicId;
S2_1: $max = SELECT max(p.created_at) FROM post p, topic t
      WHERE t.id = p.topic_id
```

```
        AND t.id = $topicId;
S2_2: SELECT u.nick FROM post p, user u, topic t WHERE
      t.id = p.topic_id AND t.id = $topicId AND
      p.user_id = u.id AND p.created_at = $max;
S3_1: $min = SELECT max(p.created_at) FROM post p, topic t
      WHERE t.id = p.topic_id
      AND t.id = $topicId;
S3_2: SELECT u.nick FROM post p, user u, topic t WHERE
      t.id = p.topic_id AND t.id = $topicId AND
      p.user_id = u.id AND p.created_at = $min;
S4:   SELECT count(p.post_id) FROM post p, topic t WHERE
      p.topic_id = t.id AND t.id = $topicId
```

The first statement gets a row from the *topic* table. $S2_1, S2_2$ and $S3_1, S3_2$ are very similar and are used for getting data of the first and the last post in the topic. Additionally $S4$ is performed to evaluate the number of posts in a topic. In our example update statements can also be identified. These are the five queries that manipulate data:

```
U1: INSERT INTO user VALUES ...
U2: INSERT INTO forum VALUES ...
U3: INSERT INTO topic VALUES ...
U4: INSERT INTO post VALUES ...
U5: UPDATE user SET nick WHERE user_id = ...
```

The select statements that are not used for creating objects are not considered. This is a big difference in contrast to the update propagator in the previous chapter. We have presented a system that maintains consistent data among storages. Now we assume there exist a master RDBMS, and each time an update is applied, it is registered to our system which invalidates cached data when necessary.

Construction of the graph We have already identified the queries used in the system. The construction of the graph is similar to the one from the previous chapter and we will not cover it in details. Each update operation and each schema attribute constitutes a vertex. Edges among vertices are added the same way as before: each update connects by a strong edge the primary key of the modified

relation and by a weak edge all other modified attributes. Within attribute vertices, the primary key of each relation connects by a strong edge all foreign keys within the same relation. As a difference to the previous construction, edges between attributes are added in both directions. Each foreign key connects by a strong edge the primary key of the foreign relation.

Queries used for object creation can be parameterized with the primary key of some relation. We assume that each cached object contains data of queries that are parameterized with the key of the same relation and we call it the primary relation of an object. Objects contain data from the primary relation and attributes from other relations that are joined via a sequence of *one-to-many* or *many-to-one* relationships with the primary relation. This is a significant difference with the previous graph description as we also allow *many-to-one* relationships. This can be done as there exist a master relational database. The model in previous chapters, when dealing with *one-to-many* associations, could not traverse the edge in both directions. Given a tuple of the foreign relation, based on the foreign key, it could have reached the single tuple in the foreign relation. However when there are many tuples associated with the one given, it could not have fetched them, which can be easily done in this model within SQL syntax. Each type of queries constitutes a vertex and a strong edge connects the primary key of the primary relation. Then, weak edges connect attributes that are projected by a query. These may include attributes from the primary relations and other relations that are joined in this query.

The rules already described are similar to the graph definition from the previous chapter. There are vertices that represent cached objects. Each cached object can be parameterized with the primary key of its primary relation. Each class of such objects constitutes a vertex and we connect by strong edges a cached object's vertex with all queries that are used to create an object.

Figure 7) displays the consistency graph created for the forum example. We have already shown queries used for creating objects and updating data. For the sake of clarity, we present only the *topic* object.

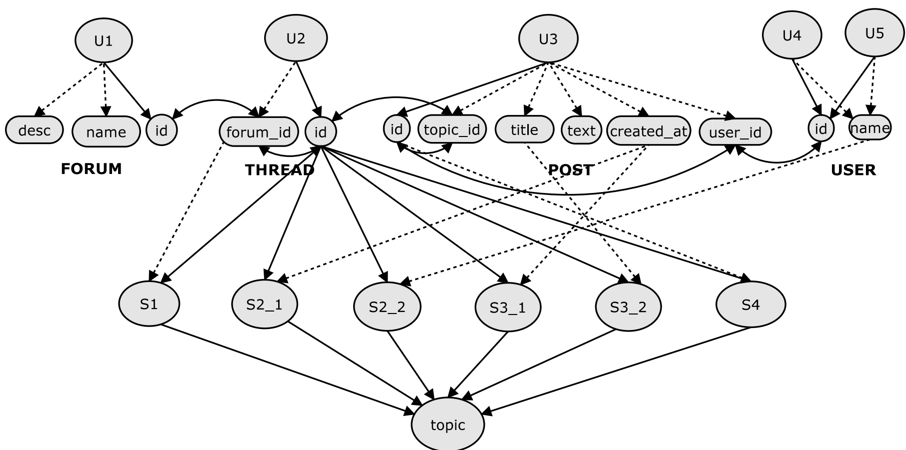

Fig. 7. The dependency graph created for the topic object in the community forum example

Invalidation clues The propagator described in the previous chapter, the propagator applies all changes, when an update occurred. It modifies at first the primary storage and then secondary storages are modified simultaneously. In this chapter, we assume the application modifies data in RDBMS itself. Then the update is submitted to the cache manager, which detects objects that need invalidation and removes them from the cache. It does not update cached data. This is also in contrast to the algorithms considered in the previous chapter.

