

Charles University in Prague
Faculty of Mathematics and Physics

Habilitation Thesis



**Efficient Processing of Semi-Structured and Large
Data in Distributed and Parallel Environment**

Filip Zavoral

Prague, September 2015

Acknowledgement

I would like to thank the head of the Department of Software Engineering Tomáš Skopal and his predecessors Jaroslav Pokorný, Peter Vojtáš, and František Plášil. They always encouraged my work and valuably supported the research of me and my colleagues for many years.

The work presented here would not be possible without a tight cooperation with my colleagues. Among many of them, I would like to thank especially Jakub Yaghob and David Bednárek for their long time every day common work on many projects, some of them resulted in papers presented in this thesis. I would like to thank Martin Kruliš, Zbyněk Falt, Jiří Dokulil, and Jana Dvořáková for their activity, enthusiasm, and high technical skills that significantly enhanced our research and mutual cooperation.

Beside these explicitly named, I would like to thank all the co-authors of my publications and many researches, colleagues, and Ph.D. students who contributed to common projects; their work without any doubt has valuably influent presented results.

Table of Contents

1. Introduction.....	4
1.1 Background	4
1.2 Presented Papers	7
1.3 Data Integration Using DataPile Structure.....	9
1.4 Using Input Buffers for Streaming XSLT Processing.....	10
1.5 High-Level Web Data Abstraction Using Language Integrated Query	12
1.6 Parallel SPARQL Query Processing Using Bobox.....	13
1.7 Highly Scalable Sort-Merge Join Algorithm for RDF Querying.....	14
1.8 Resistance of Trust Management Systems against Malicious Collectives.....	15
1.9 Metro-NG: Computer-Aided Scheduling and Collision Detection	16
1.10 Locality Aware Task Scheduling in Parallel Data Stream Processing	17
2. Data Integration Using DataPile Structure	19
3. Using Input Buffers for Streaming XSLT Processing	31
4. High-Level Web Data Abstraction Using Language Integrated Query	39
5. Parallel SPARQL Query Processing Using Bobox	51
6. Highly Scalable Sort-Merge Join Algorithm for RDF Querying	67
7. Resistance of Trust Management Systems against Malicious Collectives	77
8. Metro-NG: Computer-Aided Scheduling and Collision Detection.....	89
9. Locality Aware Task Scheduling in Parallel Data Stream Processing.....	119
10. Conclusions and Future Work	135

Chapter 1.

Introduction

The main topics of this thesis consist of methods, technologies, data structures, and algorithms used for efficient processing of large data sets. Although most of the presented results are applicable for large variety of data, the essential motivation was efficient processing of semantic data.

The thesis consists of a commented collection of eight papers which appeared in international journals and conference proceedings disseminated by world-wide publishers.

1.1 Background

The results presented in this thesis are based mainly on three computer science branches: semantic technologies, parallelism, and distributed processing. This combination is implied by the specialization of the author: a long time interest in distributed computing and results from a large scientific project, which was focused on semantization techniques funded by the Czech Science Foundation for several years, were combined with recent advances in the field of parallel processing. This section briefly discusses the scientific background in these branches.

Semantic Technologies

Web technologies belong to the most popular areas of computer science today. The amount of data stored on the web has been growing rapidly and applications providing access to this data became a part of everyday life. Examples of these applications are searching engines such as Google and specific web applications driven by business or by social needs such as eBay, Amazon, Flickr, YouTube, or Facebook. They attempt to add some semantic value to the web resources in order to provide a possibility to query them by users. This is achieved by categorization of accessed resources and by determining mutual relationships among these resources.

Several approaches can be recognized in the web based research and development to enhance its functionality. Among the most prominent are Web 2.0, the social web, the semantic web and the web services [atz10]. The first two are oriented to human for their more comfortable access to the web resources, whereas the last two aims to support both human and machine processing.

The web semantization initiative aims to create a universal medium for the data exchange via machine-understandable data. One of the main approaches to making data

machine understandable is to annotate them according to an ontology. The machine processing of the data is represented by web services that provide standard means of interoperability between different software applications running on a variety of platforms. Web services in the semantic web are supposed to process semantically annotated data in order to enhance the provided functionality. The annotation is supposed to be done by means of binding to the ontology.

One of the long-term issues of web semantization is an insufficient ratio of semantically annotated data on the web. Manually and uniformly annotated data usually appear only in narrow, well organized domains, such as medicine, musea, or libraries. Most of the resources are created for human reading; their automatic capturing and processing is hard and even many of the resources are hidden for machine access (e.g., hidden web or restrictions of robot access). Hence, one of the most severe problems of annotation is the level of automation. The fully automated processing is hard to achieve and the manual processing is expensive and does not scale well.

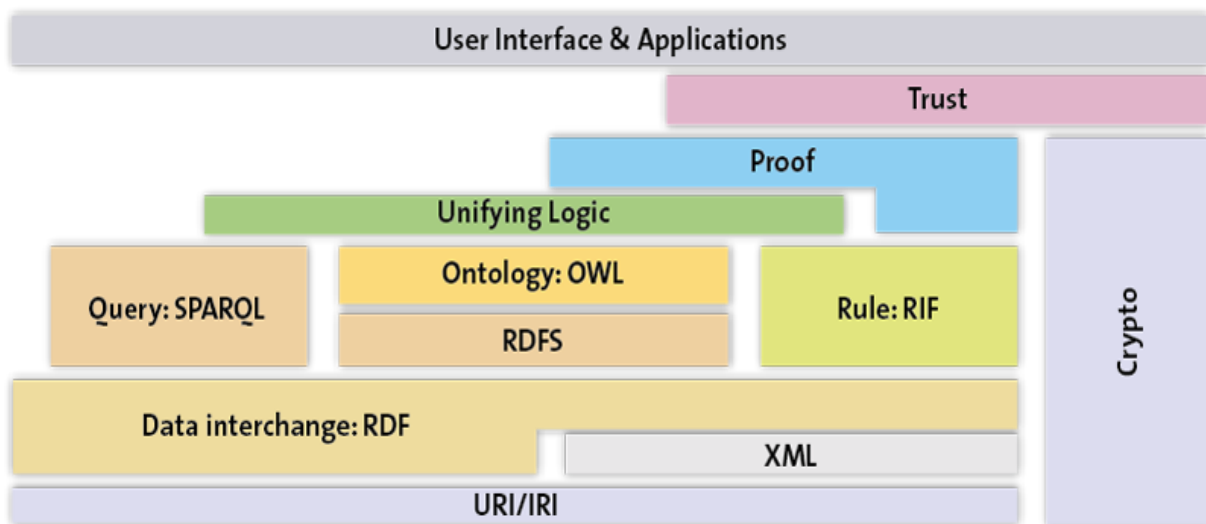


Fig. 1 – Semantic web stack

There are developed and relatively stabilized standards for ontology description (OWL), standardized specialized query languages (SPARQL, RQL, SeRQL, RDQL), and storage systems for data and metadata (Sesame, Jena, OWLIM, RDF-3X, Virtuoso, C-Store and many others). However, most of currently available semantic platforms are limited in performance and scalability. As the amount of RDF data continues to scale, a single machine not able to store entire data sets; therefore, distributed architectures are necessary. Some of these stores support parallel computation of multiple queries (inter-query parallelism), but they do not use the potential of parallel computation of query itself.

Distributed Computing

The field of distributed computing studies methods, algorithms, and technologies applicable to utilizing a wide set of independent computing nodes mutually connected for synchronized execution of tasks spanned across the nodes.

From the architectural point of view, there are many real-world applications of distributed computing today such as distributed data storage and processing, peer-to-peer networks, ubiquitous and agent systems, content-management networks, cooperative computing and problem solving, mobile and sensor networks, cluster and grid high-performance computing, social networking, platforms for cloud computing, and many others.

Such a wide range of diverse architectures and systems usually share similar issues. Due to absence of shared physical memory, various distributed algorithms are needed for synchronizing individual nodes. Higher levels of scalability cannot be achieved without replication, by which resources are copied and placed close to nodes which need them. This technique enables highly efficient utilization of available resources and computing nodes. Nevertheless, it implies additional consistency issues, such as mutual consensus, delivery protocols, replica placement, etc. Moreover, the CAP theorem [gil02] tells us that we simply cannot combine consistency and availability in the presence of network partitions. Therefore, building scalable distributed systems continues to be one of the most challenging tasks in the systems design [ste12].

Parallel Processing

Nowadays, parallel processing is one of the main trends in software development. Wide spectrum of computing equipment from workstations to supercomputers utilizes parallel technologies. The main reason is a significant increase in the processing power enabled by parallelism. The processing capacity of a regular desktop computer with a multicore processor supersedes the computational power of a supercomputer of two decades ago at a fraction of the cost.

Moreover, the emergence of new computational platforms that are parallel in nature can be observed, such as GPGPUs or Intel Xeon Phi accelerator cards. These new technologies encourage programmers to consider parallel processing not only in a distributive way (horizontal scaling), but also within each node (vertical scaling). The parallelism is getting involved on many levels and it is commonly believed that it will play even more significant role in the future.

The issues implied by the introduction of parallel processing have been studied for nearly five decades. Although much progress was made in the areas of parallel architectures, algorithm and software design, major problems remain to be solved. The increasing amounts of processing units and the use of standardized components for the a streamline production of parallel systems comprising of large number of processors call for scalable methods and tools to support the development of software that effectively and efficiently utilizes parallel hardware.

Writing parallel software is a very complex task at present time. Contemporary programming languages are often accommodated to parallel features but their application is not straightforward. To make programming easier, several libraries and compiler extensions have been created, such as OpenMP, MPI library, or Intel Threading Building Blocks (TBB). Although these frameworks are popular for development of parallel processing software, parallel programming is still a complex and error-prone task. Therefore, algorithms, methods, languages, and other approaches for simplifying the development of parallel applications on various architectures are very hot research topics.

1.2 Presented Papers

The collection of papers presented in this thesis can be divided into three main groups according to a type of the most significant presented result.

- [P1] (column-oriented data store for historic records) and [P2] (processing of large XML collections with low memory requirement) propose *data structures* for efficient data processing;
- [P5] (scalable join algorithm for RDF data in streaming systems), [P6] (achieving of mutual trust in distributed processing), and [P7] (automatic detection of structure and semantics of data) present newly proposed *algorithms* for these problems;
- [P3] (efficient access to strongly typed data available remotely), [P4] (implementation of parallel SPARQL engine), and [P8] (high-performance scheduling in streaming systems) present and evaluate *implementations* of tools developed within a scope of our scientific projects.

The following list contains papers comprising the main part of the thesis:

- [P1] *David Bednárek, David Obdržálek, Jakub Yaghob, Filip Zavoral:*
Data Integration Using DataPile Structure,
ADBIS 2005, Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems, Springer Verlag, ISBN: 3-540-42555-1, pp. 178-188, 2005
- [P2] *Jana Dvořáková, Filip Zavoral:*
Using Input Buffers for Streaming XSLT Processing,
DBKDA 2009, International Conference on Advances in Databases, Knowledge, and Data Applications, IEEE Computer Society Press, ISBN: 978-1-4244-3467-1 , pp. 50-55, 2009
- [P3] *Jakub Míšek, Filip Zavoral:*
High-Level Web Data Abstraction Using Language Integrated Query
Intelligent Distributed Computing IV, Springer Verlag, ISBN: 978-3-642-15210-8, pp. 13-22, 2010
- [P4] *Zbyněk Falt, Miroslav Čermák, Jiří Dokulil, Filip Zavoral:*
Parallel SPARQL Query Processing Using Bobox

International Journal On Advances in Intelligent Systems, Vol. 5, Num. 3, ISSN: 1942-2679, pp. 302-314, 2012

- [P5] *Zbyněk Falt, Miroslav Čermák, Filip Zavoral:*
Highly Scalable Sort-Merge Join Algorithm for RDF Querying
Proceedings of the International Conference on Data Technologies and Applications, SciTePress, ISBN: 978-989-8565-67-9, pp. 293-300, 2013
- [P6] *Miroslav Novotný, Filip Zavoral:*
Resistance of Trust Management Systems against Malicious Collectives
Proceedings of 2nd International Conference on Context-Aware Systems and Applications, Springer Verlag, ISBN: 978-3-319-05938-9, ISSN: 1867-8211, pp. 67-76, 2014
- [P7] *David Bednárek, Jakub Yaghob, Filip Zavoral:*
Metro-NG: Computer-Aided Scheduling and Collision Detection
Computing and Informatics, ISSN: 1335-9150, Vol. 34, Num. 2, pp. 1-27, 2015
- [P8] *Zbyněk Falt, Martin Kruliš, David Bednárek, Jakub Yaghob, Filip Zavoral:*
Locality Aware Task Scheduling in Parallel Data Stream Processing
Proceedings of the 8th International Symposium on Intelligent Distributed Computing - IDC'2014, Springer Verlag, ISBN: 978-3-319-10421-8, ISSN: 1860-949X, pp. 331-342, 2014

[P1] received so far 12 citations, [P2] was cited by Michael Kay in his invited talks about the future of XSLT processing. The citations of the rest of the papers are listed in a separate document. [P7] is especially important in the context of Charles University; scheduling of the university lectures is based on the proposed formal model and the developed tools.

The research presented in these papers was included in and partly financed by the following grants where the author was one of the participating investigators:

- Czech Science Foundation - GACR 13/08195, Highly Scalable Parallel and Distributed Methods of Data Processing in e-Science, 2013-2015
- Czech Science Foundation - GACR 202/10/0761, Web Semantization, 2010-2012
- Czech Science Foundation - GACR 201/09/H057, Res Informatica, 2009-2012
- Czech Science Foundation - GACR 201/09/0990, XML Data Processing, 2009-2011
- Ministry of Education of the Czech Republic, grant MSM0021620838, Modern Methods, Structures and Systems of Computer Science, 2005-2011
- National programme of research, Information society project 1ET100300419, Intelligent Models, Algorithms, Methods and Tools for the Semantic Web, 2004-2008
- GAUK 13/472313 Recovery methods for distributed data stream processing, 2013-2015
- GAUK 2010/28910 - Abstraction and Automatic Data Extraction for Semantic Web, 2010-2012
- SVV-2014-260100 Advanced methods and applications of software, data and web engineering, 2014

- Intel Corp. Grant PO#4507012020

The papers present an extracted output of several scientific and software projects. Since all the main research areas: semantic web, distributed systems, and parallel processing belong to the area of experimental computer science, the results could not be achieved by an individual research. Instead, many of colleagues and students participated in these projects where the author of this thesis was one of the main researchers.

The following subsections of Chapter 1 contain a brief summary of the basic ideas and scientific contributions of these papers. Each of these papers is presented in a separate chapter starting with a copy of the front page of the publication in which the paper was published.

1.3 Data Integration Using DataPile Structure

The objective of the first paper [P1] originated from a mid-range real-world project – the design and development of an information system based on replication and synchronization of data coming from a large number of different data sources. We proposed a *DataPile* structure with the following objectives: an efficient storage of historical versions of the data, straightforward adaptation to global schema changes, separation of data conversions and replication logic, and a support for an evaluation of data relevance.

The main idea of this structure is data verticalization. Instead of keeping all the data in separate tables (in one or more DBMS) with many attributes, the data are stored in one table without any “well-formed” internal structure or hierarchy. Each row in the pile represents one attribute whose value is valid during certain interval of transaction time. Such data entry is supplemented by several system values for proper implementation of the functionality needed such as relevance values and time stamps. The overall logical structure of the data is kept in few separate metatables.

The structure described avoids some of the typical integration problems of the information systems: the number of relational tables used does not grow with an expansion of the system, data changes are preserved with minimal overhead, and the extensibility of the system (e.g. defining new entities or attributes) is achieved by inserting new rows into metatables.

During the evaluation of the prototype implementation, we have discovered some disadvantages of this approach. The export and matching processes were not very efficient due to a relatively complex matching algorithm. Fortunately, the time complexity is not very important in everyday life because number of data changes is orders of magnitude smaller in magnitude in comparison with the data volume of the initial migration.

Another challenge was the construction of application-level queries. Since the structure of the central repository makes construction of such direct queries more complex, the concept of caches was introduced and queries to operational data are performed on the caches instead of the DataPile itself.

The project showed that the DataPile approach is suitable for certain class of large applications where data warehousing is coupled with maintaining consistency of local databases. In this class of applications, the drawbacks mentioned above are outweighed by integration of data warehousing features with the support for the data replication, synchronization, and cleaning using back-propagation.

Later, we recognized that, besides the data integration, such structure can be used also in systems that keep the data with relaxed internal structure and different levels of relevance. Since this characterization is very close to the semantic data processing, we used the modified principle of verticalization for the design of a semantic data storage [dok07].

The basic idea of the proposed data store (i.e., data verticalization) employs a similar principle as column-oriented databases. Nevertheless, one of the first (and probably the most cited) paper [sto05] about this topic was published about one year after finishing our project, thus the design of the DataPile was not influenced by these publications.

1.4 Using Input Buffers for Streaming XSLT Processing

During our work with large semantic data we have recognized a strong need for an efficient processing of large XML data sets such as the RDF and OWL metadata, the output of external gathering tools, the result sets of semantic querying, or the data streams of semantic services.

The work described in this paper as well as in several preceding papers is based on the previous theoretical research on the formal models of the streaming processing made by Jana Dvorakova [dvo07]. In our mutual cooperation, we extended the results into practically implemented algorithms and developed a framework for the efficient implementation and evaluation of different classes of analyzing and transformation algorithms.

We introduced Xord - a framework for efficient XSLT transformations. It enables to develop streaming algorithms that differ in time/space complexity and their applicability. In turn, we proposed several algorithms with clearly characterized transformations classes.

The main contribution of this paper [P2] is using reading buffers for streaming transformations. Although the previously proposed transformation algorithms are highly memory effective, the class of possible transformations is quite restricted. The most important restriction is the order-preserving condition - the ordering of the output nodes must follow the ordering of the input document. In this paper, we present a BUXT analyzer (called according to the underlying formal model - Buffering XML Transducer) and a transformation algorithm that overcome these limitations. Some parts of the input document can be stored in buffers for future processing so that the ordering of the subtrees generated to the output can be independent to the ordering of the input document.

We designed and implemented the BUXT transformer which is able to process all top-down XSLT transformations [mar05]. We exactly characterize this class of transformations. Based on the static analysis of schema and XSLT stylesheet, the BUXT analyzer computes the

information about contexts when buffering is needed. The information is provided in the form of schema fragments and passed to the BUXT transformer. Moreover, by examining schema fragments, it is possible to compute maximal amount of memory needed for processing the stylesheet on XML documents defined by a given schema.

We compared the BUXT transformer space complexity against the publicly available tree-based XSLT processors (Figure 4). All the tree-based processors consumed large amounts of memory when processing large XML data regardless the simplicity of the transformation.

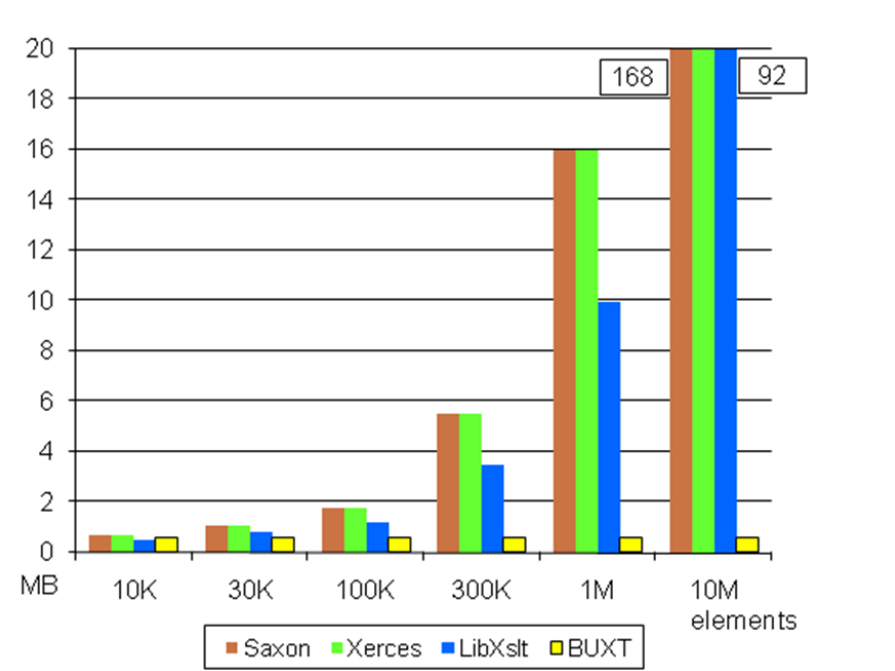


Fig. 4 – Memory consumption of XSLT processors

The evaluation confirmed that the BUXT algorithm basically requires a memory proportional to a depth of the input XML document. Since the document depth is generally not depending on the document size and documents are relatively shallow, the memory requirements for the most XML documents are low and independent on the document size.

Additionally, there is an extra memory required for each buffer bound to a fragment item detected during the transformation. The size of such memory does not depend on the whole input size but on the schema and the XSLT structure. As long as the ordering of the output document remains close to the input document (e.g., the transformation is mostly local), the space complexity remains low. The most typical example of such processing is filtering, mapping, and local reordering of a huge sequence of relatively small subtrees, such as logs, structured data streams, or XML databases.

1.5 High-Level Web Data Abstraction Using Language Integrated Query

Since the amount of data available on the Internet is growing steadily, the automatic data extraction is an important issue for every web-semanticization project. The web pages are mostly designed solely for the human readers. Although recommended formats exist for the web developers, which allow to export information in a machine-readable format (RSS, RDF, or OWL), vast majority of the web pages does not contain such semantic or structural description. The main objective of scraping or extraction frameworks and applications is to retrieve desired information into structuralized form; they periodically download all requested web resources and export specific or requested values into a local storage like database or XML. Two main categories of scraping approaches can be identified:

- Specific extraction mechanisms are implemented as a *standalone large scale application* that provide methods for extracting data from various sources and several options how to save the results. The tasks are configurable in a declarative way, usually using a GUI or scripting. These solutions have high requirements for storage capacity and the whole extraction process can be quite time consuming.

- Programmers can take benefits from *extraction frameworks*. Using such libraries programmers have to take care about all the processes. The implying advantage is a possibility to develop more customized and optimized extraction. For example, such application can modify extraction parameters during runtime or it can download only specific parts of the web sites.

Most of these solutions assume extracting information into a local storage. Such a behavior is sometimes not desirable, because most of the downloaded information is not subsequently used. Downloading everything locally causes high requirements for storage capacity and difficult updates of already extracted information. Moreover, many implementations solve the updates simply by re-downloading all the data from the beginning. The other issue of such frameworks is the efficiency of development. The data are usually represented as unstructured strings, binding to language data types is weak, no syntax checking is possible at compile time etc.

The main contribution of the paper [P3] is the design of the high-level integration of tools for web information extraction into production software. The framework LinqToWeb benefits from advantages of previously described approaches. The extraction tasks can be defined in a declarative way, while the programming interface uses type safe object model representing the abstraction of the web resources. The tasks are compiled, thus the performance of the extraction is maximized. The programmer can use web resources in the same way as data in local memory. In contrast to contemporary solutions, local storage is not used explicitly, but only as a transparent caching mechanism. Moreover, particular data items are accessed only when requested, which makes the extraction process much more efficient since the data access is not delayed by long extraction queues. The high-level object oriented approach takes benefits of modern language features such as Language Integrated Query (LINQ) or code sense capability provided by development environments automatically.

The architecture is inspired by LinqToSQL [box07] integration. The main principle is based on generating strongly typed object model. The description of data sources is used for generating type-safe objects that encapsulate all possible use-cases of the data. The complete process of generating objects is automatically performed by the development environment - every time the source data description is changed, the programmer works with up-to-date objects implementation. Generated code file becomes a part of the program sources, so the compiler is able to perform type checks and optimizations.

The description of each data source consists of the structure of information and the extraction tasks that collect it. A special-purpose declarative language was designed to describe both parts. Its main features include simplicity, declarative task description, procedural processing logic, intuitive object oriented interface, and support for inherent and transparent parallelism.

Using this framework, development of web-based applications such as data semantization tools is more efficient, type-safe, and the resulting product is easily maintainable and extendable.

1.6 Parallel SPARQL Query Processing Using Bobox

SPARQL as a query language for RDF is widely used in semantic web databases. Several database engines are capable of evaluating SPARQL queries such as SESAME, JENA, Virtuoso, OWLIM, or RDF-3X, which is currently considered to be one of the fastest single node RDF-store. One way of improving their performance is the utilization of modern, multicore CPUs in parallel processing. These stores support concurrent computation of multiple queries; however, they do not utilize the potential of parallel computation of individual queries.

The Bobox framework was designed to support development of data-intensive parallel computations [bed12]. The main idea behind Bobox is to divide a large task into many simple tasks that can be arranged into a nonlinear pipeline while preserving transparency of the distribution logic. The tasks are executed in parallel and the execution is driven by the availability of the data on their inputs. The developers do not need to be concerned about technical issues such as synchronization, scheduling, or race conditions. The system can be easily used as a database execution engine; however, each query language requires its own front-end that translates a request (query) into a definition of the structure of the pipeline that corresponds to the query.

In the paper [P4], we present a tool for efficient parallel querying of RDF data using SPARQL build on top of the Bobox framework. We provide a description of query processing using SPARQL-specific parts of the Bobox and provide results of benchmarks using the SP2Bench [sch08] query set and data generator.

SPARQL compiler for Bobox generates execution plans from the code of the queries. During query processing, the compiler uses specialized representation of the query. We proposed the SPARQL Query Graph Pattern Model (SQGPM) as the model that represents query

during optimization steps. It is used to describe relations between group graph patterns. An example of the SQGPM model graphical representation is shown in Figure 6.

With appropriate definition of the operations, this model can be easily transformed into a Bobox pipeline definition – the *execution plan*. It is built during the syntactical analysis and is modified during the query rewriting step. The query parsing step uses standard methods to perform syntactic and lexical analysis according to the W3C recommendation. The input stream is transformed into a SQGPM model in the first step. The transformation also includes expanding short forms in queries, replacing aliases and a transformation of blank nodes into variables. The second step is query rewriting. We cannot expect that all queries are written optimally; they may contain duplicities, constant expressions, etc. Therefore, the goal of this phase is to optimize queries to achieve a better final performance.

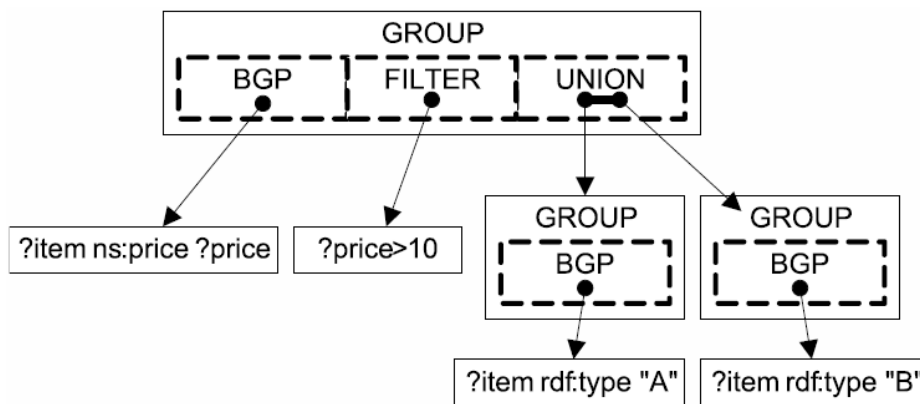


Fig. 6 – SQGPM Model

The main objective of the execution plan generation step is to transform the model into an execution plan. This includes ordering join operations, selecting join types, and applying the best strategy to access the data in the physical storage. The query execution plan is built from the bottom to the top using dynamic programming to search the space of all possible joins. The final execution plan is serialized and passed to the Bobox framework for evaluation. Before the execution, the operators contained in the plan must be replaced by particular implementations of the operators. The standard operators (index scan, filter, sort, various types of joins, etc.) are implemented as optimized Bobox boxes (units of execution, operators).

The performance measurements show that the engine scales very well in a multiprocessor environment. Using SP2Bench queries we have identified that our solution is able to process many queries significantly faster than other engines.

1.7 Highly Scalable Sort-Merge Join Algorithm for RDF Querying

Join is one of the most important database operation since the overall performance of data evaluation engines depends highly on the performance of particular join operations. In the

paper [P5], we propose a highly scalable sort-merge join algorithm for RDF databases. The algorithm is designed especially for streaming systems. Besides the task and data parallelism, it also tries to exploit the pipeline parallelism in order to increase its scalability. The algorithm also handles well skewed data (data with asymmetric or irregular distribution) which may cause load imbalances during the parallel execution [#skewed].

The main idea of the algorithm is splitting the input streams into many smaller parts which can be processed concurrently. The sort-merge join consists of two independent phases – sorting phase that sorts the input stream by join attributes and joining phase.

The algorithm makes use of the fact that the streams are represented as a flow of envelopes (basic units of the data flow). First, the flow of input envelopes is transformed into the flow of pairs of envelopes. The tuples in these pairs can be joined independently in parallel. Dispatch boxes dispatch these pairs among join boxes which perform the operation. When join box receives a pair of envelopes, it joins them and creates the substream of their results. Therefore, the outputs of join boxes are sequences of such substreams which subsequently should be consolidated in a round robin manner by consolidate box. The execution plan is depicted in Figure 7.

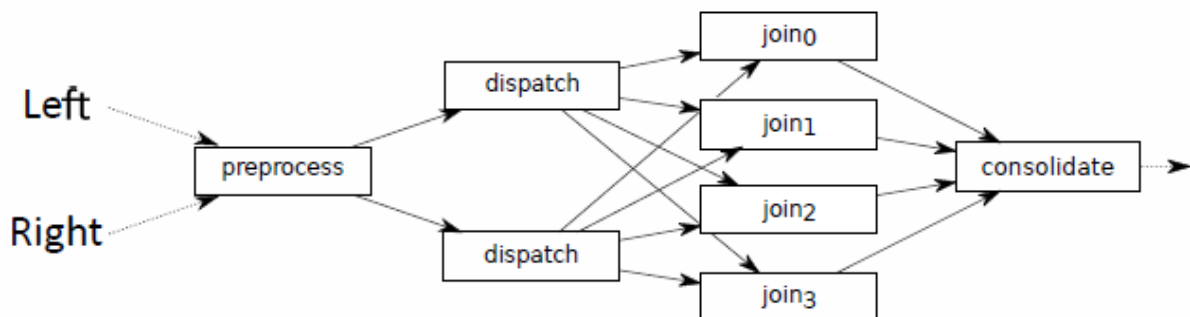


Fig. 7 – Execution plan of parallel merge join

A rich set of benchmarks performed show that the pilot implementation of the algorithm within the Bobox SPARQL engine significantly outperforms other RDF engines such as Jena, Virtuoso, and Sesame in all relevant queries. Moreover, the algorithm behaves well also with skewed data.

1.8 Resistance of Trust Management Systems against Malicious Collectives

In our previous work [nov11], we addressed a problem of mutual trust in large-scale distributed peer-to-peer networks. We identified and classified techniques of malicious peers trying to spread their inauthentic or even harmful resources or to discredit the underlying trust management (TM) system. Beside individual techniques that are sufficiently covered by the majority of contemporary trust management systems, we focused to much more dangerous type of possible attacks - malicious collectives. Several malicious peers know each other and give each other positive recommendations. These recommendations improve reputations of all

members and increase their chance to be chosen as a resource provider. We identified distinct classes of such behavior: malicious spies, malicious camouflage, evaluator collusion, evaluator spies, and evaluator camouflage.

Based on this classification, we developed TM system called BubbleTrust. Its unique idea is separation of a node role as a resource provider and as a transaction evaluator, so that each peer is evaluated for both roles separately.

In the paper [P6], we analyzed the behavior of selected trust management systems against different malicious strategies. Our goal was to verify the effectiveness of various TM systems under sophisticated malicious strategies. We have chosen five contemporary TMS: EigenTrust, PeerTrust, H-Trust, WTR, and BubbleTrust. These systems represent main contemporary approaches in Trust Management.

We also proposed several efficiency criteria which can be evaluated using the P2PTrustSim framework. We expected that malicious peers working in a collective try to use the most effective strategy against TMS currently used in the particular peer-to-peer network. Therefore, the quality of a TMS has to be assessed according to the most successful malicious strategy.

Our results indicate that most of the traditional trust managements are vulnerable to sophisticated malicious strategies. H-index calculation used in H-Trust proved to be vulnerable to traitors. It takes too long to detect traitors and malicious peers are rehabilitated too quickly. The system WTR permits the highest number of bogus transactions from all tested systems, but it is followed closely by PeerTrust and H-Trust. EigenTrust has better results than H-Trust, WTR, and PeerTrust but it has advantage in the form of pre-trusted peers. Our tests proved that it is very difficult to resist against the sophisticated malicious techniques. Especially the calculation of the evaluator rating is susceptible to rigging. The previously published TMSs do not pay as much attention to the evaluator rating as they pay to the provider rating. This must be changed if a TMS should be resistant against the Evaluator Collusion or the Evaluator Spies.

The best results of all evaluated TMSs in our comparison were achieved by BubbleTrust. It has the shortest treason detection time, the longest rehabilitation time and allows only a small portion of bogus transaction under the most successful malicious strategy. As far as we know, it is the only one TMS using global experience as a feedback verification.

1.9 Metro-NG: Computer-Aided Scheduling and Collision Detection

The next paper represents our results from rather different point of view. We explored possibilities of semi-automatic support for university scheduling.

In the paper [P7], we propose a formal model of the objects involved in classroom scheduling at universities which allow a high degree of liberty in their curricula. Using the formal model, we present efficient algorithms for the detection of collisions of the involved objects and for the inference of tree-like navigational structure in interactive scheduling software allowing the selection of the most descriptive view of the scheduled objects.

The problem of general scheduling is very well studied and lots of methods were proposed based on heuristic orderings, genetic/evolutionary algorithms, particle swarm optimization, etc. Nevertheless, such general solvers require exact specification of formal constraints, quality criteria or the fitness functions. These constraints and objective functions are fuzzy and unclear by nature and ordinary users of the system are not able to specify them in a sufficiently formal manner. Thus, the usage of such a solver requires skilled personnel, who are able to formalize the requirements of the users. Moreover, the cost or fitness functions are individually dependent, so they should be adjusted individually per each scheduling object in an ideal scenario.

An alternative approach is to create the schedule manually, using a software application to efficiently manipulate the schedule. The cost of manual or semi-automatic schedule creation may be balanced or even outweighed by the cost of formalization of the constraints. In addition, human-driven scheduling offers diverse advantages such as more effective work with incompletely defined constraints, intuitive recognition of erroneously entered constraints, explaining the rationale behind the particular schedule events, etc.

The paper proposes a formal model for the complex collision detection system which is used to identify situations, like where a group of students should attend two classes scheduled for the same time. In order to use this formal model practically, we propose three algorithms that compute the useful knowledge from the underlying data. The most important algorithm is the detection of collisions among events in the given context. Other two tightly coupled algorithms (the extraction algorithm and the generator of functional dependencies) are used for user navigation in the attribute trees.

The algorithms were used in the application for supporting the whole process of creating a complex university and curricula schedule. Its efficiency and usability suggests that our approach may be applied in many areas where multi-dimensionally structured data are presented and manipulated in an interactive application.

1.10 Locality Aware Task Scheduling in Parallel Data Stream Processing

The last presented paper [P8] targets system-level efficiency in parallel computing. One of approaches of simplification of the design of concurrent processing is the stream data processing. A streaming application is usually expressed as an oriented graph, where the vertices are processing stages and the edges prescribe how the data are passed on. The main advantage of our solution from the perspective of parallel processing is that each stage contains a serial code and multiple stages may be executed concurrently.

One of the key components of these systems is the task scheduler which plans and executes tasks on available CPU cores. Our objective was to design a task scheduler that reflects three important issues:

- the interpretation of the execution plan is data dependent, thus the tasks must be spawned dynamically;
- the tasks should be planned with respect to the overall throughput of the system, since they work on a problem which needs to be solved as whole;
- the scheduling strategy should incorporate important hardware factors such as cache hierarchies and non-uniform memory architectures (NUMA).

During the initialization, the task scheduler detects the configuration and properties of the CPUs. CPU cores which share at least one level of cache are bundled together in logical core groups and a thread pool is created for each group. Each core group maintains one queue of immediate tasks (work that immediately relates to the task being currently processed) per core and one shared queue of deferred tasks (work that is not closely related to the task).

The main paradigm of the scheduling strategy is to emphasize data locality awareness. When the scheduler assigns another work to a thread, it attempts to select a task which is as close as possible to the previous work done by that thread. For this purpose, we exactly defined the distance between any two cores. Using the distance, a set of rules for fetching the most appropriate task was proposed.