The cache manager only detects changes and removes stale objects from a cache. There are several arguments to do so. It is encouraged by design patterns to work with *key-value* cache storages like *Memcached*. Applications that make use of RDBMS and such caches mostly decide on the cache-aside design pattern and lazily put data into the cache. It is the application logic which takes care of putting data into the cache. Each time data is needed, the application checks if it is present in the cache. If it is, it fetches data. If it is absent, the data is retrieved from RDBMS and loaded into the cache. This is the fallback which assures that no matter what is stored, the proper data is returned.

When a new update is received, the cache manager checks if a vertex, corresponding to this update, exists. If not, it is being

added with all edges: a strong edge to the primary key attribute and weak edges to all attributes of modified columns. Then the system identifies classes of objects that may need an invalidation. It simply checks if there exists an attribute that is modified by the update and is projected by the query creating cached objects. If such an attribute is found, there exist instances that need to be invalidated. This can be also seen on the example graph from Figure 7. When a new forum is added there is no path between $U1$ and the *topic* vertex. The *topic* object does not need to be invalidated since no attribute used to create it have been changed.

At first, the cache manager determines classes of objects which may contain invalid objects. For each query used for creating objects ($S1, S2_1, S2_2, S3_1, S3_2, S_4$) the trace of this query contains the information on the sequence of its joins. As in the previous chapter, these traces are used to create strong paths between update vertexes and query vertexes. For each strong path the cache manager traverses it and gathers values of the traversed vertices. The previously described algorithm is run and it ends up with the primary keys of queries used for creating objects. These are the primary keys of cached objects that need to be invalidated. Then they are removed from a cache.

Let us now return to our example. When a new post is added, the cache manager goes from an *id* vertex of the *post* table to the *topic_id* in the tables *post* and *topic*. Using the value of the *topic_id*, it knows the primary key of the object to invalidate since objects and select statements are parameterized by the same primary key. In this case no additional queries need to be performed on the database. However, this is not true in all cases.

Let us now suppose an editing operation of a user's nick. Having the *id* of a user, the cache manager needs to find all posts written by the user and performs queries in the database. At this stage when traversing a strong path, the system traverses through *one-to-many*, and also *many-to-one* associations. This is possible, as all data is stored in the relational database. It requires an additional query of the form:

```
SELECT p.id from post p where p.user_id = $user_id$
```

In the previous chapter the algorithm could only traverse through *one-to-many* associations in one way, as there were no assumptions about databases that store it. Now, it is possible and the algorithm gathers multiple values when traversing a path. Then it invalidates all topics where the user has written posts. This can be seen as a drawback but it is impossible to examine if the user's post is the first or the last without querying the database. One can argue if it can be improved for $min()$ and $max()$ functions but it surely cannot be done for $avg()$ so no general solution without digging into SQL syntax exists.

The other thing is the chain of joins between tables. If a found path goes through several joins which require querying database, the system can be inefficient. However it applies well in most cases since in OLTP database queries shall be kept as simple as possible. One should also resemble that even when dealing with complicated database schemata, not all of data has to be cached and objects should be kept as granular as possible to prevent extensive invalidation.

According to us, the implementation of the invalidation algorithm should strongly rely on the data model of the development framework. Let us suppose using an object relational mapper (ORM) based on the active record pattern. The graph can be constructed dynamically, i.e. each time the query is performed the system checks if it is included in the graph. If not, it is being added. The algorithm needs only to identify the update and select statements used for creating cached objects. When using the active record pattern, update methods are performed on a single row and are easy to discover. On the other hand select statements are performed only via built-in ORM's methods. These methods can be overridden by new ones that execute select statements for creating cached objects. When doing so, we can clearly distinguish between select statements used for creating objects and other statements.

6.4 Experimental results

The presented model has been tested on the RUBiS benchmark [13]. RUBiS is an auction site prototype modeled after *www.ebay.com* and provides a web application and a client emulator. The emulator is

modeled according to a real life workload and redirects the client from one webpage to another due to a predefined transition probability. In the presented benchmark we have used 100 client threads running at the same time for 18 minutes.

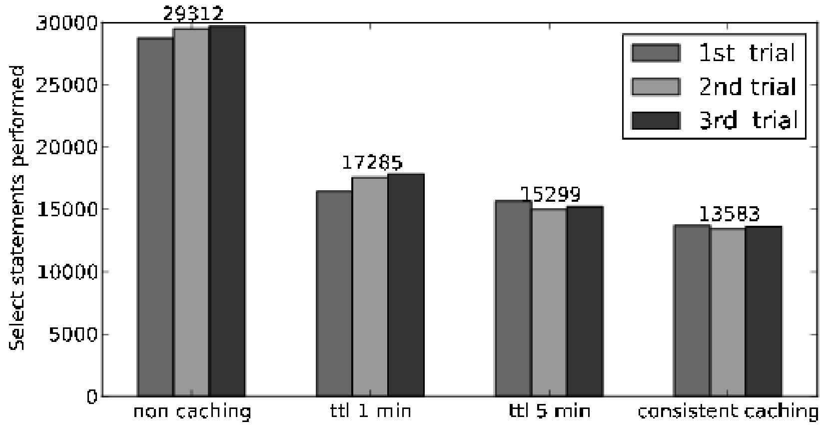

Fig. 8. The comparison of different caching techniques. The numbers indicate average count of select statements for each technique

We have tested the application in different modes: (1) without caching, (2) with *time-to-live* caching, (3) with the cache management based on the dependency graph.

The application benchmark has been run three times in each mode. Figures 8-10 display achieved results. Our cache manager reduces up to 54% of performed queries. It is more efficient than techniques based on *time-to-live* and does not store stale data. In the experiment, no cached objects have been invalidated when unnecessary and there have been no queries that did not fit into to the SQL syntax boundaries defined in the presented model.