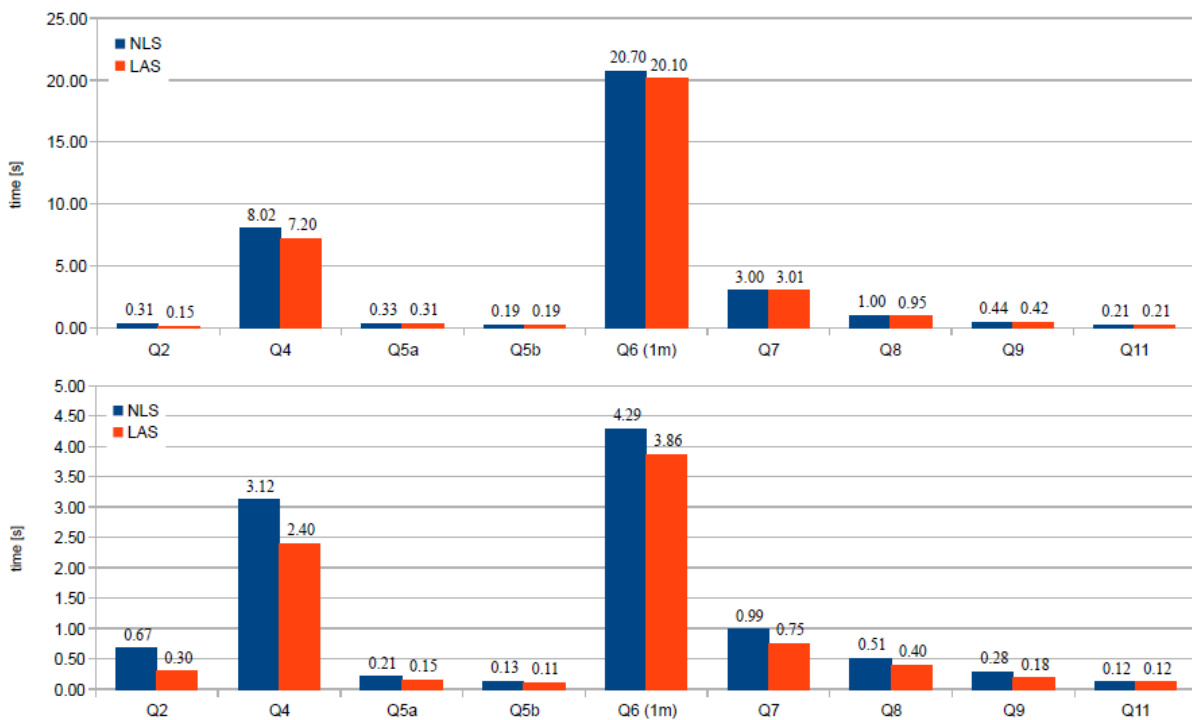


Fig. 8 – Query results on SMP and NUMA

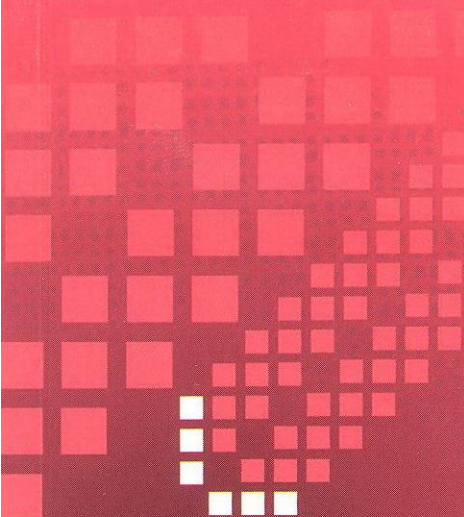
The proposed task scheduler was integrated into the Bobox framework and it was used with our parallel in-memory SPARQL engine [P4]. When applied on a SPARQL benchmark that process RDF data, the system achieved up to 10% speed up on double-processor SMP system and up to 3x speed up on four processor NUMA system (see Figure 8) for selected queries with respect to a standard task-stealing scheduler.

Chapter 2.

Data Integration Using DataPile Structure

David Bednárek, David Obdržálek, Jakub Yaghob, Filip Zavoral

ADBIS 2005, Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems



Advances in Databases and Information Systems

Proceedings of the 9th East-European
Conference, ADBIS 2005
Tallinn, September 12-15, 2005

»»» Johann Eder
»»» Hele-Mai Haav
»»» Ahto Kalja
»»» Jaan Penjam (Eds.)

<http://www.cs.ioc.ee/adbis2005/>

Data Integration Using DataPile Structure

David Bednárek, David Obdržálek, Jakub Yaghob, Filip Zavoral

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University Prague
{david.bednarek, david.obdrzalek, jakub.yaghob, filip.zavoral}@mff.cuni.cz

Abstract. One of the areas of data integration covers systems that maintain coherence among a heterogeneous set of databases. Such a system repeatedly collects data from the local databases, synchronizes them, and pushes the updates back.

One of the key problems in this architecture is the conflict resolution. When data in a less relevant data source changes, it should not cause any data change in a store with higher relevancy.

To meet such requirements, we propose a DataPile structure with following main advantages: effective storage of historical versions of data, straightforward adaptation to global schema changes, separation of data conversion and replication logic, simple implementation of data relevance.

Key usage of such mechanisms is in projects with following traits or requirements: integration of heterogeneous data from sources with different reliability, data coherence of databases whose schema differs, data changes are performed on local databases and minimal load on the central database.

1 Introduction

The concept of data integration covers many different areas of application [3,13]. In this paper, we focus on one kind of applications characterized by the following requirements:

- Data warehousing: The data originated at the local data sources should be replicated into a central repository (data warehouse) in order to allow efficient analytical processing and querying the central system independently of local systems.
- Back-propagation: Any update which occurs in a local database (performed by its local application) should be distributed to other local databases for which this kind of data is relevant.
- History records: The central repository should maintain full history of all data stored therein.

Each one of the requirements forms a well-known problem having well-known solutions [2,8,9,10]; nevertheless, combining the requirements together introduces new, interesting problems, and disqualifies many of the traditional solutions. This paper presents a technique, called DataPile, which combines flexible storage technology

(built upon a standard relational database system) with system architecture that separates the replication mechanisms from the schema-matching and data-conversion logic. Since the approach is inspired by XML techniques rather than relational databases, its combination with modern XML-based technologies is straightforward. Nevertheless, the system is created over relational database system and direct integration with traditional database systems is also possible.

One of the most difficult problems in the area of data integration is handling of duplicate and inconsistent information. The key issue in this problem is entity identification, i.e. determining the correspondence between different records in different data sources [11, 14]. The reality requires that the system administrators understand the principles of the entity matching algorithm; thus, various difficult formalisms presented in the theory [7] are not applicable. Our approach uses a simplified entity matching system which allows the users to specify matching parameters that are easy to understand. Some researchers [6] advice that successful entity identification requires additional semantics information. Since this information cannot be generally given in advance, the integrated system should be able to defer decision to the user. The system should detect inconsistencies and either resolve them, or allow users to resolve them manually. The need for user-assisted conflict resolution induces a new class of problems: The repository should be able to store data before final resolution while their relationship to the real world entities is not consistent. Consequently, the system should be able to merge entities whenever the users discover that the entities describe the same real-world entity, and, conversely, to split an entity whenever the previous merge is found invalid. Under the presence of integrity constraints and history records, this requirement needs special attention.

The relationship between the global system and local database is usually expressed using the global-as-view and local-as-view approaches [5]. In our system, a mixture of these methods is used depending on the degree of integration required.

Maintenance of history records falls in the area of temporal databases and queries, where many successful solutions are known [1, 4, 12]. The theory usually distinguishes between the valid time, for which the data element is valid in the real world, and the transaction time, recording the moments when the data entry was inserted, updated, or deleted. In our approach, the central system automatically assigns and stores the transaction time, while the local systems are responsible for maintaining the valid time where appropriate. Queries based on transaction time are processed by special algorithms implemented in the central system; queries related to valid time are processed in the same manner as queries to normal attributes.

The rest of the paper is organized as follows: The second chapter describes the principles of the DataPile technology used to flexibly store structured data in a relational database system. The next chapter focuses on entity identification using data matching and relevance weighing. The fourth chapter shows the overall architecture of the integrated system. The fifth chapter presents an evaluation based on a commercial data-integration project where the DataPile approach was used.

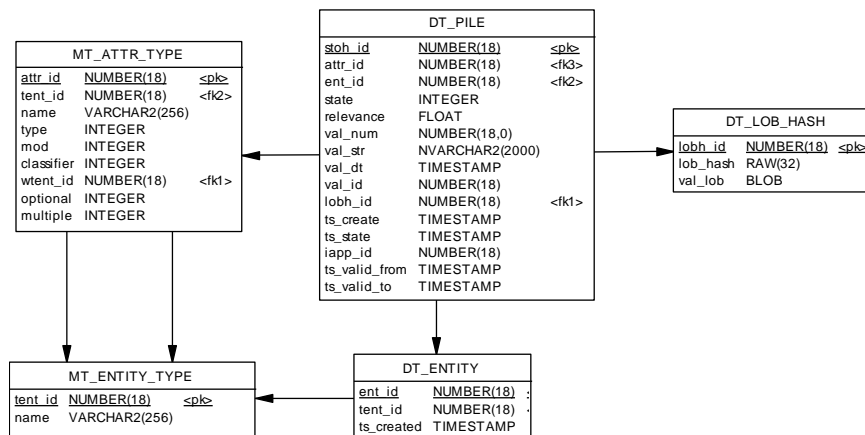
2 The DataPile

2.1 Terminology

We have used an own terminology, which is partly derived from the XML terminology. The first term is *entity*, which represents a type of the traditional database row. An entity consists of *attributes*, which are analogous to the traditional database columns. An *entity instance* is an instance of entity and directly equals to traditional database row contents. An *attribute value* is an instance of attribute and forms a value of one column in one row. A *metatable* is a conventional database table used by the DataPile to store schema information and other system data.

2.2 Data Verticalization

Usual information systems consist of some nontrivial number of conventional database tables; huge information systems have huge number of such tables. Moreover, the requirement for preserving all changes in data usually leads to the scheme, where changing one value of one column in one row causes inserting a new changed row (possibly very large) and updating the old row with some state changing column (e.g. validity termination timestamp). Another problem in conventional information systems is extensibility; adding some new columns or new tables may cause large application code rewriting.



All these problems are addressed by the proposed method of storing data in different way than in traditional approaches but using standard relational databases – the DataPile. All real applications data are stored in two relational tables: one less important table DT_LOB_HASH is dedicated for storing LOBs (for performance purposes), and the second one, the most important, DT_PILE stores data of all other datatypes. This particular table is called the Pile, because all data is stored in one table without any “well-formed” internal structure or hierarchy. Each row in the pile repre-

sents one attribute, whose value is/was valid during certain interval of transaction time.

The picture represents slightly simplified schema of the heart of DataPile-based information system. Tables with prefix DT_ hold real data; all other tables (with prefix MT_) are metatables. The table DT_ENTITY holds valid “global” ID for an entity instance stored in the pile together with information about the entity in form of a reference to the metatable MT_ENTITY_TYPE which stores entities. Entities consist of attributes, and this is modeled by the metatable MT_ATTR_TYPE.

Real values are stored in columns val_xxx of the main table DT_PILE, where xxx represents logical type of the attribute (number, string, datetime, ID – foreign key). Besides the actual data, other additional data is stored in the DT_PILE table: Transaction time aspect of any attribute value is represented by two columns ts_valid_xxx. The type of given attribute value can be found by reference attr_id to the MT_ATTR_TYPE. The ent_id value compounds all attribute values into one entity instance. Other columns not mentioned here serve the system for proper implementation of the functionality needed.

Such a structure easily avoids all the problems mentioned at the beginning of this paper: The number of relational tables used does not grow with an expansion of an information system; it is constant regardless on how huge the system is. Data changes are preserved with minimal overhead – one attribute value change is represented by inserting a new value into the pile – one new row is inserted into DT_PILE table not touching the rest of attribute values related to the same entity instance. Extensibility of the system is reached by the possibility to insert some new rows into metatables and therefore the possibility of defining new entities, attributes, or both.

From the above described layout we can see that this data structure fulfils two requirements put on the information system as a whole: easy extensibility of the data scheme and full information about data changes on the timeline.

3 Data Matching and Weighing

The requirement on data unification is solved by two algorithms: data matching and data weighing.

3.1 Data Matching

Let us show an example, which represents usual situation we meet while processing the same data in different applications. Let application A1 have a record about a person with the name “Jana”, surname “Teskova” with some personal identification number “806010/7000” and an address “Mother’s home No. 10”. The same information is stored in the application A2 as well. After Jana Teskova got married, she took her husband’s surname (as it is quite usual over here). So her surname changes to “Stanclova”. She also moved to live with her new husband on the address “New home 20”. Our person notifies about her marriage and the accompanying changes only the office using application A1, and does not notify other office with application A2 - at first she

might not even know A2 does not share the data with A1 as they both are used to keep data about people in one organization, and at second she may expect that A1 and A2 are integrated together, so changes in A1 are automatically redistributed to A2 as well (but this is a so called distribution problem, which is discussed later). So the result is A1 and A2 store different data about one entity instance. What happens when we try to merge data from A1 and A2 into a common data storage?

As our example shows, nearly all attributes have changed. But some of them are constant, especially personal identification number, which should be truly unique in our country. The association of words “should be” unfortunately means that cases exist, when different persons have the same personal identification number. On the other side, these cases are rare. Having two personal records with the same personal identification number means they belong in fact to a single person with probability of roughly 0,999999.

In this example, other attributes have changed, but a combination of some attributes can have significant meaning: e.g. name and surname together form a whole name. Even name and surname aren't commonly unique in a state, equality of such attributes means some nontrivial probability these two records describe a single person.

This example leads us to attribute classification. Every attribute is assigned one of these classes: determinant, relevant, uninteresting.

- Determinant – identifies an entity instance with very high probability (e.g. personal identification number, passport number etc.).
- Relevant – significant attribute, which helps identify unambiguously equality of entities (e.g. attribute types “name” and “surname” for entity type “person”).
- Uninteresting – has no impact on entity matching.

Following algorithm describes entity matching for two entity instances (one is already stored in the database, the second one is a newly integrated/created entity):

1. All determinant and relevant attribute values are equal – quite clear match with very high probability.
2. A nonempty subset of determinant and nonempty subset of relevant attribute values are equal, remaining determinant and relevant attribute values have no counterpart in the complimentary entity instance – very good match with quite high probability yet (example: let us extend our example with another attribute “passport number”. The first entity instance has attributes “personal identification number” and “passport number” filled. The second entity instance has only “personal identification number” filled and “passport number” is missing.).
3. A nonempty subset of determinant attribute values is equal, remaining determinant attribute values has no counterpart in the complimentary entity instance, but some nonempty subset of relevant attribute values differ – this case seems to be clear as well, because the probability of match for determinant attribute values outweighs probability of different relevant attribute values, but some uncertainty remains as the probability of determinant attribute values is always <1.0 . We must not lose any data and their history, so the system solves such a case by considering these two entity instances as different with notification to the system administrator. The administrator can investigate this case more precisely and can merge these two entities together using an administrator application. This case directly describes the situation during entity matching from our first example – the per-

sonal identification number as a determinant attribute value is the same, but surname as a subset of relevant attribute values differs.

4. A nonempty subset of determinant attribute values differs, remaining determinant attribute values with counterpart are equal, some nonempty subset of relevant attribute values is equal, remaining relevant attribute values have no counterpart – this case usually arises out of misspelling one determinant attribute value. This case is solved as above – entity instances are considered to be different and the system administrator is notified.
5. All other cases – input entities are different entity instances with very high probability.

3.2 Data Weighing

Let us show another example: An employee record is usually kept in different applications in different departments, e.g. human resources department, payroll/accountants department, library, etc. Some applications and departments themselves emphasize some entities and usually some subset of attributes from entities used, e.g. staff department knows with high probability that given person has a certain name, surname, home address, etc., whereas payroll department knows with high probability his/her account number, etc.

It should be beneficial for data integration to have possibility measure somehow the probability, that an application has entity instances (or more precisely on individual attribute values) filled with correct values. Therefore, every attribute value in the DataPile keeps a number which measures probability this given value is correct. This probability is stored in the DT_PILE column “relevance”.

During processing of incoming data all incoming attribute values are somehow evaluated (this will be explained later) and the computed relevance is compared to current relevance of attribute value stored in the DataPile. If the new value has greater or equal computed relevance than current value has, the new value “wins”, becomes the current value, and the old value is marked as archive. Otherwise (when the relevance is lower than current relevance) the new value is stored as well, but only as a remark saying the application has ineffectually tried to change this attribute value.

But there is a problem: when an unimportant application with low relevance keeps correct data (replicated from the central repository) and wants to change some attribute values (because a user has made some changes), these changes will be always ignored. This problem is solved by “approving” the data. Every application must confirm to central repository it agrees with current data replicated from central repository to this application. This confirmation is stored in the DataPile and the system knows the given application has accepted the current attribute value. When such an application (which approved a current attribute value) changes the value, the rule about weighing relevance is ignored and the attribute value is changed.

For example, a large company has usually some branch offices, where department branches can be located as well. Such branches usually show different credibility, which should be reflected by the relevance computation as well.

For computing relevancy of an attribute value following equation is therefore used:

$$R_a = R_{ap} \cdot R_{iap} \cdot R_{et} \cdot R_{ec} \cdot R_{at} \quad (1)$$

where R_a means attribute value relevancy, R_{ap} is static application relevancy (e.g. application used by staff department), R_{iap} is static instance of application relevancy (e.g. branch of department), R_{et} is a static entity relevancy, R_{ec} is a computed entity instance relevancy, and R_{at} is a static attribute relevancy. All these static values are stored in metatables as floating-point values. R_{ec} represents computed relevancy for given entity instance and its value is computed as follows:

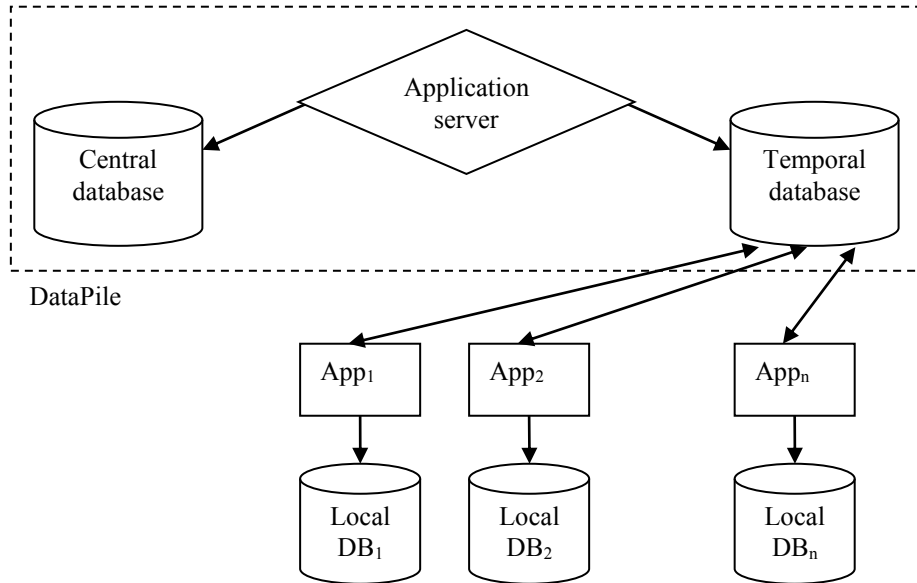
$$R_{ec} = \prod_i R_{ac}(i) \quad (2)$$

where $R_{ac}(i)$ is either a value stored in metatables (when attribute i from given entity has some particular value stored in metatables as well), or it has value of 1. This computed relevancy supports changing of the relevancy based on the presence of selected attributes in the entity.

4 Implementation

4.1 The DataPile Architecture

The whole DataPile-based system is in general shown on the following picture.



Every application (marked in the picture as $App_{1..n}$) has its own local database (Local $DB_{1..n}$). The DataPile machinery is constituted by the Trinity of Central database, Temporal database, and Application server: the Central DB contains the DataPile as data structure for collected data storage, the Temporal DB serves only as communication medium between the DataPile machinery and applications (data is revealed here only during replication and immediately deleted when replication ends). The Application server, which gives life to the whole system, is discussed in following section.