We have also measured the number of data modifications and the number of cache invalidations as presented in Figure 10. In the presented benchmark 7.1% of database statements have modified data. None of them updated more than one row. This proves that our principal assumption that we deal with the read dominant database communication is correct. Figure 9 shows that the number of cache invalidations does not grow rapidly when compared to database statements. The right figure shows that almost 61% cache

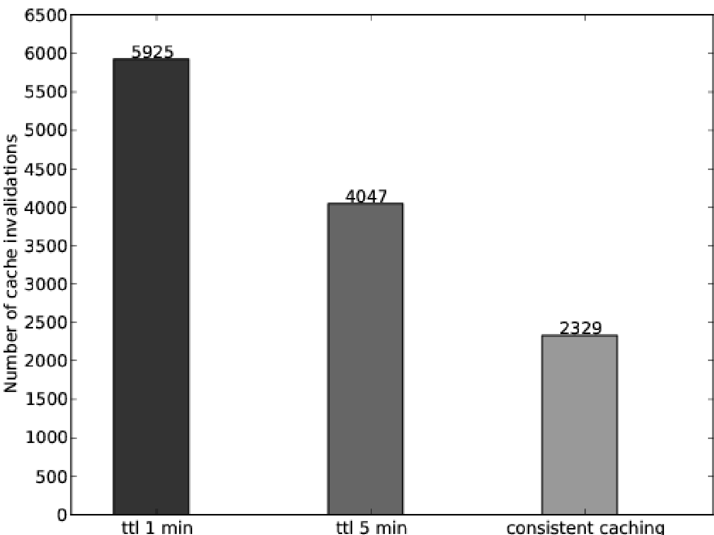

Fig. 9. Number of data modifications.

invalidations can be saved by the presented model when compared to *time-to-live* techniques. This proves the significant improvement of the presented model with respect to those techniques.

6.5 Analysis

The database bottleneck is the most serious problem in modern web applications. The widely applied industry solution is a scalable *key-value* cache. However, this causes the consistency problem between the relational database and the cache. In this chapter we described our solution that is based on the dependency graph to detect invalidations of the cached objects when updates occur. Since a general solution to the consistency problem of relational and *key-value* storages is difficult and may be inefficient, we defined boundaries to the SQL syntax used to solve the problem efficiently. We have provided series of tests exploiting the RUBiS benchmark. From that we observed that SQL boundaries are not harmful in the context of web applications and our model assumptions prove to be correct. We remark the significant reduction of performed database statements. The presented approach is more efficient than *time-to-live* techniques and does not allow serving data which is not up to date. When com-

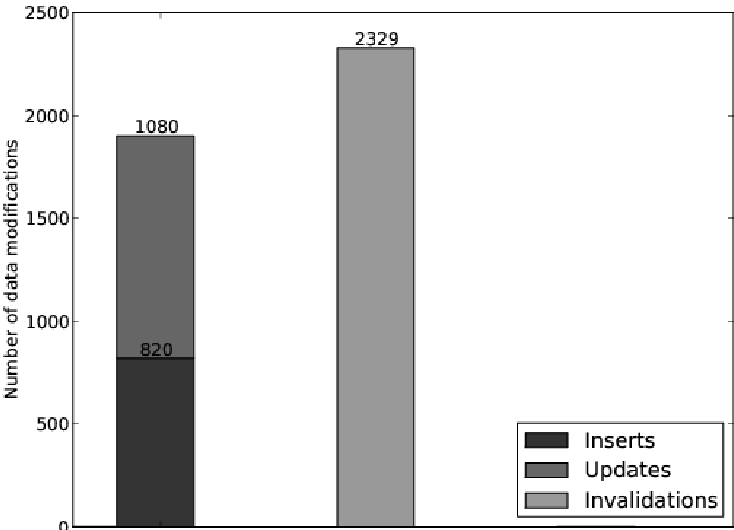

Fig. 10. The number of cache invalidations in different models.

pared to the template approach several improvements need to be stated. Firstly, it allows join and aggregation in select statements which is very important since many aggregation functions are used in the modern web applications to provide frequent counters displayed on websites. Secondly, template based approaches need to know all performed statements classes in advance since the evaluation of invalidation rules is time consuming. Our dependency graph can be easily updated at any time since adding or removing vertices does not require complex operations.

When compared to maintaining materialized views our mechanism does not exploit any knowledge of cached data and its structure. As the invalidation policy does not rely on the cached data and its structure, it allows storing semi-structured data. The future work may involve caching whole HTML code fragments. This can be also understood as an interesting consistency mapper between database and websites components for storing the current HTML.

7 PropScale: an update propagator service for a joint storage

In this chapter we describe *PropScale*, i.e. our proof-of-concept implementation of the update propagator for joint scalable storage systems. We present the architecture of the propagator and design issues that have arisen including synchronization, transactional properties and fault tolerance.

7.1 System architecture

We have implemented *PropScale* in Java 6. We believe that Java suits best our needs because of its portability. *PropScale* contains drivers to backend storages. All SQL and NoSQL storages do have Java drivers. This does not have to be true for other programming languages. However, clients connecting to the update propagator need not to be implemented in Java.

Propscale API The communication with the propagator API is done by Thrift. Thrift is "a software framework for scalable cross-language service development" [55]. It has been developed at Facebook and became open source in April 2007. In 2008, it entered the Apache Incubator. We have chosen Thrift since it is portable to several languages and it allows automatic source code generation for services. According to [55], it successfully integrates with C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, and OCaml.

Thrift generates code in several languages while the communication is done via a binary protocol. This is a clear advantage over SOAP (Simple Object Access Protocol) [14], which encodes messages into XML.

The API of the update propagator consists of three methods: *insert*, *update* and *remove*. We describe briefly structures that are used when communicating with the propagator. First, we introduce two enumerators that define types of operations on relations and elementary data types.

```
enum UpdateType{
    SET = 1,
    INCR = 2,
    DECR = 3,
    CONCAT = 4
}
enum DataType{
    INTEGER = 1,
    STRING = 2,
    FLOAT = 3,
    DATE = 4,
    BOOL = 5
}
```

Then we define the structure $Value$ which represents values of single attributes sent in requests:

```
struct Value{
 1: string name
 2: string value
 3: DataType type
 4: UpdateType update
}
```

A request contains the name of the updated relation. It also contains the list of modified values (*columns*). Eventually, the *condition* list stores conditions selecting tuples to be updated. Currently, we only allow updates that modify tuples identified by their primary key. This can be extended in the future, thus conditions for updates are stored as a list.

```
struct Request{
 1: string table
 2: list<Value> columns
 3: list<Value> condition
}
```

Each response contains a string message. When adding a new tuple, the created primary key is returned:

```
struct Response{
 1: i16 type
 2: string message
}
```

As stated earlier, our service consists of three methods. Each of them takes only one argument: a structure *Request*.

```
service Propagator{
  Response insert(1: Request arg)
  Response update(1: Request arg)
  Response remove(1: Request arg)
}
```

Using the abovementioned definitions of structures, the Thrift compiler generates code of RPC clients and servers that can communicate across boundaries of programming languages.

Drivers Together with our implementation of the propagator we have also developed drivers for some storages. However, it is easy to build a new *PropScale* driver for a storage. As defined in Section 5.4, the algorithm uses five methods: *addPrimary*, *add*, *modify*, *retrieve* and *delete*. In order to create a custom driver, one has to implement the Java interface *Storage* defined as:

```
public interface Storage {

    public int update(String proj, Column key,
                      List<Column> values);

    public String insert(String proj,
                         List<Column> values,
                         boolean genId);

    public int delete(String proj, Column key,
                      List<Column> columns);

    public void select(String proj, Column key,
                       List<Column> select);
}
```

The first argument of all these methods is a string determining which projection is to be changed. The methods: *update*, *delete*, and *select* contain *Column* as a second parameter. This contains the primary key of the modified tuple and allows its identification in a backend storage. Methods, *update* and *insert*, contain the parameter *values* which is a list of *Column* elements and contains values of attributes. Each *Column* object contains an attribute of the relation it modifies, and a new value. Additionally, in the *insert* method the Boolean *genId* determines if the backend storage has to generate the primary key of the new tuple. If set to *true*, the primary key is generated when adding a new tuple. When it equals *false*, the primary key is contained in *values*. Generating the primary key is not obligatory for a driver and a storage, as long as it is not used as the primary projection of some relation. The last argument of the *select* method contains a list of attributes that are fetched from the backend storage.

The method *delete* contains parameter *columns*, which specifies the list of attributes that need to be cleared. This is not important in case of relational databases, since deleting a tuple may be done by a single delete query without any knowledge of the relation's attributes. However, in case of *key-value* storages a driver may keep each attribute as a separate value. For instance, the *name* of a *user* with a given *user_id* primary key may be stored as a value of the key *user::name::user_id*. Then in order to remove all user data, say *name* and *address*, a list of attributes needs to be specified and this is contained in the last argument of the *delete* method.

We have implemented four drivers: *PostgreSQL*, *MongoDB*, *Redis* and *Solr*. They cover completely different approaches to storing and retrieving data. However the simplicity of the operations run on backend storages allows easily implementing their drivers for the update propagator. This relies on our basic assumption that we only provide a layer for write operations. Designing and implementing such drivers for read operations would be extremely difficult. It is possible to provide reasonable API for most SQL queries (e.g. JDBC). On the other hand, *MongoDB* allows to send a Map/Reduce recipe as a query and execute it on the server side. According to this, there is no good option for providing read and write API's at once.

Thus, the requirements of *PropScale* are fairly simplistic. Integrating a new storage system with *PropScale* is therefore relatively easy.

7.2 Synchronization and Multithreading

PropScale works in a multithreaded environment. It contains threads dedicated to retrieving a Thrift request and generating the response to the client. Threads also run the dependency graph and the list of tasks to be run on backend storages. Each driver has a specified number of dedicate threads that apply changes in the given storage. This allows storing persistent connections and limiting their number.

Additional group of threads is dedicated to logging. When a new request arrives, all changes that are going to be applied on backend storages are logged into a file with sent tasks. Then, after being applied, they are also logged to the file with finished tasks. The system allows configuring how frequently the logs are written to the disk. This can be done at each request. In this case the disk file always stores the current log. However, in order to reduce the time spent on disk writes, log flushes can be aggregated into batches. Then, the memory log is dumped to the disk at specified intervals.

Logging is applied in order to prepare the system for sudden breakdown. In such an event, it is possible that some backend storages are updated while others are not. When the system restarts, it compares two logs with registered tasks and tasks that have been done, and executes tasks that have been registered but not accomplished. This protects *PropScale* against data loss in case of a sudden breakdown.