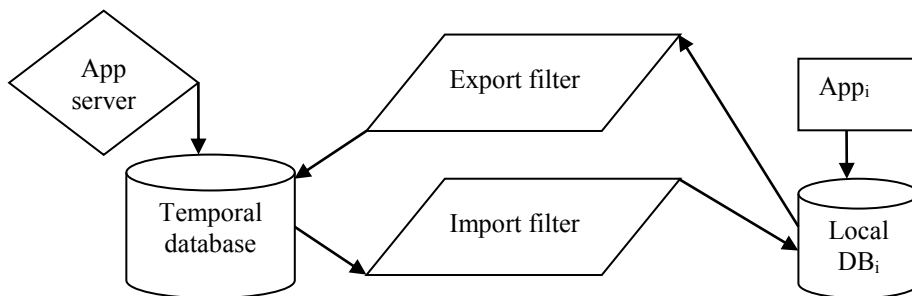
4.2 Application Server

The whole DataPile architecture utilizes the request/reply paradigm. The application server behaves to the rest of world passively; it waits for requests inserted into the temporal DB, fulfills them using central DB data, and writes a reply back into the temporal DB.

Application server is by intention implemented so that it doesn't understand any data semantic. Everything is controlled by the content of metatables and nothing is hard-coded. The application server is primarily responsible for replication, computing algorithms like all the above mentioned data matching and weighing, and can handle other requests too (e.g. perform specialized queries upon the data stored in the DataPile).

4.3 Replication channel

The communication channel between an application and the DataPile machinery is not as simple as it may appear on the first look. In reality there are inserted two filters: export and import filter. They are traditional adapters, which are responsible for adapting different database schemas used by the DataPile machinery and local database. A watchful reader may note that the direction from application to the DataPile is marked as "export", opposite direction as "import"; the marking is taken from the application point of view.



4.4 Cache

New applications may advantageously use collected data in central repository. Unfortunately the DataPile structure is not very well suited for direct access (e.g. searching is not very effective). Almost all applications in fact need to know only current attribute value, ignoring all history stored in the DataPile. To support such applications, the application server actively builds and maintains caches, where only current attribute values are stored and these caches are presented to applications in the form of traditional relational tables. The applications can easily search and use all other RDBMS functions on the caches.

5 Evaluation and Conclusions

The architecture described in this paper brings an alternative to traditional techniques. It brings several advantages but also some disadvantages.

All the concepts described in this paper were used in a real project – design and development of an information system based on data replication and synchronization of data coming from bigger number of different data sources (approx. 30 local information systems and 20 other applications producing 30 millions entries per year for 60 000 users, in this project). The project lasts from the fall 2003 and now is in the phase of finishing the pilot phase and starting roll-out.

Basic and commonly usable advantage of the DataPile structure is its maintainability, easy extensibility and ability to keep the track of the whole data history. All current applications used at all branches remain preserved and functional without any change according to a strongly desired requirement. The central data repository integrates data from all data sources including all their history and sources of their changes. This enables recovering of any historical snapshot of any data. All data changes are redistributed to all other applications that contain these data, even if the source and destination schemas are different. The global schema changes affect neither data in the central repository nor local applications.

During the development of the project, we have discovered several disadvantages of our approach:

Efficiency, especially during export and matching, is low. During the initial export of one certain local system, about 500 000 entries had to be processed. This took more than 24 hours. This time complexity is caused by a relatively complex matching algorithm. Fortunately, this time complexity is not very important in everyday life because number of data changes is smaller in magnitude in comparison to the initial migration data volume.

The second disadvantage is the fact, that the structure of the central repository makes constructing direct queries difficult. Therefore the concept of caches was introduced and all the queries to non-historical data are performed on the caches instead of the DataPile itself.

The project showed that the DataPile approach is suitable for certain class of large applications, where data warehousing is coupled with maintaining consistency of local

databases. In this class of applications, the drawbacks mentioned above are outweighed by integration of data warehousing features with the support for data replication, synchronization, and cleaning using back-propagation.

References

- [1] R. Bruckner, B. List, J. Schiefer, A.M. Tjoa. Modeling Temporal Consistency in Data Warehouses, In 12th International Workshop on Database and Expert Systems Applications (DEXA'01), IEEE Computer Society Press, pp. 901-905, Munich, Germany, September 2001.
- [2] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In Proc. of IPSJ Conference, pages 7--18, 1994.
- [3] I. K. Ibrahim, W. Schwinger. Data Integration in Digital Libraries: Approaches and Challenges, Software Competence Center Hagenberg, Austria, 2001.
- [4] C. S. Jensen and R. T. Snodgrass. Temporal Data Management. IEEE TKDE, 11(1): 36--45 (1999).
- [5] M. Lenzerini. Data integration: A theoretical perspective. In Proc. of the 21st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2002), pages 233--246, 2002.
- [6] E-P. Lim, J. Srivastava, S. Prabhakar & J. Richardson. Entity identification in database integration, in Proceedings Ninth International Conference on Data Engineering, Vienna, Austria, April 19--23, 1993, IEEE Computer Society Press, Washington, DC, 1993, 294—301.
- [7] M. Mecella, M. Scannapieco, A. Virgillito, R. Baldoni, T. Catarci, and C. Batini. Managing Data Quality in Cooperative Information Systems, Proceedings of the 10th International Conference on Cooperative Information Systems, Irvine, CA, 2002.
- [8] A. Mostéfaoui, M. Raynal, M. Roy, D. Agrawal, A. el Abbadi. The Lord Of The Rings: Efficient Maintenance Of Views At Dataware Houses. Publication interne No. 1441, IRISA, Rennes, France, 2002.
- [9] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In Proceedings of International Conference on Extending Database Technology (EDBT'98), pages 359--373, Valencia, Spain, March 1998.
- [10] E. A. Rundensteiner, A. Koeller, and X. Zhang. Maintaining Data Warehouses Over Changing Information Sources, Communications of the ACM, Vol. 43, No.6, June 2000.
- [11] E. Schallehn, K. Sattler, and G. Saake. Extensible and similarity-based grouping for data integration. In 8th Int. Conf. on Data Engineering (ICDE), San Jose, CA, 2002.
- [12] K. Torp, C. S. Jensen, and R. T. Snodgrass. Stratum Approaches to Temporal DBMS Implementation. In Proceedings of IDEAS, Cardiff, Wales, pp. 4--13 (1998).
- [13] J. Widom. Research Problems in Data Warehousing. In Proceedings of the 4th Int'l Conference on Information and Knowledge Management (CIKM), November 1995.
- [14] T. W. Yan and H. Garcia-Molina. Duplicate removal in information dissemination. In Proceedings of VLDB-95, September 1995. Information Systems, Irvine, CA, 2002.

Chapter 3.

Using Input Buffers for Streaming XSLT Processing

Jana Dvořáková, Filip Zavoral

DBKDA / GlobeNet 2009, International Conference on Advances in Databases, Knowledge, and Data Applications, GlobeNet 2009, IEEE Computer Society Press, 2009

The First International Conference
on Advances in Databases,
Knowledge, and Data Applications

DBKDA 2009

1-6 March 2009

Gosier, Guadeloupe/France

Editors

Qiming Chen

Alfredo Cuzzocrea

Takahiro Hara

Ela Hunt

Manuela Popescu

Using Input Buffers for Streaming XSLT Processing

Jana Dvořáková, Filip Zavoral
Charles University in Prague, Czech Republic
{dvorakova, zavoral}@ksi.mff.cuni.cz

ABSTRACT

We present a buffering streaming engine for processing top-down XSLT transformations. It consists of an analyzer and a transformer. The analyzer examines given top-down XSLT and XSD, and generates fragments which identify parts of XSD need to be buffered when XSLT is applied. The fragments are passed to the transformer which processes XSLT on an input XML document conforming to XSD. It uses auxiliary memory buffers to store temporary data and buffering is controlled according to the fragments. We describe implementation of the engine within the Xord framework and provide evaluation tests which show that the new engine is much more memory-efficient comparing to the common XSLT processors.

1. INTRODUCTION

XSLT is typically processed by tree-based processors which store the whole input document in the memory and then apply the transformation. XML has started to be used extensively in domains where such traditional processing is not suitable, e.g.:

- data streams needed to be processed "on the fly",
- data processed in portable devices with limited memory,
- huge database exports exceeding available memory.

In this paper we focus on automatic streaming processing of XSLT transformations since currently there does not exist an appropriate alternative to the traditional automatic XSLT tree-based processors.

During the previous work on the Xord project [5, 6], the Xord framework for the streaming processing of XSLT transformations was designed and implemented. The framework is intended to contain several streaming engines for processing XSLT. Each engine consists of an analyzer and a transformer. The analyzer analyzes given XSLT transformation and it determines whether it can be processed by given engine. It may pass some information collected during the analysis to the transformer which performs the transformation itself. The transformers are based on formal models called streaming XML transducers. This formal base enables us, for each transformation algorithm, to explicitly determine the class of XSLT transformations captured and the memory consumed. Within the framework, the SSXT¹ engine was implemented. The SSXT transformer [5] processes a subset of top-down XSLT using stack of the size proportional to the depth of the input XML document. The SSXT analyzer [6] takes a schema and a top-down XSLT stylesheet, and it determines whether transformation can be processed by the

¹SSXT stands for simple streaming XML transducer.

SSXT algorithm on the XML documents defined by schema.

Although the SSXT transformation algorithm is highly memory efficient, the class of possible transformations is markedly restricted. The most important restriction is the order-preserving condition - the ordering of the output nodes must follow the order of the input document. In this paper, we present a BUXT² engine that overcome these limitations. Some parts of the input document can be stored in buffers for future processing so that the output can be potentially in any order according to the input document.

The main contributions are the following:

- We design and implement the BUXT transformer which is able to process all top-down XSLT transformations. The base of the algorithm is the SSXT transformer. The BUXT transformer is extended with buffers for temporary storage and it is able to process more complex transformations.
- We design and implement the BUXT analyzer which is an extension of the SSXT analyzer. The BUXT analyzer statically computes the information about moments when buffering is needed based on an analysis of given schema³ and XSLT stylesheet. The information is provided in the form of *schema fragments* (shortly *fragments*) and passed to the BUXT transformer. Moreover, by examining fragments, it is possible to compute maximal amount of memory needed for processing the stylesheet on XML documents defined by given schema. See Fig. 1 for overall schema of the BUXT engine.
- We provide evaluation tests of the space complexity of the BUXT transformation algorithm and a comparison to commonly available XSLT processors.

Related work. Existing automatic XQuery streaming processors (BEA/XQRL [7], FluXQuery [9], XSM [10]) are typically designed for specific purpose. Moreover, they appear as black boxes - the streamability is achieved by ad-hoc optimizations and the amount of memory used for certain types of transformations is not known. XSLT automatic streaming processor SPM [8] uses known amount of memory, but it can process only very simple transformations and the class of transformations captured is not clearly characterized. Low-level streaming languages (STX [1], StAX [2]) based on event-based programming represent another alternative for handling transformations in the streaming manner. This approach, however, requires the user to write the transformation explicitly. Other research direction deals with streaming queries [11], but this is only one of the subproblems of the whole transformation process.

²BUXT stands for buffering XML transducer.

³We use the terms *XSD* and *schema* interchangeably.

2. XSLT AND SCHEMA REPRESENTATION

We briefly describe subsets of XSLT and XSD considered in this work as well as the way how both structures are modeled in the BUXT engine.

2.1 XSLT representation

We consider a top-down fragment of XSLT language. It allows matching XSLT templates with modes and top-down XPath axes. A transforming template is called by an element name and a mode:

```
<xsl:template match="a" mode="m1">
  ... body ...
</xsl:template>
```

The template body consists of output elements (possibly nested) and template calls which call application of other templates by an XPath expression and a mode. The template calls are of the form:

```
<xsl:apply-templates
  select="child::a/descendant::b" mode="m2"/>
```

A subset of XPath expression is allowed in transforming templates - they may contain child and descendant axis, and they select nodes by name:

```
XPath := Step | Step/XPath
Step  := (child | desc)::name
```

Template model. A template in BUXT engine is a structure *tmp* that consists of the following components:

- *tmp.match-name* - the name of the matching element,
- *tmp.mode* - the matching mode,
- *tmp.calls* - a sequence of template calls, a single call *call* consists of two components: *call.expression* and *call.mode*,
- *tmp.output-parts* - a sequence of output parts, the *i*-th output part is a sequence of tags to be generated between calls *i* - 1 and *i* (see the example below),
- *tmp.fragments* - a set of fragments.

Example. Let us consider the following XSLT template with two template calls:

```
<xsl:template match="a" mode="m0">
  <output-a1>          <!-- output part 1 -->
    <xsl:apply-templates select="child::b" mode="m1"/>
    <output-a2>          <!-- output part 2 -->
    <xsl:apply-templates select="desc::c" mode="m2"/>
  </output-a2>
</output-a1>          <!-- output part 3 -->
</xsl:template>
```

We obtain a BUXT template *tmp* of the form:

```
tmp.match-name = a
tmp.mode       = m0
tmp.calls      = (child::b, m1), (desc::c, m2)
tmp.output-parts = <output-a1>, <output-a2>,
                  </output-a2><output-a1>
tmp.fragments  = ∅
```

The set of fragments is initially empty, it is filled up first during the analysis (see Section 4).

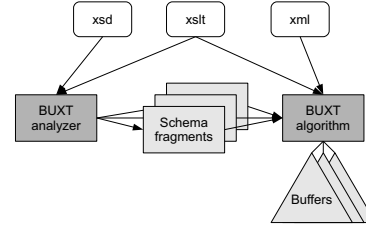


Figure 1: A schema of the BUXT algorithm

2.2 Schema representation

We consider schemas without *choice* constructor and recursive definitions. We represent such schema hierarchically as a *schema tree*. It consists of two kinds of nodes:

- *element nodes*: correspond to element types defined within schema,
- *constructor nodes*: correspond to constructors used in the schema (sequence, choice, *, +, ?).

The relationships among element types and constructors are represented by the structure of the tree. An example of schema tree is depicted on the left-hand side in Fig. 2.

Schema model. A schema node in the BUXT engine is a structure that consists of the following components⁴:

- *node.label* - an element name or a constructor symbol,
- *node.type* - either element node or constructor node,
- *node.children* - a sequence of references to child nodes.

Some subtrees of the schema tree may be identical - this situation occurs if we derive the schema tree from XSD containing shared element types. However, during the analysis, the order of the particular schema nodes is important. Therefore, such DAG structure is transformed to a tree during the analysis by duplicating shared nodes.

3. BUXT ENGINE OVERVIEW

The buffering streaming engine consists of two components: a BUXT analyzer and a BUXT transformer (see Fig. 1).

Analyzer. The analyzer takes two inputs: an XSLT stylesheet *xsl*, and a schema *xsd*. It accomplishes a static analysis of both inputs. As a result, it generates a set of *fragments*. Fragments basically store information on which parts of an XML document defined by *xsd* need to be buffered when the transformation *xsl* is processed on this document in the streaming manner. The fragments are passed to the transformer.

Transformer. The transformer takes two inputs: an XSLT stylesheet *xsl*, and an XML document *xml* valid with respect to *xsd*. It is based on the non-buffering SSXT streaming algorithm [5] which is extended by a possibility to store parts of the input temporarily in memory buffers. The decisions on when to start buffering and when to process buffer content are taken according to the information stored in the fragments.

⁴Although we consider XSD format of schema, note that it is the BUXT schema model can be applied easily to another common format DTD as well.

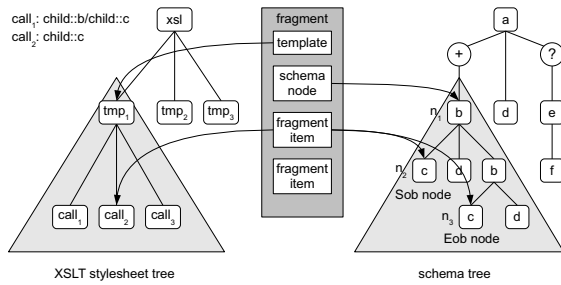


Figure 2: An example fragment

Fragments. The fragments represent the most key structure of the BUXT engine. A single fragment consists of the following components:

- *frag.tmp* - a reference to a template of *xsl*,
- *frag.node* - a reference to a node of *xsd* (*schema context node*),
- *frag.items* a set of *fragment-items*:
 - *item.call* - a reference to a call of template,
 - *item.sob-node* - a reference to a node of *xsd* (*start-of-buffer node*),
 - *item.eob-node* - a reference to a node of *xsd* (*end-of-buffer node*).

A fragment identifies a subtree in the schema tree parts of which require buffering when processed by the referenced template. A fragment-item identifies one of these parts, which is a subtree as well, and a specific call within the template which invokes buffering of this subtree.

A simple fragment is shown in Fig. 2. On the left-hand side, the tree of input XSLT stylesheet *xsl* is depicted and on the right-hand side, a schema tree of input schema *xsd* is depicted. The fragment identifies a subtree in the schema at node n_1 and associates it with tmp_1 of *xsl*. The fragment-item depicted identifies the subtree at n_2 (which is a part of subtree at n_1) and associates it with the call $call_1$ of tmp_1 . The semantics of such fragment-item is as follows: If, during the transformation processing,

- the currently processed template is tmp_1 ,
- the current context tag corresponds to the schema node n_1 ,
- a match has been found for $call_2$,
- the current tag corresponds to the schema node n_2 ,

then the subtree at the current tag is stored in a new buffer *buf*. During consequent processing, when the first two conditions above holds and moreover

- the current tag is end-tag,
- the current tag corresponds to the schema node n_3 ,

then the content of *buf* is processed. See Section 5 for more detailed description of the buffer manipulation.

4. ANALYZER

The analysis is driven by the structure of the schema tree, starting at the root node and continuing downwards to the leaves. The analyzer searches template of *xsl* which matches the current schema node at the current mode and applies *AnalyzeNode* recursively, see Algorithm 1.

Algorithm 1 *AnalyzeNode(tmp, schemaNode)*

```

1: if tmp.calls is empty then {end of analysis in current subtree}
2:   ; {do nothing}
3: else if schemaNode is leaf then {end of analysis in current branch}
4:   ; {do nothing}
5: else
6:   frag = CreateFrag(tmp, schemaNode);
7:   if frag.items is not empty then
8:     add frag to tmp.fragments;
9:   end if
10:  for each call in tmp.calls do
11:    for each node in EvalCall(call, schemaNode, ) do
12:      let called-tmp be template called by call;
13:      AnalyzeNode(called-tmp, node)
14:    end for
15:  end for
16: end if

```

In case the current template does not contain any call (1), or the current schema node is leaf (3), the analysis terminates. Otherwise, the function *CreateFrag* for creating fragment is called (6), see Algorithm 2. It finds a fragment for the current template *tmp* which refers to the current schema node *schemaNode*, and adds it to the set *tmp.fragments* in case the item set is not empty. After that, the analyzing function is called recursively (13) for all pairs (*called-tmp, node*) such that

- *called-tmp* is a template called by a call (*call.expression, call.mode*) of *tmp*,
- *node* is a schema node selected by *call.expression* in the schema tree if the evaluation starts at *schemaNode*.

The evaluation of *call.expression* is accomplished by the function *EvalCall(call, schemaNode)* (11). It corresponds to the evaluation against the XML tree which is formed from the schema tree by omitting all schema constructors.