The other possible threat to the system is implied by the differences between backend storages. Problems arise because of their dissimilar characteristics. Some storages allow extremely fast writes while other may suffer from slow writes. *MongoDB* is an example of a fast writer, while *Solr* is slow in this respect. Suppose an operation which adds a tuple into a primary projection run by *MongoDB*. Then the operation also adds this tuple to *Solr*. The response to the client is asynchronous and may be sent before applying changes to *Solr*. As the workload increases, this can lead to a hazardous situation where the number of unfinished *Solr* tasks gets too high. For a short time this can cause a delay between data modifications in

different storages. As a long term effect, the system may be unable to finish all *Solr* tasks and ends up with inconsistent data between storages.

In order to avoid such hazardous situation, we apply the Java mechanism implemented by the class *SynchronousQueue*. This class implements a blocking queue that blocks a thread putting an element into the queue until there is a thread registered to pull from it. As a result, the size of such a queue always equals zero. When created with the special argument, an instance of the *SynchronousQueue* behaves as a FIFO queue. The only difference is that it blocks threads trying to add new elements.

In our implementation, we use *SynchronousQueue* to store tasks for backend storages' drivers. Before a response is returned to the client, the system assures that all asynchronous tasks for backend storages have been successfully added to a *SynchronousQueue*. This assures that driver threads have fetched tasks for execution. Then a response is returned to the client. This assures that execution of all tasks in backend storages have been started and prevents from the abovementioned problem. When the workload increases significantly, the client response time may increase. As the number of threads retrieving a client request and sending the response is limited, the unacceptable workload results with no more connections available. This is a clear signal for the application that something goes wrong with the designed architecture for the given workload. This protects consistency of data in backend storages even if the configuration of the schema and backend storages is wrong.

8 The benefits of applying PropScale

In this chapter we describe sample scenarios of applying *PropScale*. They are provided with experimental results that evaluate the improvement when compared to traditional methods. In our tests we have not used YCSB [20]. Although it is a valid benchmark for cloud data storages, it does not allow more complicate scenarios which have been used for benchmarking *PropScale*.

In our tests we assume each storage is hosted separately on a single Intel i5-2400 machine with 3.10GHz CPU's and 4GB RAM. Distinct machines of the same type have been used to generate the workload and for *PropScale* web service.

8.1 Introduced overhead

Before evaluating the benefits, we check if the additional computation performed by *PropScale* introduces a significant overhead. In this test, we assume there is only one relation *book* in the system and the only projection is stored within a single PostgreSQL database which contains all attributes of *book*. We add new tuples to the relation and test the overhead of the propagator web service layer. The results are presented in Figure 11. The diagram presents the total time of the client request to the propagator compared to the time of the operation in PostgreSQL. This allows comparing the overhead within different workloads. Figure 11 shows that the overhead introduced by the propagator web service is independent of the system workload and remains at the acceptable level.

8.2 Offset between updates in storages

In this test we evaluate the offset between actual update operations performed in the storages. When applying changes on multiple storages *PropScale* updates at first the specified projection. It is called the primary projection of the relation and is predefined in the schema. As the storage with the primary projection is updated, the response is returned to the client. Other storages are updated asynchronously.

As the benchmark environment, we have chosen the bookstore example with relations: *book*, *user*, *book_sold* and *book_comment*. The

database stores books' data, data on customers, information on sold items and book comments put by users. In our benchmarks, we have assumed 1 million of books and users, 5 millions of sold items and 10 millions of comments.

In this test, we examine a scenario when the book data are distributed among different databases: PostgreSQL for storing financial data, MongoDB as a scalable storage with book information and Redis as fast storage for simple book statistics. The projection stored in PostgreSQL has been defined as the primary projection of the relation *book*. Thus changes are first applied to PostgreSQL. We measure the time until they are applied in MongoDB and Redis. Figure 12 shows the results. Again we can observe that the tested offset did not grow when the workload increased. The relational schema in this test is the same as in the previous one, where we have inserted all book data into RDMBS. On the functional level this is the same. Although storing the same data in different storages introduced extra workload, the system runs more than 12000 operations per second, i.e. more than in the first test where only PostgreSQL database was used.

8.3 PropScale for cloud integration

Cloud databases became an interesting issue in recent years. Several companies outsource their databases to external services like Amazon SimpleDB. In the bookstore example, it is worth storing book comments in a cloud. In that case, one does not have to take much care of scalability issues and service layer agreement is clearly defined. However, companies may be reluctant to store their financial or customer data outside of their company's system. Thus, the business critical data are kept in a local storage, while outsourcing less-critical data to cloud database suppliers. The research described in [43] compares the cost of a single request from different cloud database providers measured under a different workload. External cloud databases are cheaper (per request cost), when the number of requests increases. According to this, it may be a business decision to move frequently read data to cloud data stores while leaving the rest in the local database. In that case the problem of integrating storages into a single system arises and *Propscale* solves it. It tracks the

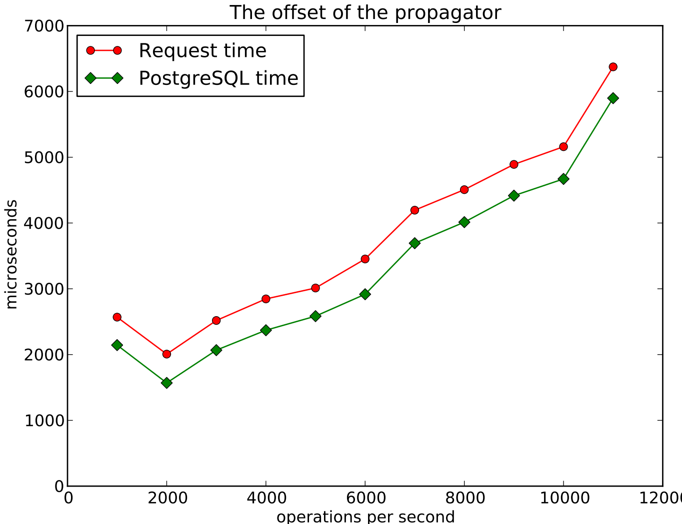

Fig. 11. The results of the experiment on the overhead of the propagator. The gap between the curves is the additional time above the PostgreSQL operation needed when using the propagator.