Algorithm 2 *CreateFrag(tmp, schemaNode) : fragment*

```

1: create new fragment frag such that
2: frag.node = schemaNode, frag.tmp = tmp, frag.items = ∅;
3: for each calli in tmp.calls do
4:   set matchedNodesi = EvalCall(calli, schemaNode);
5: end for
6: for each callk do
7:   for each node node in matchedNodesk do
8:     set eobCandidates = {candNode ∈ matchedNodesi |
9:                         i < k, candNode >preorder node}
10:    if eobCandidates is not empty then
11:      set eobNode = maxpreorder(eobCandidates);
12:      add fragment-item (callk, node, eobNode) to frag;
13:    end if
14:  end for
15: end for

```

The function *CreateFrag* first creates a fragment with empty set of items referencing to the current schema node and current template (1). Then particular fragment items are generated stepwise for each call of *tmp*. We index the calls by an integer which corresponds to the order of the call in the sequence *tmp.calls*. First,

the nodes matching $call_i$ are found for each i and stored in the sequence $matchedNodes_i$. The order of the matched nodes conforms to the preorder with respect to the schema tree. Then the calls and their matching nodes are processed one by one (6-15). Let $call_k$ be the currently processed call and $node$ be the currently processed node from within the sequence. In next steps, the algorithm determines whether a fragment item exists such that

- $item.sob-node = node$,
- $item.call = call_k$.

The item exists if and only if some end-of-buffer node is found. First, all candidates for end-of-buffer node are collected. Each such candidate, let denote it $cand-node$, must conform to the following two conditions:

- $cand-node \in matchedNodes_i, i < k$,
- $cand-node >_{preorder} node$

It means, $cand-node$ must appear *after* the currently processed node $node$ and at the same time it must be a matching node of some call which appear *before* the currently processed call $call_k$. This is exactly the situation when buffering is inevitable. The maximum candidate node (with respect to the preorder) is chosen as the end-of-buffer node (11) since it represents the position within the schema tree where all calls appearing before the current call have been definitely processed. In case no candidate has been found, the buffering is not needed and the fragment item is not generated.

Note that the overall fragment is added to the set $tmp.fragments$ in the *AnalyzeNode* function if and only if the set of generated items is not empty.

Example. Let us consider the fragment shown in Fig. 2. The referenced template tmp contains the following two calls:

```
call1:  child::b/child::c (matched at  $n_3$ )
call2:  child::c (matched at  $n_2$ )
```

According to the order of calls in $tmp.calls$ it holds

$$call_1 <_{call-order} call_2,$$

but according to the preorder of the schema tree it holds

$$n_3 >_{preorder} n_2.$$

We thus obtain that $call_1$ must wait with generating output until $call_2$ is processed. The node n_2 is set as the start-of-buffer node since the subtree at node n_2 needs to be stored in the buffer buf for further processing. The node n_3 is the end-of-buffer node since at this node it is sure that $call_2$ has generated all of its output. This implies that buf can be processed by $call_1$ and the corresponding output can be generated.

5. TRANSFORMER

The algorithm of the BUXT transformer extends the original stack-based SSXT algorithm by memory buffers for temporary storage. It is based on the model called buffering XML transducer.

Buffering XML transducer. The transducer represents a combination of two models - the simple streaming XML transducer (SSXT) and the general XML transducer (GXT), both introduced in [4]. Following the SSXT behavior, it reads the input XML and generates the output XML in the streaming manner. It is equipped with a stack in order to enable stepwise evaluation of XPath expressions on the input stream SSXT. In addition, BUXT stores part of the input stream in buffers. These buffers are later processed in

the traditional tree-based manner - the buffer content is stored in the memory as a tree and processed from the root to the leaves. The tree-based processing is accomplished by a GXT which is a straightforward formalization of the standard XSLT processors. The significant measures of space complexity are as follows:

- number of buffers,
- maximal size of the buffers utilized during the transformation.

Note that, for a given input document and XSLT transformation, the values of both measures can be computed statically. Such algorithm is however outside the range of this paper.

5.1 Structures and actions

We describe the structures and the actions used in the transformer algorithm.

Stack. Similarly to the original SSXT algorithm, the BUXT transformer uses a stack of the size proportional to the depth of the input document to remember information about particular element levels of the input XML document which is necessary to accomplish evaluation of XPath expressions. Two kinds of data are stored in the stack:

- *DFA* - a sequence of current DFA states,
- *CC* - cycle configuration.

A set of DFAs is used to evaluate XPath expressions in the current template concurrently - a single DFA is associated with a single expression. Such technique has been used for example in the Y-filter algorithm [3].

The cycle configuration contains information about the currently processed part of the XSLT stylesheet xsl and the current position in schema of the input document. The position in the schema is determined according to the current position in the input XML stream, and it is updated at each *advance* action (see action description below). A configuration cc consists of the following components:

- $cc.tmp$ - a reference to a template of xsl (*current template*),
- $cc.call$ - a reference to the lastly matched call of the current template,
- $cc.context-node$ - a reference to a schema node (*context schema node*).

During a single cycle, one template call in a template of xsl is processed. A special initial cycle handles initialization of the transformation. A CC is pushed on the stack when a match is found for some expressions (i.e., a final DFA state appears in the current sequence of DFA states). Here, new cycle for processing the called template starts. A CC is popped after the called template has been processed and the control moves back to the previous template. More detailed description of SSXT stack manipulation can be found in [4].

The context schema node is a new component added first in the BUXT algorithm. It represents a position in the schema tree which corresponds to the current context tag of the transformation, i.e., the tag at which the current evaluation has started. Moreover, the transformation itself keeps a reference to the current schema node which corresponds to the currently processed tag. The pair (*context schema node, current schema node*) is called the *current context*. The current context is necessary in order to make proper decision on when to start buffering and when to process buffer contents.

For simplicity of the presentation, when describing the algorithm itself, we do not explicitly mention keeping references to the cur-

rent context. These references are supposed to be updated in a straightforward way at each *advance* action.

Buffers. The buffers are in-memory tree XML structures which are used to store temporary data for later processing. They are processed in the tree-based manner, mimicking behavior of the standard XSLT processors.

Actions. Based on the SSXT algorithm, three actions are available for stack manipulation, one action for manipulating the input XML stream and one action for generating the output XML stream:

- *push DFA, push CC, pop*,
- *advance* (advances to the next input tag),
- *generate* ($call_1, call_2$) (generates all output XML tags between $call_1$ and $call_2$ ⁵).

The BUXT algorithm uses, in addition, two more actions for manipulating buffer content:

- *fill buffer* - stores the content of the current element in a buffer,
- *process buffer* - process a buffer in the tree-based manner.

The decision about the buffer actions is based on the information stored in the fragments. If the current context corresponds to the fragment context and current node (tag) corresponds to a start-of-buffer node of one of its fragment-items, then a new buffer is filled by the current subtree. Similarly, the decision about processing a buffer is made, but the end-of-buffer node is checked instead of the start-of-buffer node.

5.2 Algorithm

The transformer algorithm uses the following variables (common for all functions mentioned below):

- *cc* - the current cycle configuration,
- *current-tag* - the currently processed XML tag,
- *la-tag* - the lookahead tag.

After an initialization, the transformer calls a proper function depending on the symbol on the top of the stack (see Algorithm 3).

Algorithm 3 *Transform(xsl, xml)*

```

1: set current-tag to first tag of xml;
2: set cc.tmp = template matching current-tag.name in mode  $m_0$ ;
3: set cc.call = cc.tmp.start;
4: push initial DFA states for cc.tmp;
5: while stack is not empty do
6:   if stack.top = sequence  $S$  of DFA states then
7:     ProcessDFA( $S$ );
8:   else if stack.top = cycle configuration stack-cc then
9:     ProcessCycleConfiguration(stack-cc);
10:  end if
11:  generate fragment (cc.call, cc.tmp.end)
12: end while

```

A sequence of DFA states is processed by the *ProcessDFA* function (see Algorithm 4). When a start-tag is encountered, all DFAs perform a transition according to the tag name and a new sequence of states denoted by $S.transition(name)$ is determined (2). In case a final state appears in the new sequence, the transformer checks whether buffering is needed by examining all fragment items (6). If some of them contains start-of-buffer node for

⁵Note that $call_1$ may be the beginning of the template and $call_2$ may be the end of the template.

the current context, a new buffer is filled with the content of the current tag. Otherwise, the content is processed in the streaming manner and a new cycle starts (9).

When an end-tag is encountered, the transformer first checks whether some of the buffers might be ready for processing. It again examines fragment items and selects those which contains end-of-buffer node for the current context (17). For each such item, all associated buffers are processed in the tree-based manner. Then the streaming processing continues.

Algorithm 4 *ProcessDFA(S)*

```

1: if current-tag is start-tag then {Downwards evaluation}
2:   let  $S' = S.transition(current-tag.name)$ ;
3:   if  $S'$  contains no final state then {No match}
4:     push  $S'$ , advance;
5:   else if  $S'$  contains final state for call new-call then {Match found}
6:     if fragment item item exists which contains start-of-buffer node
       for current context then {Buffer filling}
7:       create new buffer buf and fill it with contents of current-tag;
8:       add buf to item.bufbers;
9:     else
10:      generate(cc.call, new-call);
11:      push cc;
12:      set cc.tmp = new-call.tmp;
13:      set cc.call = cc.tmp.start;
14:    end if
15:  end if
16: else if current-tag is end-tag then {Upwards evaluation}
17:   if fragment item item exists which contains end-of-buffer node for
       current context then {Buffer processing}
18:     process all buffer in item.bufbers;
19:   end if
20:   if la-tag is end tag then
21:     pop; advance;
22:   end if
23: end if

```

A cycle configuration is processed by the *ProcessCycleConfiguration* function (see Algorithm 5). In case a new cycle starts (1), a sequence of initial DFA states for the current template is pushed. In case a cycle ends (6), last output part of the current template is generated and the previous configuration is reset.

Algorithm 5 *ProcessCycleConfiguration(stack-cc)*

```

1: if current-tag is start-tag then {Cycle start}
2:   if la-tag is start-tag then
3:     push initial DFA states for cc.tmp;
4:   end if
5:   advance;
6: else if current-tag is end-tag then {Cycle end}
7:   generate(cc.call, cc.tmp.end);
8:   set cc = stack-cc;
9:   pop;
10: end if

```

6. IMPLEMENTATION AND EVALUATION

The BUXT engine was implemented and tested in the Xord framework. The implementation is based on the SSXT engine by extending both its analyzer and transformer part according to the formal algorithms described above. Besides extending the algorithms themselves, there are two structural extensions:

- the set of fragments and their fragment items as the output of the analyzer phase and the input of the transformation phase
- keeping references to the relevant schema nodes while reading symbols from the input document.

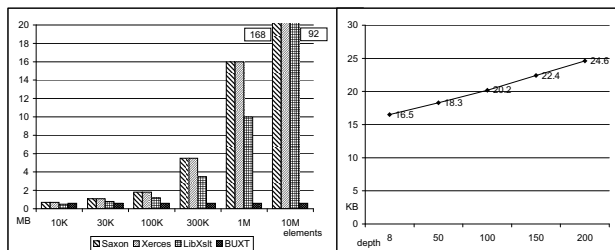


Figure 3: Memory complexity evaluation

We have compared the BUXT algorithm space complexity against the publicly available tree-based XSLT processors (Saxon, Xalan and XsltProc) using both synthetic and real data. Fig. 3a. shows a comparison of transformation memory requirements of 10000 to 1 million entities. All the tree-based processors consumed large amounts of memory when processing large XML data (above 100K of entities) regardless the simplicity of the transformation.

The evaluation confirmed that the BUXT algorithm basically requires a memory proportional to the depth of the input XML. Fig. 3b. shows a net memory consumption of the algorithm (without libraries, runtime environment etc.) processing the input data of different depth. Since the document depth is generally not depending on the document size and documents are relatively shallow [12], the memory requirements for most of the XML documents is low, independent to the document size. Even for large documents like DBLP (700 MB), the BUXT algorithm required below 100 KB of net memory while the above mentioned DOM-based processors crashed or hanged after allocating about 1.5 GB of memory.

Additionally, there is an extra memory required for each fragment item detected during the transformation. The size of such memory does not depend on the whole input size but on the schema and the XSLT structure. As long as the ordering of the output document remains close to the input document (the transformation is mostly local), the space complexity remains low. The most typical example of such processing is filtering, mapping and local reordering of a huge sequence of relatively small subtrees, such as logs, structured data streams or XML databases.

On the other side, the BUXT transformer is not very suitable for some classes of transformations. The example of inappropriate transformations is swapping two large subtrees or moving a little subtree from the end of the input document to the beginning of the output. For such transformations all of the input that should be processed later must be stored into the buffers and the space complexity may achieve the tree-based processors in the worst case.

7. CONCLUSION

We introduced an enhancement of the Xord framework for efficient XSLT processing. The functionality of the framework is currently based on the stack-based streaming algorithm which is able process a class of top-down XSLT transformations using stack of the size proportional to the depth of the input document. Additionally, some parts of the input document can be stored in buffers for later processing. The analyzer can detect the context when such buffering starts and when the content of such stored buffers should be processed instead the regular input. The results of our experiments show that the engine is much less memory-consuming when processing huge data sets or data streams comparing to the common tree-based processors for a wide class of transformations.

Several issues are left for the future work. First, we intend to

overcome some restrictions to XSLT and schema constructs that can be processed by the Xord engine such as conditions and choices. The algorithms can be also improved by an appropriate schema inference strategies [13]. Next, we plan to design multipass algorithms that could be much more memory efficient for some classes of transformations at the cost of processing the input in several passes.

Acknowledgments.

This work was supported by the Grant Agency of the Czech Republic, grant number 201/09/0990 - XML Data Processing and by the Slovak Grant Agency, grant VEGA 1/3106/06. A part of the results presented comes from a PhD thesis of Comenius University in Bratislava, Slovakia.

8. REFERENCES

- [1] O. Becker. Transforming XML on the Fly. In *Proceedings of XML Europe 2003*, 2003.
- [2] The Codehaus. *StAX*, 2006. <http://stax.codehaus.org/>.
- [3] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.
- [4] J. Dvořáková. Towards Analyzing Space Complexity of Streaming XML Transformations. In *The Second IEEE International Conference on Research Challenges in Information Science*. IEEE Computer Society, 2008.
- [5] J. Dvořáková and F. Zavoral. An Implementation Framework for Efficient XSLT Processing. In *Proceedings of IDC 2008*, Studies in Computational Intelligence. Springer-Verlag, 2008.
- [6] J. Dvořáková and F. Zavoral. Schema-Based Analysis of XSLT Streamability. In *Proceedings of ADVCOMP 2008*, Studies in Computational Intelligence. IEEE Computer Society, 2008.
- [7] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Ricciardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In *Proceedings of VLDB 2003*, pages 997–1008, 2003.
- [8] Z. Guo, M. Li, X. Wang, and A. Zhou. Scalable XSLT Evaluation. In *Advanced Web Technologies and Applications, LNCS 3007/2004*. Springer Berlin / Heidelberg, 2004.
- [9] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An optimizing XQuery processor for streaming XML data. In *VLDB'2004: Proceedings of the Thirtieth International Conference on Very Large Databases*, pages 1309–1312, 2004.
- [10] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB 2002*, pages 227–238, 2002.
- [11] N. S. Martin Grohe, Christoph Koch. Tight lower bounds for query processing on streaming and external memory data. In *Theor. Comput. Sci.* 380(1-2): 199-217. Springer Berlin / Heidelberg, 2007.
- [12] K. Toman and I. Mlynkova. Statistics on the Real XML Data. In *XML Prague'06*, pages 87–102, Prague, Czech Republic, 2006. Ginger Alliance.
- [13] O. Vosta, I. Mlynkova, and J. Pokorny. Even an Ant Can Create an XSD. In *DASFAA'08, LNCS*, pages 35–50. Springer, 2008.

Chapter 4.

High-Level Web Data Abstraction Using Language Integrated Query

Jakub Míšek, Filip Zavoral

Intelligent Distributed Computing IV, Springer Verlag, pp. 13-22, 2010

Studies in Computational Intelligence 315

Mohammad Essaïdi
Michele Malgeri
Costin Badica (Eds.)

Intelligent Distributed Computing IV

Proceedings of the 4th International Symposium
on Intelligent Distributed Computing - IDC 2010,
Tangier, Morocco, September 2010

 Springer

High-Level Web Data Abstraction Using Language Integrated Query

Jakub Misek and Filip Zavoral

Abstract. Web pages containing huge amount of information are designed for human readers; it makes their automatic computer processing difficult. Moreover web pages live their content is changing. Once a page is downloaded and processed, few seconds after that its content can be different. Many scraping frameworks and extraction mechanisms have been proposed and implemented; their common task is to download and extract required data. Nevertheless, the complexity of development of such application is enormous since the nature of data does not conform to common programming paradigms. Moreover, the changing content of the web pages often implies repetitive extracting of the whole data set.

This paper describes the LinqToWeb framework for web data extraction. It is designed in an innovative way that allows defining strongly typed object model transparently reflecting data on the living web. This mechanism provides access to raw web data in a completely object oriented way using modern techniques of Language Integrated Query (LINQ). Using this framework development of web-based applications such as data semantization tools is more efficient, type-safe, and the resulting product is easily maintainable and extendable.

1 Introduction

Since Internet lives in its own limitless world where everybody is allowed to join and contribute, all information placed on the web is growing every second. It is full of various data, like daily newspapers, discussion forums, shop catalogs, images or videos, which are accessible by almost anyone. The automatic data extraction represents huge nowadays problem. The web pages are intended for human readers. Moreover single information is spread on more pages. Although there exist recommended formats for web developers how to export their information to be

Jakub Misek · Filip Zavoral
Charles University in Prague, Czech Republic
e-mail: {misk, zavoral}@ksi.mff.cuni.cz

readable by machines (RSS, web services or more advanced and self-describing RDF or OWL), vast majority of the web data does not contain such semantic or structural description. The main goal of extraction tasks is to retrieve desired information into some structuralized form. That is the most common purpose of scraping frameworks and applications. They are able to download e.g. all needed web pages, and export specific values into a local storage like database or XML. Two main categories of scraping software approaches can be identified:

- **Standalone application.** Specific extraction mechanisms are implemented as a standalone large scale application [12, 13]. Such applications provide methods for extracting data from various sources and several options how to save the results. Everything used to be configurable in a declarative way, usually using a graphic user interface or scripting. Occasionally the application is able to perform updates automatically. These solutions have high requirements for storage capacity and the whole extraction process is time consuming.
- **Framework.** Programmers can take benefits from extraction frameworks [1, 8]. Using such libraries programmers have to take care about all the processes; the implying advantage is a possibility to develop more customized and optimized extraction. For example such application can modify extraction parameters during runtime or it can download only specific parts of web sites.

Most of these solutions assume that the user wants to extract information into a local storage. Such behavior is sometimes not desirable, because most of the downloaded information is not subsequently used. Downloading everything locally causes high requirements for storage capacity and difficult updates of already extracted information. Moreover many implementations solve the updates simply by redownloading everything from the beginning. The other issue when using such frameworks is efficiency of development data are usually represented in an unstructured form of strings, binding to language data types is weak, no syntax checking is possible in compile time etc.

The main contribution of this paper is a design of a high-level integration of web information extraction into a software development process. The framework LinqToWeb [11] benefits from advantages of previously described approaches. The extraction tasks can be defined in a declarative way, while the programming interface uses type safe object model representing the abstraction of the web resources. The tasks are compiled; it maximizes performance of the extraction.

The programmer can use web resources in the same way as data in a local memory. In contrast to contemporary solutions local storage is not used explicitly, but only as a transparent caching mechanism. Moreover, particular data items are accessed only when requested that makes the extraction process much efficient the data access is not delayed by long extraction queues. The high-level object oriented approach takes benefits of modern language features like Language Integrated Query (LINQ) [4], code sense capability provided by development environments etc. automatically.