data stored in different locations and makes sure that the storages constitute a consistent database.

8.4 Custom statistics

The real advantage of *PropScale* is the ability to define frequently accessed statistics and keep them in storages with quick data accesses like *key-value* databases. *PropScale* allows directly defining which statistics have to be stored. To evaluate this feature we introduce a benchmark built on a community forum. Suppose the system contains three relations: *forum*, *thread* and *post*. In our benchmark we assume 100 fora, 10K threads with 100 posts each, which results in 10 million posts.

We examine the following data access patterns, which we believe are the most frequent: (1) adding a single post, (2) showing the list of fora, (3) retrieving the list of threads within a forum and (4) listing posts within a thread. When retrieving the list of fora and threads, the system needs to read the information on the last post in a thread/forum, its author and the date when an item has been

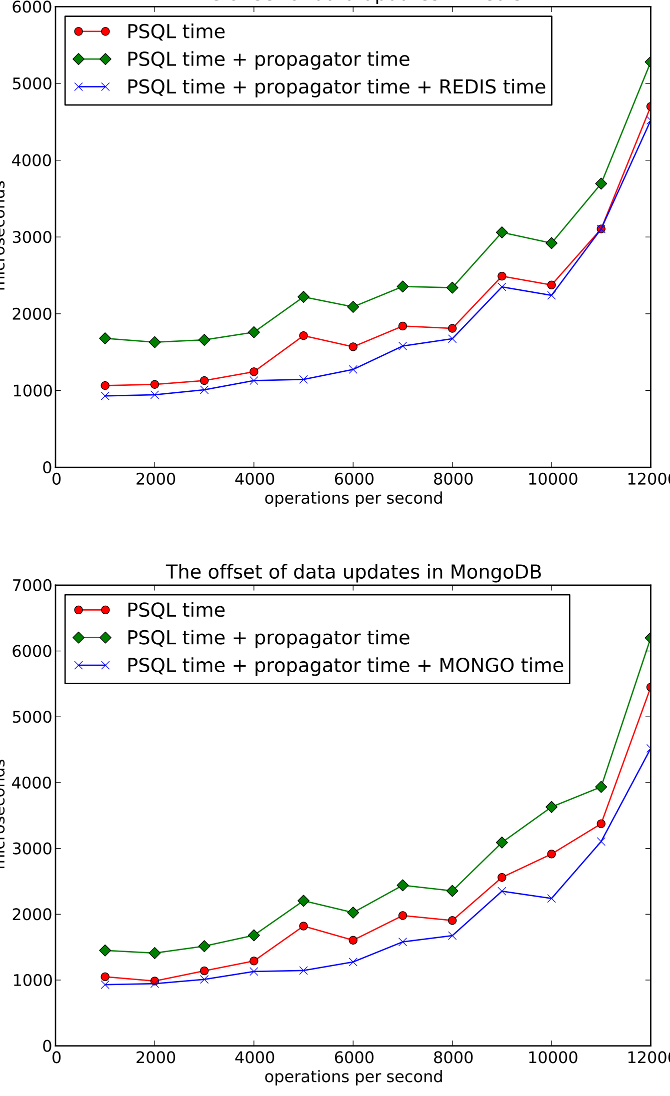

Fig. 12. The offset between updating data in the primary storage and in secondary storages. The bottom curve represents time spent in the primary storage PostgreSQL. The middle curve includes the offset introduced by the propagator. The upper curve presents the total time till all changes appeared in secondary storage.

added. Moreover, when listing fora/threads, we need to retrieve the number of contained threads/posts. In our benchmark we examine the queries that add a single post and retrieve the first 20 posts of a thread. These three queries are run randomly by the benchmark with the probability of: 10% for adding posts and 45% for retrieving posts and 45% for listing threads.

In the naive architectural choice, we store data in MySQL in the third normal form without any redundant columns. In that case queries that add or retrieve posts perform well, however retrieving threads with all needed data is significantly slower than the acceptable level. In that case the achieved throughput is 5 operations per second when the workload is generated by a single thread. At this workload the average time for retrieving a thread is 376 milliseconds, while adding and retrieving posts requires 4 and 49 milliseconds respectively.

The obvious countermeasure is to add redundant data to the system. This can be done by adding extra columns to the relational database or by storing the redundant data in other storage. We compare these two options. The second one is implemented by the *PropScale* and the redundant data is stored in *Redis*. The clear advantage of the second choice is that it scales well. The disadvantage of the first choice is that when a new post is added, the database needs to update the corresponding thread and the forum tuple. We know that query caches implemented in relational databases perform well when data are not modified. However the frequent read and write accesses to the same tuples significantly reduce the performance of most RDBMS. This has been confirmed in our tests. The results are presented in Figures 13 and 14. It is even hard to present them on a single graph. When storing data only in MySQL the request time started to increase at the rate of 2000 ops per second and the next test with 2400 ops per second failed. The similar test with *PropScale* performed well until more than 3000 ops per second and failed on 5200.