2 Architecture

The main idea of the LinqToWeb design is working with web resources in the same way as with the statically declared objects instantiated during runtime. The object oriented approach takes advantage of safe type checking, efficient compiled task processing and auto-completion support provided by development environments - object members are automatically offered during code typing; the programmer can see the data structure immediately. Also the source code is automatically checked for typing errors; it makes the development easier and more type-safe.

The architecture is inspired by LinqToSQL [7] integration which was introduced as a part of Microsoft Visual Studio 2008. The main principle is based on generating strongly typed object model from something that is not so easy to use and causes programming issues in general. The description of data sources is used for generating type-safe objects that encapsulate all possible usages of data. The complete process of generating objects is automatically performed by the development environment - every time the source data description is changed, the programmer works with up-to-date objects implementation.

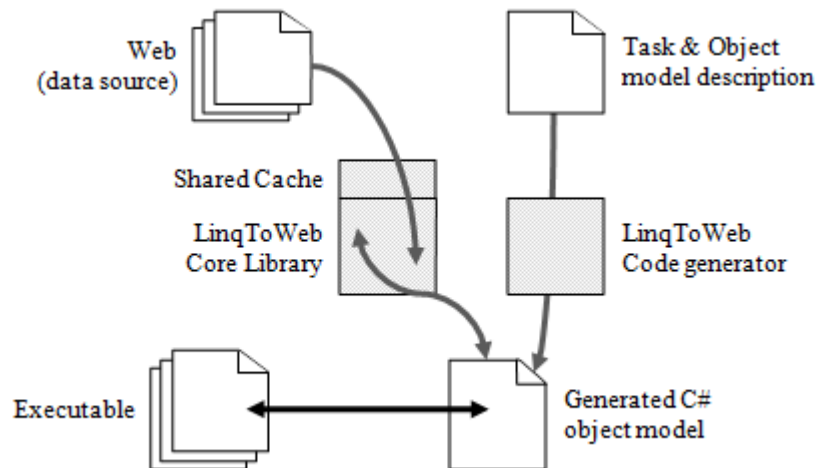


Fig. 1 LinqToWeb architecture. The programmer accesses only the generated object model, the rest is hidden under the generated abstraction.

The typical usage of LinqToWeb is depicted on Fig. 1. Assume we need to use information from specific web pages or another not necessarily web based sources. The programmer provides description of the information its abstract object model and how to fill this object with data (see Sect. 3). The LinqToWeb code generator transparently generates C# code containing objects and extraction tasks using functionalities hidden in LinqToWeb core libraries.

Generated code file becomes a part of the program sources; the compiler is able to perform type checks and optimizations. To achieve an effect of objects with all values locally initialized, the whole mechanism makes use of the .NET framework platform and its mechanism of properties. Moreover, generated objects can be integrated to all languages supporting the .NET platform. During the extraction the system uses associative cache. Its purpose is to speed up repetitious requests for web resources or repetitious extraction tasks. This cache manages expiration time of resources; expired cached resource is simply downloaded again as part of evaluation of next relevant request. The storage of the cache can be implemented in various ways, e.g. in memory only or in a database. Sharing the cache among more threads or processes simplifies concurrent access to a particular object.

3 LinqToWeb Description Language

The description of each data source consists of two parts - the structure of information and the extraction tasks that collect it. A special-purpose declarative language was designed [10] to describe both parts. Its main features include: simplicity, declarative task description, procedural processing logic, intuitive object oriented interface, and support for inherent and transparent parallelism.

3.1 Type Declarations

The LinqToWeb language offers built-in frequently used types and the ability to define user-defined classes. The type system is divided into value types and extraction object types. Built-in value types are: *int*, *double*, *datetime*, *bool* and *string*. The extraction objects are passed by reference through the extraction tasks. Therefore they can be filled with data subsequently and used as a result of extraction. They are represented by built-in container list and all user-defined classes.

Fig. 2 demonstrates the declaration of user-defined classes containing both list (denoted by *[]*) and built-in properties, they represent the abstraction that the programmer work with.

```

class SearchResults {
    Result[] GoogleResults;
    Result[] BingResults;
}

class Result {
    string Title;
    string Url;
    Result[] SubResults;
}

```

Fig. 2 Example of structuralized type declaration, classes and lists

3.2 Extraction Task Definition

The process of obtaining data is represented by extraction methods. Every method is defined by a method name, its typed arguments and a body. The arguments are

used to pass values and references to objects that the method will use or fill in. Arguments of value types are passed by value and arguments of other types are passed by reference. Fig. 3 demonstrates the extraction methods definition. The entry point of extraction is represented by the method `main`. Value-typed arguments of these methods are automatically exposed as arguments of the extraction, arguments typed as extraction objects are exposed as public objects containing extraction results.

```
main(string query, Result[] GoogleResults){
    [open("http://www.google.com/search?q="+query)]
    googlepage(GoogleResults);
}
googlepage(Result[] items){
    // ... }
```

Fig. 3 Extraction methods definition example. Calls a method `googlepage` on given web page.

Body of extraction method. Method body consists of statements and expressions. The language supports common literals and common operations like numeric operators, strings concatenation and assignments. The code looks like a single-threaded extraction process, but the real execution uses the code that is generated from this declaratively defined task which may process the requests in a parallel and optimized way. Hence it allows the programmer to define the abstraction without concerning e.g. any parallelization issues.

Extraction method invocation. The calling syntax is as usual as in other procedural languages. But there is a difference in the semantics. Methods represent the smallest tasks which execution can be postponed. The code looks like a single process but in fact method calls are not processed immediately the call is postponed until the data is requested. Within the method call, only the method arguments remember that their content can be affected by the called method. Therefore the method is really processed only if the content of the required object is unknown yet.

The language also allows definition of more methods with the same identifier and signature (multimethods). When multimethods are invoked, all their instances are called in an undefined order; such behavior is especially useful for exploiting parallelism and other optimizations.

Data contexts. Since the compound result set can be extracted from various sources, the language introduces data-contexts. Every statement is executed within a data-context which represents current open resource, e.g. current web page. Syntactically, new data-context can be open using square brackets before the statement, including command that opens the resource. There is the `open` command including location of the web page in Fig. 3, 5. The location is relative to the current data-context; opening a resource simulates e.g. following a link from the current page.

The purpose of data-contexts is reuse of the code with different source data. They also simplify design of the extraction tasks. It allows the programmer to write the code more declaratively in a similar way as the human browses the web pages.

Other language constructs. Several other constructs are needed to provide fully usable language. Although most of them are known from procedural languages, there are some differences. Following list describes the most important features:

- **Item addition.** The list object is used as container for elements. In extraction methods lists are write-only. To add an element into the list the following syntax is used: `items [] = element`; In the program that uses generated code, the lists are read-only; they are commonly used as data pipes.
- **Object creation.** To create and initialize an object within a single expression the syntax similar to a method call is used. It is useful when new object has to be created, its properties initialized and the result object added to a list or passed as a method call argument. The syntax looks as follows: `Result(Title=...,Url=...)`.
- **.NET code call.** Sometimes it is useful to use standard C# code. The system supports such methods; they can be used and called in the same way as LinqToWeb methods, but without a possibility of delayed processing.

Extractor patterns. Easy data extraction is the main purpose of the extraction methods. The example of collecting results from the Google search is shown on Fig. 5. All required fragments of a source web page are enumerated; data are extracted from the current data-context. The argument of the `foreach` specifies the searched extractor pattern. When a match is found the system automatically creates local variables and initializes them with the matched data. The system currently supports following extractor patterns (particular examples are shown on Fig. 4, 5):

- **Regexp.** Matching the regular expression [6] is the basic and most commonly used method of navigating the web page. However this method is very hard to use

```

regex(@'\<h3\sclass=r\>\<a\shref=\\"(?<rurl>[^\"]*)\"
"[^>]*\>(?(rtitle>[^(.)]*)\</a\>\</h3\>')
match(@'\<h3 class=r\>\<a href=~@rurl@~\"@IGNORE@~\"@r
title@~\</a\>\</h3\>')

```

Fig. 4 Examples and comparison of *Regexp* and *Mat Extractor Pattern*

```

googlepage(Result[] items) {
    foreach(xmlmatch(@'\<h3 class="r">\<a href=~@rhref@~"
class="l">~@rtitle@~\</a\>\</h3\>'))
        {items[] = Result(Url=rhref, Title=rtitle);}
    foreach(xmlmatch(@'\<a href=~@rhref@~">\<span class="csb
ch"/>Next\</a\>')) [open(rhref)] googlepage(items);
}

```

Fig. 5 Extraction method example, using *XML Extractor Patterns*

in case of complex expressions. The variables in the expression can be identified in the following way: (`?<var_name>_expression_`).

- **Match.** Instead of regular expressions, *Mat Extractor Patterns* can be used. They were introduced in scraping framework AgentMat [3]. *Mat Extractor Patterns* are much more easy to use, but they are not as strong as *regexp*. They consist of piece of text that will be matched. This text is converted into corresponding regular expression. The variables are marked by prefix `~@` and suffix `@~`.
- **Xmlmatch.** LinqToWeb system introduces *XML Extractor Patterns*. They have a form of a fragment of valid XML. The content of the data-context is parsed into XML DOM. *XmlMatch* searches for all matches against the provided XML fragment. Variables are marked in the same way as it is in the *Mat Extractor Patterns*, but they can occur only in the XML attributes value and inner nodes text. *XML Extractor Patterns* allows finding matches within the context of HTML nodes easily. It is also more tolerant for changes in the source web page structure, the programmer is not forced to describe elements and attributes of the source HTML text that are not important for the match, only the structure must be specified. In this case, matched HTML fragment can contain other XML attributes or even other XML child elements. The XML pattern is well-arranged with a possibility to specify wider context of searched information on a higher level of declaration.

4 Generating Strongly Typed Objects

The tasks described in the previous section are used to generate objects in C# language. The result is in a form of a context class that contains all exported objects declared and instantiated inside.

The context class has public read-only properties corresponding to exported objects. They are defined as arguments of the *main* extraction methods typed as extraction objects. Declarations of user-defined classes are placed within the context class too. Finally the constructor of the context class is parameterized. The parameters correspond to value-typed arguments of the *main* method.

4.1 Just-In-Time Extraction

Every non-value-typed object is derived from an abstract class which defines the mechanism of actions. It manages a collection of actions that lead to collecting data for this object. Every action consists of a method to be called and its parameters. When a property of an extraction object is to be read, the getter method of the property processes and removes some action if the value is not known yet. New actions appear in the collection by calling extraction methods as it was mentioned before and by a context class constructor.

By processing the action, some unknown information can be obtained and stored in object properties. Also new actions can be added into the collection. The getter method of the requested property processes actions until the property gets a value.

If the particular property is read repeatedly, its value is extracted only once. If the property is not read, its extraction is not performed at all.

Enumerating lists. Typically, the main result of the extraction task is a set of collections of data. The collections are represented by the list-typed objects. It is also derived from the abstract base class; therefore it manages the mechanism of actions. Instead of using properties, the list object is targeted to act like an enumerator [9] (or a read-only pipe in other words); it can be used only for getting items sequentially.

The runtime uses this enumerator automatically when the program is accessing the collection. The actions associated with the collection are used when the runtime picks up next element from the enumerator, next action from the actions list is processed to obtain more elements if necessary.

Every time the program enumerates the list new enumerator is created and list elements are obtained from beginning. Since frequently used resources and results of extraction operations are cached, such behavior does not affect the performance. Moreover it allows defining endless collections without requirements for endless storage capacity. Also only the beginning of the collection is extracted.

4.2 Language Integrated Query

The system of properties and enumerators allows taking benefits from LINQ easily. It automatically extends all the enumerator objects in .NET with other methods. They implement common operations over collections like *Sort*, *Count*, *GroupBy*, *Sum*, *TakeWhile* etc. As the query is processed all the missing information (e.g. value of properties or enumerated items) are collected and cached transparently.

```
var context = new ContextClass("some query");
foreach (var x in context.GoogleResults.Take(100).Where(x
=> x.Url.StartsWith("https:"))) {
    Console.WriteLine(x.Title + ": " + x.Url);
}
```

Fig. 6 Enumerating lists and LINQ usage in C#. Advanced queries on pipes representing web collection. Writing the single elements onto the output in object oriented and type safe way.

Fig. 6 demonstrates the usage of lists and LINQ extensions. The sample collects results from the Google search; it takes first 100 results and selects only items which URL uses secured HTTP protocol. The programmer can work with collections extracted from the web in a completely abstract object oriented and type safe way. He has also access to more advanced queries without a need of implementing them.

5 Evaluation

Using the LinqToWeb framework several applications have been developed, especially for web semantization subprojects [2, 5]. The code fragments referenced from previous sections illustrate use of the language. The object model is generated from Google search pages; this model is subsequently used to fetch and process the data.

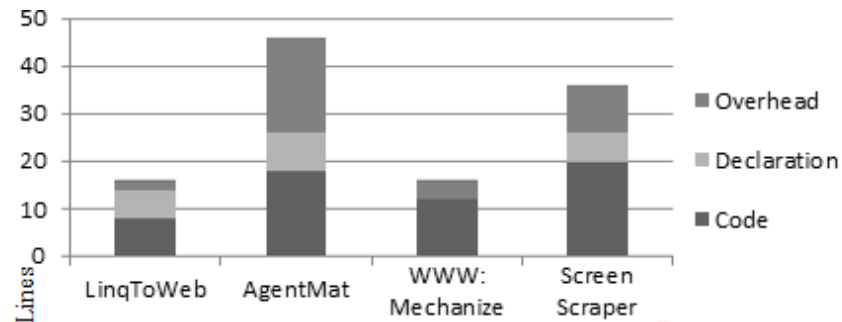


Fig. 7 Extraction frameworks and written code complexity

The size of the code needed to use Google results in the application is depicted on Fig. 7. Most of the frameworks are able to define the task in a declarative way, but the usage of extracted data is not trivial (*AgentMat* and *ScreenScraper* systems). Smaller code means faster development and less chance for errors. Also other frameworks used as libraries force the programmer to mix extraction task within the program itself (e.g., *WWW: Mechanize*). The graph shows amount of code lines defining the extraction task and its usage, declaring object structure or tasks and also the syntax overhead. The overhead is needed to proper usage of the framework, but it does not affect the extraction itself.

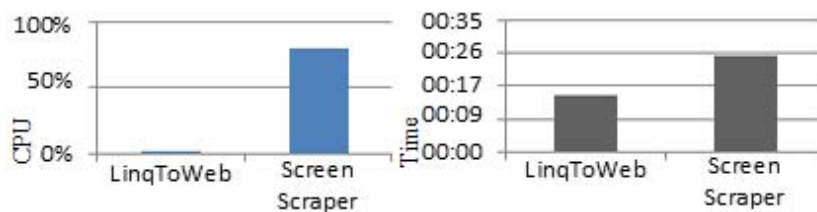


Fig. 8 CPU usage and extraction task speed, compiled vs. interpreted tasks

The LinqToWeb extraction tasks are compiled; it results in faster processing and lower CPU usage, the process is waiting for web responses most of the time. The difference of compiled and interpreted task is depicted on Fig. 8. Both frameworks were used to obtain 400 results from Google search on Core2Duo CPU and a standard broadband connection.

6 Conclusion and Future Work

This paper describes methods used for integrating information from web-based sources into application development. Implemented framework LinqToWeb enables reading data in a type safe and object oriented way. The methods are suitable for frequently updated live data. Using generated object model, data are accessed as strongly typed objects. Hence it allows taking benefits of compiler type checks and IDE code sense capabilities. Moreover, downloading, updating and data extraction is performed on request fully transparently.

The main future enhancement is intelligent ordering and automatic parallelization of processed extraction tasks. Other possible enhancements are related to using a semantic storage for caching. Also some LINQ methods can be extended to modify the extraction process by using the information from the query, e.g. data can be downloaded already sorted if the source web allows it. Other kinds of improvements include automatic detection of a web page structure and its semantic content. Since the framework is intensively used in a web semantization research project, we expect its future development in other not mentioned aspects.

Acknowledgements. This work was supported in part by grants GACR 202/10/0761, SVV-2010-261312, and GAUK 2010/28910.

References

1. Gisle Aas: HTML Parser informations,
<http://search.cpan.org/~GAAS/HTML-Parser/>
2. Bednarek, D., Dokulil, J., Yaghob, J., Zavoral, F.: Using Methods of Parallel Semi-structured Data Processing for SemanticWeb. In: Proceedings of SEMAPRO 2009. IEEE Computer Society Press, Los Alamitos (2009)
3. Beno, M., Misek, J., Zavoral, F.: AgentMat: Framework for Data Scraping and Semantization. In: RCIS, Fez, Morocco (2009)
4. Box, D., Hejlsberg, A.: LINQ:NET Language-Integrated Query. In: MSDN (2007)
5. Dokulil, J., Yaghob, J., Zavoral, F.: Trisolda: The Environment for Semantic Data Processing. International Journal On Advances in Software 2008, IARIA 1(1) (2009)
6. Friedl, J.: Mastering Regular Expressions. O'Reilly Media, Inc., Sebastopol (2006)
7. Kulkarni, D., Bolognese, L., Warren, M., Hejlsberg, A., George, K.: LINQ to SQL:NET Language-Integrated Query for Relational Data
8. Lester, A.: WWW:Mechanize,
<http://search.cpan.org/~petdance/WWW-Mechanize-1.52/>
9. Mackay, C.A.: Using .NET Enumerators, The Code Project (2003),
<http://www.codeproject.com/KB/cs/csenumerators.aspx>
10. Misek, J.: LinqToWeb Language Definition, Technical report KSI 2010/01, Charles University in Prague (2010)
11. Misek, J.: LINQ to Web project, <http://lingtowebs.codeplex.com/>
12. Ekiwi: Screen scraper informations, <http://www.screen-scraper.com/>
13. Kapow Technologies: Kapowtech Mashup Server informations,
<http://www.kapowtech.com>

Chapter 5.

Parallel SPARQL Query Processing Using Bobox

Zbyněk Falt, Miroslav Čermák, Jiří Dokulil, Filip Zavoral

International Journal On Advances in Intelligent Systems Vol. 5, Num. 3, pp. 302-314, 2012

**International Journal on
Advances in Intelligent Systems**



2012 vol. 5 nr. 3&4

Parallel SPARQL Query Processing Using Bobox

Zbyněk Falt, Miroslav Čermák, Jiří Dokulil, and Filip Zavoral
Charles University in Prague, Czech Republic
{falt,cermak,dokulil,zavoral}@ksi.mff.cuni.cz

Abstract—Proliferation of RDF data on the Web creates a need for systems that are not only capable of querying them, but also capable of scaling efficiently with the growing size of the data. Parallelization is one of the ways of achieving this goal. There is also room for optimization in RDF processing to reduce the gap between RDF and relational data processing. SPARQL is a popular RDF query language; however current engines do not fully benefit from parallelization potential. We present a solution that makes use of the Bobox platform, which was designed to support development of data-intensive parallel computations as a powerful tool for querying RDF data stores. A key part of the solution is a SPARQL compiler and execution plan optimizer, which were tailored specifically to work with the Bobox parallel framework. The experiments described in this paper show that such a parallel approach to RDF data processing has a potential to provide better performance than current serial engines.