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

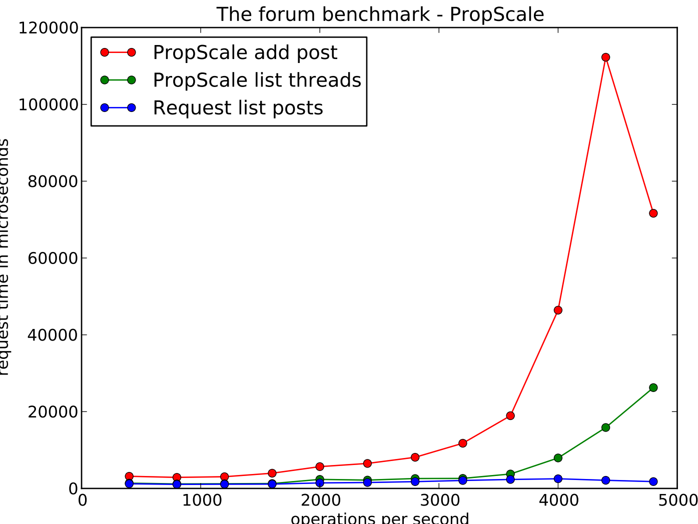

Fig. 13. PropScale correspond to the forum application built on MySQL/Redis integrated with PropScale respectively. The end of graph determines the maximal number of operations per second such that, the benchmark did not fail, i.e. the expected number of operations has finished.

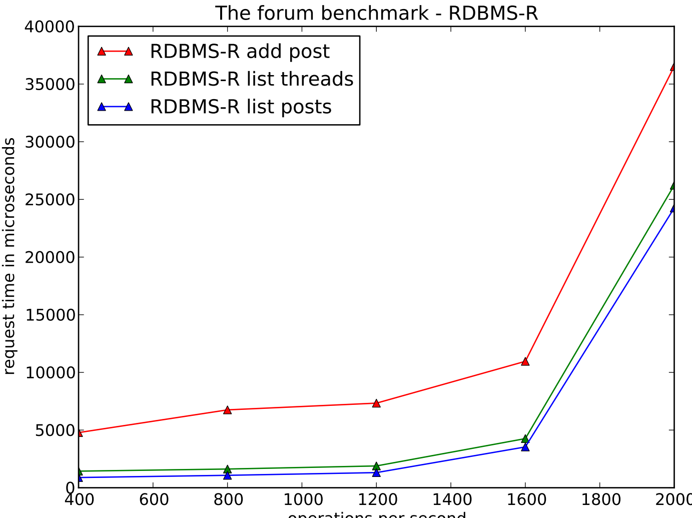

Fig. 14. RDBMS-R correspond to the forum application built on MySQL with redundant columns. The end of graph determines the maximal number of operations per second such that, the benchmark did not fail, i.e. the expected number of operations has finished.

the number of different update types can grow exponentially, with the number of attributes, the implementation of the algorithm can be optimized to precompute the update of each attribute separately. The update propagation mechanism is scalable since the graph size does not depend on the data size.

9 Conclusion

According to the CAP theorem [10], there exists a trade-off between consistency and availability. Some storages like relational databases provide ACID properties, but do not scale well. On the other hand, possibly inconsistent NoSQL storages provide high availability of the storage. In this thesis we have presented a solution that allows tuning the trade-off in a better way. Using our solution one can integrate dissimilar storage types with different data architectures: RDBMS, *key-value* storages and others. Within our model, an application's data can be easily split into smaller chunks and each of them can be provided with a storage solution of appropriate characteristics.

Creating a scalable database storage is a valid research problem. We have focused on web applications which have given us additional assumptions about the data model: (1) read and update operations are known in advance, (2) data accesses are read dominant and (3) numerous consistency levels are needed in different contexts. We believe that these assumptions facilitate developing a better trade-off.

We have presented the scalable joint storage system which is based on various underlying storages. It propagates updates to keep all data copies consistent with each other. We have shown the architecture and described a proof-of-concept implementation. 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 specific data.

We believe that it allows building scalable web applications at 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. Agrawal, A. E. Abbadi, S. Das, and A. J. Elmore. Database scalability, elasticity, and autonomy in the cloud - (extended abstract). In J. X. Yu, M.-H. Kim, and R. Unland, editors, *DASFAA (1)*, volume 6587 of *Lecture Notes in Computer Science*, pages 2–15. Springer, 2011.
3. A. Aksyonoff. Introduction to search with Sphinx: From installation to relevance tuning, 2011.
4. M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *VLDB*, pages 718–729, 2003.
5. K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Dbproxy: A dynamic data cache for web applications. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 821–831. IEEE Computer Society, 2003.
6. J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 1st edition, 2010.
7. B. N. Bershad and J. C. Mogul, editors. *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. USENIX Association, 2006.
8. J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In C. Zaniolo, editor, *SIGMOD Conference*, pages 61–71. ACM Press, 1986.
9. C. Bornhövd, M. Altinel, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Dbcache: Middle-tier database caching for highly scalable e-business architectures. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *SIGMOD Conference*, page 662. ACM, 2003.
10. E. A. Brewer. Towards robust distributed systems (abstract). In G. Neiger, editor, *PODC*, page 7. ACM, 2000.
11. M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In Bershad and Mogul [7], pages 335–350.
12. Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. ES^2: A cloud data storage system for supporting both OLTP and OLAP. In S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, editors, *ICDE*, pages 291–302. IEEE Computer Society, 2011.