Keywords—SPARQL; Bobox; query optimization; parallel.

I. INTRODUCTION

SPARQL [2] is a query language for RDF [3] (Resource Definition Framework) widely used in semantic web databases. It contains capabilities for querying graph patterns along with their conjunctions and disjunctions. SPARQL algebra is similar to relational algebra; however, there are several important differences, such as the absence of NULL values. As a result of these differences, the application of relational algebra into semantic processing is not straightforward and the algorithms have to be adapted so it is possible to use them.

As the prevalence of semantic data on the web is getting bigger, the Semantic Web databases are growing in size. There are two main approaches to storing and accessing these data efficiently: using traditional relational means or using semantic tools, such as different RDF triplestores [3] accessed using SPARQL. Semantic tools are still in development and a lot of effort is given to the research of effective storing of RDF data and their querying [4]. One way of improving performance is the use of modern, multicore CPUs in parallel processing.

Nowadays, there are several database engines which are capable of evaluating SPARQL queries, such as SESAME [5], JENA [6], Virtuoso [7], OWLIM [8] or RDF-3X [9], that is currently considered to be one of the fastest single node RDF-store [10]. These stores support parallel computation of multiple queries; however, they mostly do

not use the potential of parallel computation of particular queries.

The Bobox framework [11], [12], [13] was designed to support the development of data-intensive parallel computations. The main idea behind Bobox is to divide a large task into many simple tasks that can be arranged into a non-linear pipeline. The tasks are executed in parallel and the execution is driven by the availability of data on their inputs. The developer does not have to be concerned about problems such as synchronization, scheduling and race conditions. All this is done by the framework. The system can be easily used as a database execution engine; however, each query language requires its own front-end that translates a request (query) into a definition of the structure of the pipeline that corresponds to the query.

In the paper, we present a tool for efficient parallel querying of RDF data [14] using SPARQL build on top of the Bobox framework [1], [15]. The data are stored using an in-memory triple store. We provide a description of query processing using SPARQL-specific parts of the Bobox and provide results of benchmarks. Benchmarks were performed using the SP²Bench [16] query set and data generator.

The rest of the paper is structured as follows: Section II describes the Bobox framework. Models used to represent queries and a description of query processing is contained in Section III. Data representation and the implementation of operators using Bobox framework is described in Section IV. Section V presents our experiments and a discussion of their results. Section VI compares our solution to other contemporary parallelization frameworks. Section VII describes future research directions and concludes the paper.

II. BOBOX FRAMEWORK

A. Bobox Architecture

Bobox is a parallelization framework which simplifies writing parallel, data intensive programs and serves as a testbed for the development of generic and especially data-oriented parallel algorithms.

Bobox provides a run-time environment which is used to execute a non-linear pipeline (we denote it as the *execution plan*) in parallel. The execution plan consists of computational units (we denote them as the *boxes*) which are connected together by directed edges. The task of each box is to receive data from its incoming edges (i.e. from its *inputs*) and to send the resulting data to its outgoing edges

(i.e. to its *outputs*). The user provides the execution plan (i.e. the implementation of boxes and their mutual connections) and passes it to the framework which is responsible for the evaluation of the plan.

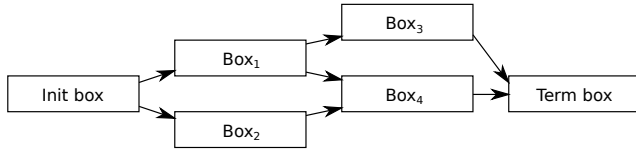


Figure 1. Example of an execution plan

Figure 1 shows an example of an execution plan. Each plan must contain two special boxes:

- *init box* – this is the first box (in a topological order) of the plan which is executed.
- *term box* – this is the last box and denotes that the execution plan was completely evaluated.

The implementation of boxes is quite straightforward and simple, since Bobox provides a very powerful and easy to use interface for their development. Additionally, the source code is expected to be strictly single-threaded. Therefore, the developer does not have to be familiar with parallel programming. Although this requirement on the source code may seem limiting, the framework is especially targeted to a development of highly scalable applications [17].

The only communication between boxes is done by sending *envelopes* (communication units containing data) along their outgoing edges. Each envelope consists of several columns and each column contains a certain number of data items. The data type of items in one column must be the same in all envelopes transferred along one particular edge; however, different columns in one envelope may have different data types. The data types of these columns are defined by the execution plan.

The number of data items in all columns in one envelope must be always the same. Therefore, we may define the list of i -th items of all columns in one envelope as its i -th *data line*. The Figure 2 shows an example of an envelope.

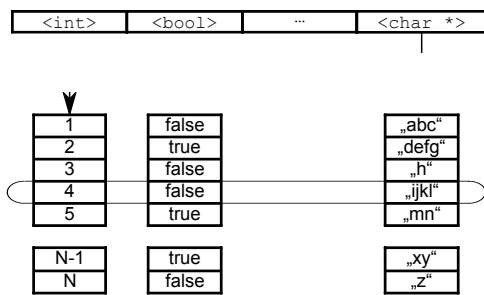


Figure 2. The structure of an envelope

The total number of data lines in an envelope is chosen according to the size of cache memories in the system.

Therefore, the communication may take place completely in cache memory. This increases the efficiency of processing of incoming envelopes by a box.

Currently, only shared-memory architectures are supported; therefore, the only shared pointers to the envelopes are transferred. This speeds up operations such as broadcast box (i.e., the box which resends its input to its outputs) significantly since they do not have to access data stored in envelopes.

There is one special envelope (so called *poisoned pill*) which is sent after the last regular envelope to close the output of a source box. For the receiver of the poisoned pill it is a signal that all data were already received on that input.

In fact, the only work which is done by the init box is sending the poisoned pill to its output and the only responsibility of the term box is to terminate the evaluation of the execution plan when it receives the poisoned pill on its input.

The interface of Bobox for box development is very flexible; therefore, the developer of a box may choose between multiple views on the data communication:

- The communication is a stream of envelopes. This is useful for efficient implementation of boxes which do not have to access data in envelopes such as broadcast box or stream splitter (see Section IV-D) or implementation of boxes which process their inputs by envelopes.
- The communication is a stream of data lines. This is useful for easier implementation of boxes which manipulate with data lines one by one such as filter box (see Section IV-C2).
- The combination of both views. For example, the sort box (see IV-C3) processes input by envelopes, but produces the output as a stream of data lines.

Although the body of boxes must be strictly single-threaded, Bobox may introduce three types of parallelism:

- 1) Task parallelism, when independent streams are processed in parallel.
- 2) Pipeline parallelism, when the producer of a stream runs in parallel with its consumer.
- 3) Data parallelism, when independent parts of one streams are processed in parallel.

The first two types of parallelism are exploited implicitly during the evaluation of a plan. Therefore, even an application which does not contain any explicit parallelism may benefit from multiple processors in the system (see Section V-A). Data parallelism must be explicitly stated in the execution plan by the user (see IV-D); however, it is still much easier to modify the execution plan than writing parallel code by hand.

B. Flow control

Each box has only limited buffer for incoming envelopes. When this buffer becomes full, the producer of the envelopes

is suspended until at least one envelope from the buffer is processed. This strategy increases the performance of the system since the operators which produce data faster than their consumers are able to process are suspended to not to consume the CPU time uselessly. This time may be used to execute other boxes. Additionally, this method yields to lower memory consumption, since there is only a limited number of unprocessed slots which occupy the memory at a time.

On the other hand, this flow control may sometimes yield to a deadlock (see Section V-C) or may limit the level of parallelism (see Section IV-C6 for an example).

C. Box scheduling

Scheduling of boxes is a very important factor which significantly influences the performance of Bobox. The scheduling strategies are described in a more detail in [12].

During the initialization of Bobox, a same number of worker threads as the number of physical processors is created. Only these worker threads may execute the code of boxes. The scheduler has two main data structures:

- Each worker thread has its own double ended queue of *immediate tasks*.
- Each execution plan which is being evaluated has its own queue of *deferred tasks*.

There are three cases when a box is scheduled:

- When a new execution plan is about to evaluate, a new queue of deferred tasks for that plan is created and its init box is put to the front of that queue.
- When a box sends an envelope to another box, the destination box is put to the front of the queue of immediate tasks of the thread which is executing the source box.
- When a box stops to be suspended because of flow control, it is put to the queue of deferred tasks of the corresponding plan.

When the working thread is ready to execute a box, it choose the first existing box in this order:

- 1) The newest box in its queue of immediate tasks. This box receives an envelope created by this thread recently. Therefore, it is probable that this envelope is completely hot in a cache so accessing its data is probably much faster than accessing other envelopes.
- 2) The oldest box in the queue of deferred tasks of the oldest execution plan. This ensures that scheduling of deferred tasks of one execution plan are scheduled fairly. However, the execution plans are prioritized according to their age – the older the execution plan is, the higher priority it has. Each evaluation of an execution plan needs some resources (such as memory for envelopes); therefore, the more plans are being evaluated at a time, the more resources are needed for them. This strategy ensures that if there is a

box to execute from plans which are currently being executed, no new evaluation is started.

- 3) The oldest box in the queue of immediate tasks of another worker thread. Worker threads with shared cache memory are prioritized. This avoids suspending of a worker thread despite the fact that there are boxes to execute. Moreover, the oldest box has the lowest probability to have its input hot in a cache memory of the thread from which the box was stolen. Therefore, *stealing* this box should introduce less performance penalty than stealing the newest box in the same queue.

If there is no box to execute, the worker thread is suspended until some other box is scheduled.

Besides the SPARQL compiler described in this paper, the Bobox framework is used in several related projects - model visualization [18], semantic processing [19], [20], query optimization [21], and scheduling in data stream processing [12], [22].

III. QUERY REPRESENTATION AND PROCESSING

One of the first Bobox applications was SPARQL query evaluator [19]. Since running queries in Bobox needs an appropriate execution plans, SPARQL compiler for Bobox was implemented to generate them from the SPARQL code.

During query processing, the SPARQL compiler uses specialized representation of the query. In the following sections, we mention models used during query rewriting and generation of execution plan.

A. Query Models

Pirahesh et al. [23] proposed the Query Graph Model (QGM) to represent SQL queries. Hartig and Reese [24] modified this model to represent SPARQL queries (SQGM). With appropriate definition of the operations, this model can be easily transformed into a Bobox pipeline definition, so it was an ideal candidate to use.

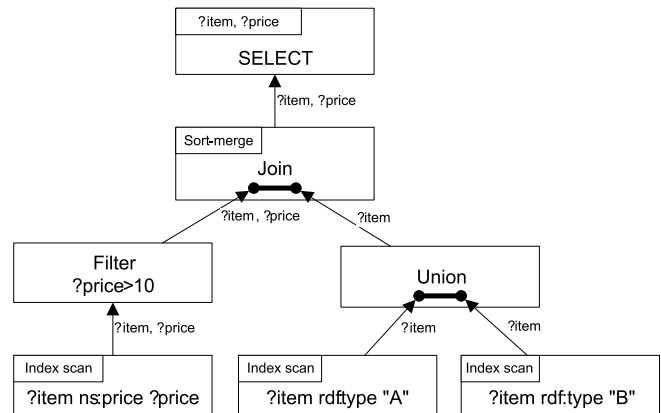


Figure 3. Example of SQGM model.

SQGM model can be interpreted as a directed graph (a directed tree in our case). Nodes represent operators and are depicted as boxes containing headers, body and annotations. Edges represent data flow and are depicted as arrows that follow the direction of the data. Figure 3 shows an example of a simple query represented in the SQGM model. This model is created during an execution plan generation and is used as a definition for the Bobox pipeline.

In [25], we proposed the SPARQL Query Graph Pattern Model (SQGPM) as the model that represents query during optimization steps. This model is focused on representation of the SPARQL query graph patterns [2] rather than on the operations themselves as in the SQGM. It is used to describe relations between group graph patterns (graph patterns consisting of other simple or group graph patterns). The ordering among the graph patterns inside a group graph pattern (or where it is not necessary in order to preserve query equivalency) is undefined. An example of the SQGPM model graphical representation is shown in Figure 4.

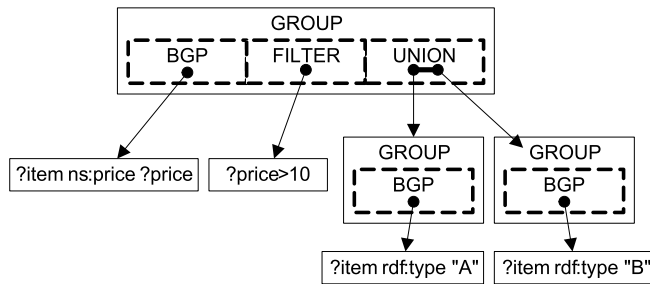


Figure 4. Example of SQGPM model.

Each node in the model represents one group graph pattern that contains an unordered list of references to graph patterns. If the referenced graph pattern is a group graph pattern then it is represented as another SQGPM node. Otherwise the graph pattern is represented by a leaf.

The SQGPM model is built during the syntactical analysis and is modified during the query rewriting step. It is also used as a source model during building the SQGM model.

B. Query Processing

Query processing is performed in a few steps by separate modules of the application as shown in Figure 5. The first steps are performed by the SPARQL front-end represented by compiler. The main goal of these steps is to validate the compiled query, pre-process it and prepare the optimal execution plan according to several heuristics. Execution itself is generated by the Bobox back-end where execution pipeline is initialized according to the plan from the front-end. Following sections describe steps done by the compiler in a more detail way.

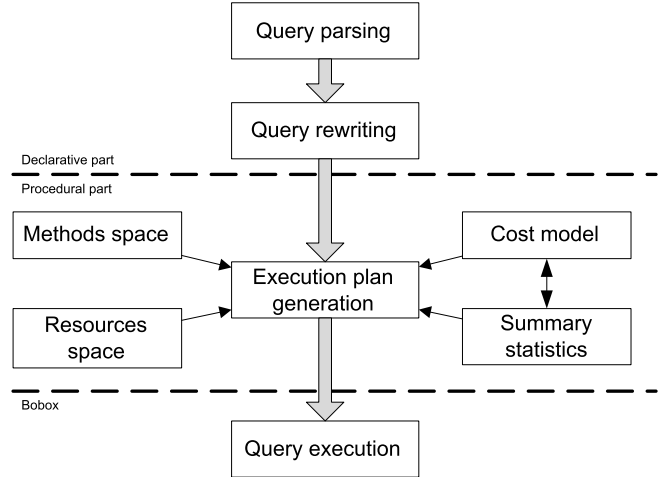


Figure 5. Query processing scheme.

C. Query Parsing and Rewriting

The query parsing step uses standard methods to perform syntactic and lexical analysis according to the W3C recommendation. The input stream is transformed into a SQGPM model. The transformation also includes expanding short forms in queries, replacing aliases and a transformation of blank nodes into variables.

The second step is query rewriting. We cannot expect that all queries are written optimally; they may contain duplicities, constant expressions, inefficient conditions, redundancies, etc. Therefore, the goal of this phase is to normalize queries to achieve a better final performance. We use the following operations:

- Merging of nested *Group graph patterns*
- Duplicities removal
- *Filter*, *Distinct* and *Reduced* propagation
- Projection of variables

During this step, it is necessary to check applicability of each operation with regards to the SPARQL semantics before it is used to preserve query equivalency [25].

D. Execution Plan Generation

In the previous steps, we described some query transformations that resulted in a SQGPM model. However, this model does not specify a complete order of all operations. The main goal of the execution plan generation step is to transform the SQGPM model into an execution plan. This includes selecting orderings of join operations, join types and the best strategy to access the data stored in the physical store.

The query execution plan (e.g., the execution plan of query q5a is depicted in Figure 6) is built from the bottom to the top using dynamic programming to search part of the search space of all possible joins. This strategy is applied to each group graph pattern separately because the order of

the patterns is fixed in the SQGPM model. Also, the result ordering is considered, because a partial plan that seems to be worse locally, but produces a useful ordering of the result, may provide a better overall plan. The list of available atomic operations (e.g., the different types of joins) and their properties are provided by the *Methods Space* module.

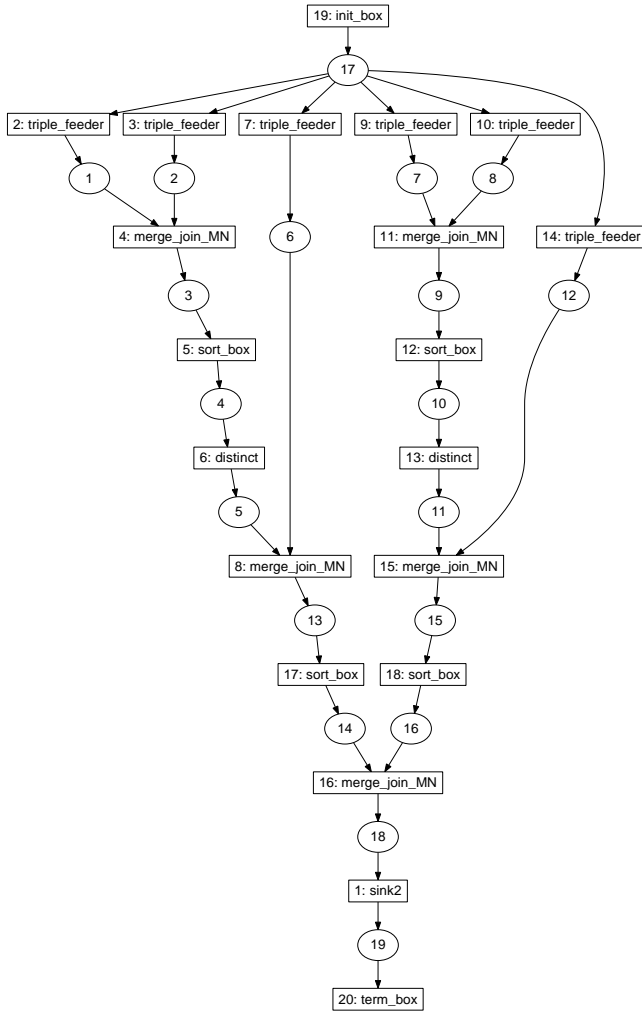


Figure 6. Query execution plan q5a.

In order to compare two execution plans, it is necessary to estimate the *cost* of both plans – an abstract value that represents the projected cost of execution of a plan using the actual data. This is done with the help of the *cost model* that holds information about atomic operation efficiency and *summary statistics* gathered about the stored RDF data.

The search space of all execution plans could be extremely large; we used heuristics to reduce the complexity of the search. Only left-deep trees of join operations are considered. This means that right operand of a join operation may not be another join operation. There is one exception to this rule – avoiding cartesian products. If there is no other way to add another join operation without creating

cartesian product, the rest of unused operations is used to build separate tree recursively (using the same algorithm) and the result is joined with the already built tree. This modification greatly improves plans for some of the queries we have tested and often significantly reduces the depth of the tree.

The final execution plan is represented using SQGM model which is serialized into a textual form and passed to the Bobox framework for evaluation.

IV. EVALUATION OF SPAQRL QUERIES USING BOBOX

When the compiler finishes the compilation, a query execution plan is generated. This plan must be transformed into a Bobox execution plan and then passed to Bobox for its evaluation. This basically means that operators must be replaced by boxes and they should be connected to form a pipe. Additionally, an efficient representation of data exchanged by boxes must be chosen to process the query efficiently.