13. E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In M. Ibrahim and S. Matsuoka, editors, *OOPSLA*, pages 246–261. ACM, 2002.
14. E. Cerami. *Web Services Essentials*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.
15. J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *INFOCOM*, pages 294–303, 1999.
16. 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.
17. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

18. K. Chodorow and M. Dirolf. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage.* O'Reilly, 2010.
19. S. Chu. MemcacheDB: A complete guide, Mar. 2008.
20. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In Hellerstein et al. [39], pages 143–154.
21. C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: a database service for the cloud. In Franklin [29], pages 235–240.
22. S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB*, pages 330–341. Morgan Kaufmann, 1996.
23. S. Das, D. Agrawal, and A. E. Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In Hellerstein et al. [39], pages 163–174.
24. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
25. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007.
26. Domain-Driven Design Community. Aggregate, Mar. 2012.
27. Flexviews. Incrementally refreshable materialized views for MySQL, Jan. 2012.
28. M. Fowler. Aggregate oriented database, Jan. 2012.
29. M. Franklin, editor. *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings.* www.cidrdb.org, 2011.
30. C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable consistency management for web database caches. Technical report, School of Computer Science, Carnegie Mellon University, 2006.
31. 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.
32. S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In M. L. Scott and L. L. Peterson, editors, *SOSP*, pages 29–43. ACM, 2003.
33. S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
34. H. Guo, P.-Å. Larson, and R. Ramakrishnan. Caching with 'good enough' currency, consistency, and completeness. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, and B. C. Ooi, editors, *VLDB*, pages 457–468. ACM, 2005.
35. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In P. Buneman and S. Jajodia, editors, *SIGMOD Conference*, pages 157–166. ACM Press, 1993.
36. H. Gupta and I. S. Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Inf. Syst.*, 31(6):435–464, 2006.
37. E. Hatcher and O. Gospodnetic. *Lucene in Action (In Action series).* Manning Publications Co., Greenwich, CT, USA, 2004.
38. P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141. www.crdrdb.org, 2007.
39. J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, editors. *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010.* ACM, 2010.

40. A. Iyengar, J. Challenger, D. M. Dias, and P. Dantzig. High-performance web site design techniques. *IEEE Internet Computing*, 4(2):17–26, 2000.
41. A. V. Jonas Partner. *Neo4j in Action*. Manning Publications, 2012.
42. D. Katsaros and Y. Manolopoulos. Cache management for web-powered databases. In *Web-Powered Databases*, pages 203–244. Idea Group Publishing, 2003.
43. D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In A. K. Elmagarmid and D. Agrawal, editors, *SIGMOD Conference*, pages 579–590. ACM, 2010.
44. 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.
45. 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.
46. A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.
47. R. M. Lerner. At the forge: Redis. *Linux J.*, 2010(197), Sept. 2010.
48. P. Leszczynski and K. Stencel. Consistent caching of data objects in database driven websites. In B. Catania, M. Ivanovic, and B. Thalheim, editors, *ADBIS*, volume 6295 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2010.
49. J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In Franklin [29], pages 123–133.
50. W.-S. Li, O. Po, W.-P. Hsiung, K. S. Candan, D. Agrawal, Y. Akca, and K. Taniguchi. Cacheportal ii: Acceleration of very large scale data center-hosted database-driven web applications. In *VLDB*, pages 1109–1112, 2003.
51. Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *SIGMOD Conference*, pages 600–611. ACM, 2002.
52. A. Manjhi, P. B. Gibbons, A. Ailamaki, C. Garrod, B. M. Maggs, T. C. Mowry, C. Olston, A. Tomasic, and H. Yu. Invalidation clues for database scalability services. In R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, editors, *ICDE*, pages 316–325. IEEE, 2007.
53. I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In J. Peckham, editor, *SIGMOD Conference*, pages 100–111. ACM Press, 1997.
54. D. Obasanjo. When databases lie: Consistency vs. availability in distributed systems, Oct. 2007.
55. J. R. Ronald Cohn. *Apache Thrift*. VSD, 1st edition, 2012.
56. P. Saab. Scaling memcached at Facebook, Dec. 2008.
57. K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: asynchronous incremental view maintenance. *SIGMOD Rec.*, 29(2):129–140, May 2000.
58. A. P. Sheth and M. Rusinkiewicz. Management of interdependent data: Specifying dependency and consistency requirements. In *Workshop on the Management of Replicated Data*, pages 133–136, 1990.
59. D. Smiley and E. Pugh. *Apache Solr 3 Enterprise Search Server*. Packt, 2011.
60. M. Stonebraker. SQL databases v. NoSQL databases. *Commun. ACM*, 53:10–11, April 2010.
61. M. Stonebraker and U. Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

62. M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.
63. S. Tiwari. *Professional NoSQL*. Wrox Programmer to Programmer. John Wiley & Sons, 2011.
64. N. Tolia and M. Satyanarayanan. Consistency-preserving caching of dynamic database content. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *WWW*, pages 311–320. ACM, 2007.
65. P. Valduriez. Principles of distributed data management in 2020? In A. Hameurlain, S. W. Liddle, K.-D. Schewe, and X. Zhou, editors, *DEXA (1)*, volume 6860 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2011.
66. W. Vogels. Eventually consistent. *Commun. ACM*, 52:40–44, January 2009.
67. C. Wasik. *Managing Cache Consistency to Scale Dynamic Web Systems*. PhD thesis, University of Waterloo, 2007.