A. Data representation

1) *Representation of RDF terms:* RDF data are typically very redundant, since they contain many duplicities. Many triples typically share the same subjects or predicates. To reduce the number of memory needed for storing the RDF data, we keep only one instance of every unique string and only one instance of every unique term in a memory. Besides the fact that this representation saves the memory, we may represent each term unambiguously by its address. Therefore, for example in case of a join operation, we can test equality of two terms just by a comparison of their addresses.

Additionally, if we need to access the content of a term (e.g. for evaluation of a filter condition) the address can be easily dereferenced. This is faster than the representation of terms by other unique identifiers which would have to be translated to the term in a more complicated way.

2) *Representation of RDF database:* The database consists of a set of triples. We represent this set as three parallel arrays with the same size which contain addresses of terms in the database. In fact, we keep six copies of these arrays sorted in all possible orders – SPO, SOP, OPS, OSP, PSO and POS. This representation makes implementation of index scans extremely efficient (see IV-C1).

3) *Format of envelopes:* The format of envelopes is now obvious. It contains columns which correspond to a subset of variables in the query in a form of an address of a particular RDF term. One data line of an envelope corresponds to one possible mapping of variables to their values.

B. Transformation of query execution plan

The output of the compiler is produced completely in a textual form. Therefore, the Bobox must deserialize the query plan first. Despite the fact that this serialization and

deserialization have some overhead, we chose it because of these benefits:

- When distributed computation support is added, the text representation is safer than a binary representation where problems with different formats, encodings or reference types may appear.
- The serialization language has a very simple and effective syntax; serialization and deserialization are much faster than (e.g.) the use of XML. Therefore, the overhead is not so significant.
- The text representation is independent on the programming language; new compilers can be implemented in a different language.
- Compilers can generate plans that contain boxes that have not yet been implemented, which allows earlier testing of the compiler during the development process.
- The query plan may be easily visualized to check the correctness of the compiler. Moreover, the plan might be written by hand which makes the testing of boxes easier. Altogether, this enables debugging of a compiler and Bobox independently on each other.

When the plan is deserialized, the operators in the query execution plan must be replaced by boxes and connected together. The straightforward approach is that each operator in the query execution plan is implemented by exactly one box. Even this approach yields to a parallel evaluation of the plan since pipeline parallelism and task parallelism might be exploited (the query plan has typically a form of a rooted tree with several independent branches). However, it is still usually insufficient to utilize all physical threads available and the most time consuming operations such as nested loops join becomes a bottleneck of the plan. Therefore, they have to be parallelized explicitly. We describe this modification in IV-D.

C. Implementation of query plan operators

1) *Index scan*: The main objective of a scan operation is to fetch all triples from the database that match the input pattern. Since we keep all triples in all possible orders, it is easy for any input pattern to find the range where all triples which match the pattern are. To find this range, we use binary search. To avoid copying triples from the database to the envelopes, we use the fact that they are stored in parallel arrays. Therefore, we may use the appropriate subarrays directly as columns of output envelopes without data copying.

2) *Filter*: A filter operation can be implemented in Bobox very easily. The box reads the input as a stream of data lines, evaluates the filter condition on each line and sends out the stream of that data lines which meet the condition.

The evaluation of the filter condition is straightforward since each data line contains addresses of respective RDF terms and by dereferencing them it gets full info about the term such as its type, string/numeric value etc.

3) *Sort*: Sort is a blocking operation, i.e. it must wait until all input data are received before it starts to produce output data. To increase the pipeline parallelism, we implemented two phase sorting algorithm [17] inspired by external merge sort.

In the first phase, every incoming envelope is sorted independently on other envelopes. This phase is able to run in parallel with the part of an execution plan which precedes the sort box. The second phase uses a multiway merge algorithm to merge all received (and sorted) envelopes into the resulting stream of data lines. In contrary to the first phase, this phase may run in parallel with the part of the execution plan which succeeds the sort box.

4) *Merge join*: Merge join is a very efficient join algorithm when both inputs are sorted by the common variables. Moreover, the merge join is the algorithm which is suitable for systems like Bobox since it reads both inputs sequentially allowing both input branches to run in parallel (in contrary to hash join, see Section IV-C6).

5) *Nested loops join*: The SPARQL compiler selects nested loops join when the inputs have no common variable and the result is determined only by the join condition. The implementation is straightforward; however, in order to increase the pipeline parallelism, the box tries to process envelopes immediately as they arrive, i.e. it does not read the whole input before processing the other.

6) *Hash join*: Hash join is used when the inputs have some common variables which are not sorted in the same order. In order to increase pipeline and task parallelism, we decided not to implement this algorithm. The problem with hash join is that it must read the whole one input first before processing the second one. However, the branch of the plan which produces data for the second input may be blocked because of flow control (see Section II-B) until the first input is completely processed.

Therefore, instead of hash join we implemented sort-merge join. The sort operation is used to transform the inputs to be usable by merge join.

7) *Optional joins*: Optional join works basically in the same way as regular join. The only difference is that data lines from the left input which do not meet the join condition (i.e., they are not joined with any data line from the right input), are also passed to the output and the variables which come from the right input are set as unbound.

This modification can be easily done when exactly one data line from the left is joined with exactly one data line from the right. In other cases we must keep information about data lines from the left which were already joined and which were not. To do this, each incoming envelope from the left input is extended by one column of boolean values initially set to `false`. When a data line from the left is joined with some data line from the right, we set corresponding boolean value to `true`. When the algorithm finishes, we know which left data lines were not joined and

should be copied to the output.

8) *Distinct*: Operator distinct should output only unique data lines. We implemented this operator by the modification of a sort operator. The first phase is completely the same; however, during the merging in the second phase, the duplicated data lines are omitted from the output.

9) *Other operators*: The rest of operators is implemented very straightforwardly. Therefore, we do not describe them here.

D. Explicit parallelization of nested loops join

With the set of boxes described in Section IV-C, we can evaluate the complete SP²Bench benchmark (see Section V). Despite the fact that the implicit parallelization speeds up the evaluation of several queries, this speed up does not scale with the number of physical cores in the host system.

Therefore, we focused on the most time-consuming operation – nested loops join – and tried to explicitly parallelize it using Bobox.

The task of nested loops join is to evaluate the join condition on all pairs of data lines from the left input and data lines from the right input.

This operation can be easily parallelized, since we can create N boxes which perform nested loops join (N denotes the number of worker threads used by Bobox). We pass one N -th of one input and the whole second input to each of these boxes and join their outputs together. It can be easily seen that this modification is valid since all pairs of data lines are still correctly processed. The whole schema of boxes is depicted in Figure 7.

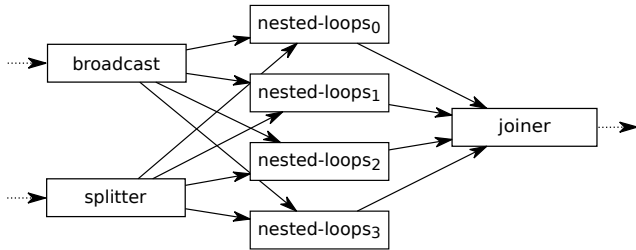


Figure 7. Parallelized nested loops join

The box *splitter* splits its input envelopes to N parts and sends these parts to its outputs. The implementation of this box must be careful since rounding errors may cause that splitted streams do not have the same length. The box *broadcast* just resends its every incoming envelope to its outputs and the box *joiner* resends any incoming envelope to its output.

Since all these three boxes are already implemented in Bobox as standard boxes, the parallelization of nested loops join is very simple.

V. EXPERIMENTS

We performed a number of experiments to test functionality, performance and scalability of the SPARQL query engine. The experiments were performed using the SP²Bench [16] query set since this benchmark is considered to be a standard in the area of semantic processing.

Experiments were performed on a server running Redhat 6.0 Linux; server configuration is 2x Intel Xeon E5310, 1.60Ghz (L1: 32kB+32kB L2: 4MB shared) and 8GB RAM. It was dedicated specially to the testing; therefore, no other application were running on the server during measurements. SPARQL front-end and Bobox are implemented in C++. Data were stored in-memory.

A. Implicit parallelization

In the first experiment, we measured the speed up caused by the implicit parallelization exploited by Bobox. To measure it, we chose some queries and evaluated them with an increasing number of worker threads. We did not use parallelized version of nested loops join in this experiment and we measured only runtime of evaluation of execution plan, i.e. we did not include the time spent by compilation of the query. The results are shown in Figure 8.

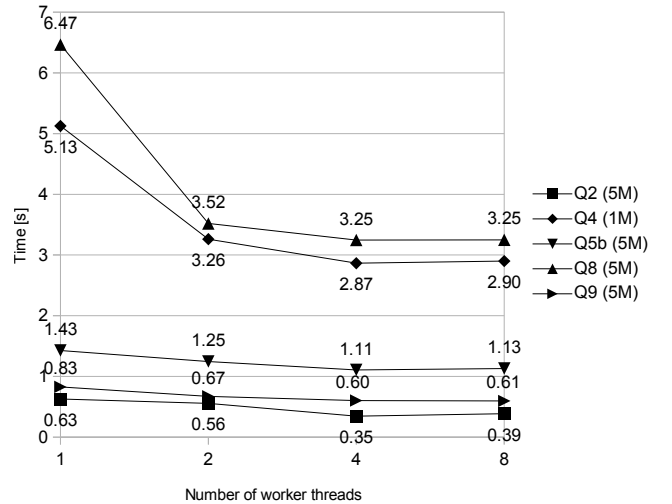


Figure 8. The speed up obtained by implicit parallelization

The results show that for some queries the speed up is quite significant; however, it does not scale with the increasing number of worker threads. This is caused by the fact that the level of parallelism is implicitly built in the execution plan which does not depend on the number of worker threads.

The query Q4 and Q8 benefits from the parallel evaluation most, since the last sort box (or distinct box respectively) runs in parallel with the rest of the execution plan. That is not the case of Q9 which contains distinct box as well; however, the amount of data processed by this box is too small to fully exploit the pipeline parallelism.

	Q1	Q2	Q3a	Q3b	Q3c	Q4	Q5a/b	Q6	Q7	Q8	Q9	Q10	Q11
10k	1	147	846	9	0	23.2k	155	229	0	184	4	166	10
50k	1	965	3.6k	25	0	104.7k	1.1k	1.8k	2	264	4	307	10
250k	1	6.2k	15.9k	127	0	542.8k	6.9k	12.1k	62	332	4	452	10
1M	1	32.8k	52.7k	379	0	2.6M	35.2k	62.8k	292	400	4	572	10
5M	1	248.7k	192.4k	1.3k	0	18.4M	210.7k	417.6k	1.2k	493	4	656	10

Table I
QUERY RESULT SIZES ON DOCUMENTS UP TO 5M TRIPLES.

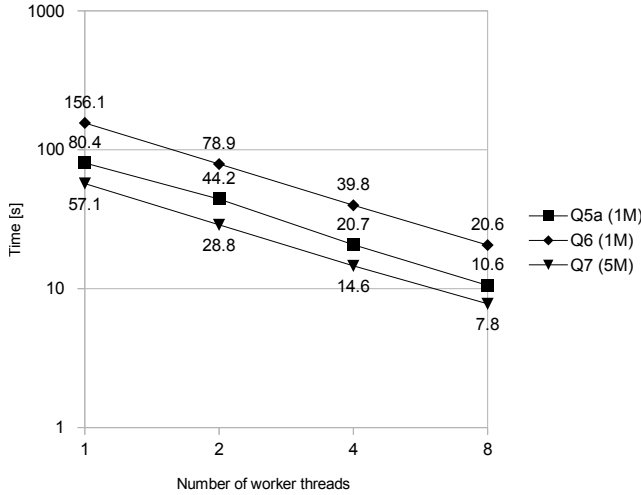


Figure 9. The speed up obtained by explicit parallelization of nested loops join

B. Explicit parallelization

In the second experiment, we focused on the speed up caused by the explicit parallelization of nested loops join. We selected the most time consuming queries with the nested loops joins. As in the first experiment, we performed multiple measurements with the increasing number of worker threads. In this experiment we also did not include the time needed by the query compilation since we focused on the runtime.

The results are shown in Figure 9. According to our expectations, data parallelism increases the scalability and causes a significant almost linear speed up on multiprocessor systems.

C. Comparison with other engines

The last set of experiments compares the Bobox SPARQL engine to other mainstream SPARQL engines, such as Sesame v2.0 [5], Jena v2.7.4 with TDB v0.9.4 [6] and Virtuoso v6.1.6.3127 (multithreaded) [7]. They follow client-server architecture and we provide sum of the times of client and server processes. The Bobox engine was compiled as a single application; we applied timers in the way that document loading times were excluded to be comparable with a server that has data already prepared.

For all scenarios, we carried out multiple runs over documents containing 10k, 50k, 250k, 1M, and 5M triples and we provide the average times. Each test run was also limited to 30 minutes (the same timeout as in the original SP²Bench paper). All data were stored in-memory, as our primary interest is to compare the basic performance of the approaches rather than caching etc. The expected number of the results for each scenario can be found in Table I.

The query execution times are shown in Figure 10. The y-axes are shown in a logarithmic scale and individual plots scale differently. In the following paragraphs, we discuss some of the queries and their results. In contrary to previous experiments, we did include the time spent by the compiler in order to be comparable with other engines.

Q2 implements a bushy graph pattern and the size of the result grows with the size of the queried data. We can see that Bobox Engine scales well, even though it creates execution plans shaped as a left-deep tree. This is due to the parallel stream processing of merge joins. The reason why our solution is slower on 10k and 50k of triples is that the compiler takes more than 1s to compile and to optimize the query.

The variants of Q3 (labelled *a* to *c*) test FILTER expression with varying selectivity. We present only the results of Q3c as the results for Q3a and Q3b are similar. The performance of Bobox is negatively affected by a simple implementation of statistics used to estimate the selectivity of the filter.

Q4 (Figure 11) contains a comparably long graph chain, i.e., variables `?name1` and `?name2` are linked through articles that (different) authors have published in the same journal. Bobox embeds the FILTER expression into this computation instead of evaluating the outer pattern block and applying the FILTER afterwards and propagates the DISTINCT modifier closer to the leaves of the plan in order to reduce the size of the intermediate results.

Queries Q5 (Figure 11) test implicit (Q5a) join encoded in a FILTER condition and explicit (Q5b) variant of joins. On explicit join both engines used fast join algorithm and are able to produce result in a reasonable time. On implicit join both engines used nested loops join which scales very badly. However, Bobox outperforms both Sesame and Jena since it is able to use multiple processors to get the results and is able to compute also documents with 250k, 1M and 5MB

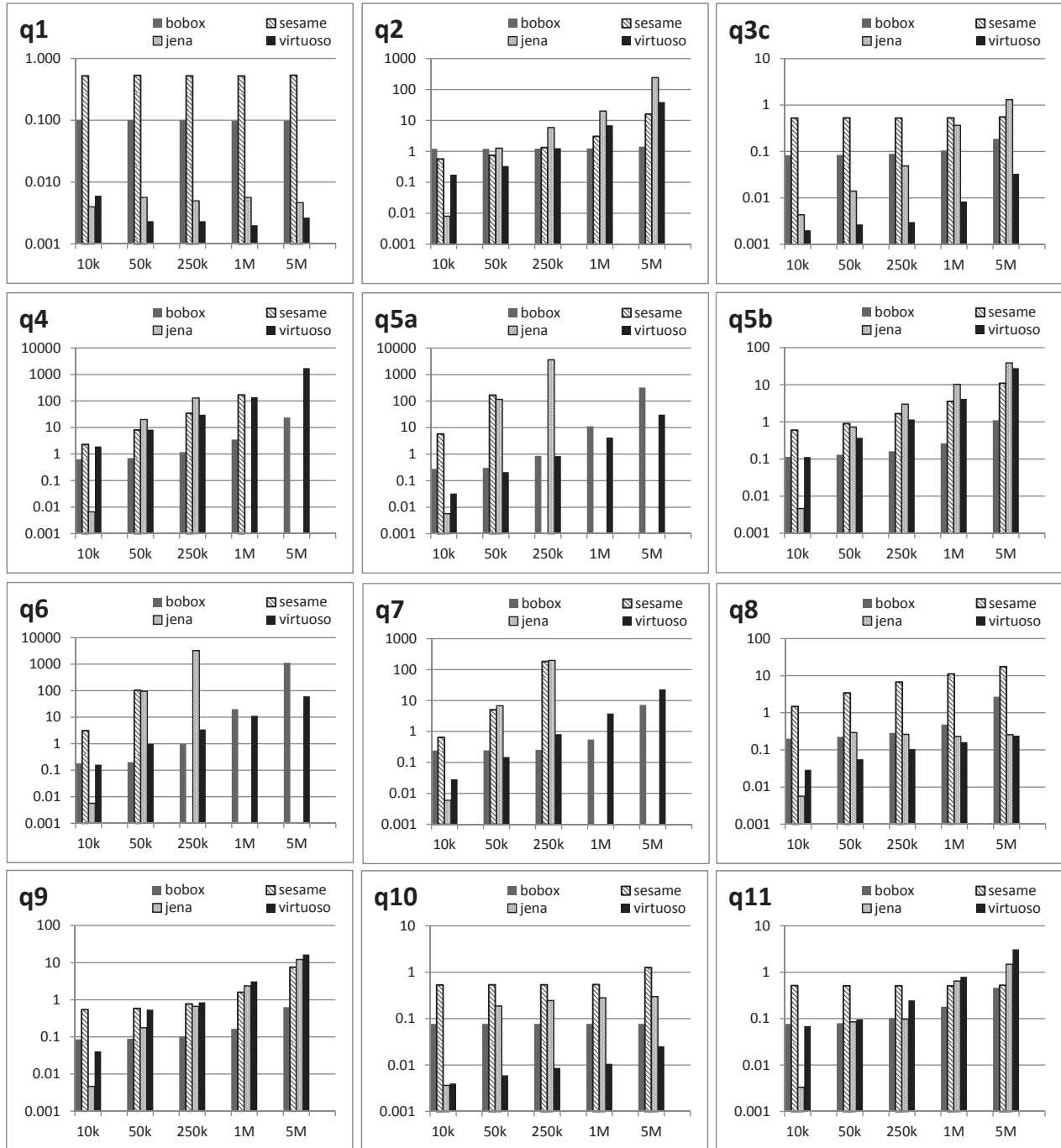


Figure 10. Results (time in seconds) for 10k, 50k, 250k, 1M, and 5M triples.

triples before the 30 minute limit is reached. On the other hand, Virtuoso outperforms Bobox mainly due to particular query optimizations [16].

Queries Q6, Q7 and Q8 produce bushy trees; their computation is well handled in parallel, mainly because of nested loops join parallelization. As a result of this, Bobox outperforms Sesame and Jena in Q6 and Q7 and outperforms

Virtuoso in Q7, being able to compute larger documents until the query times out. The authors of the SP²Bench suggest [16] reusing graph patterns in a description of the queries Q6, Q7 and Q8. However, this is problematical in Bobox. Bobox processing is driven by the availability of the data on inputs but it also incorporates methods to prevent the input buffers from being overfilled (see Section II-B).

```

SELECT DISTINCT ?name1 ?name2
WHERE { ?article1 rdf:type bench:Article.
        ?article2 rdf:type bench:Article.
        ?article1 dc:creator ?author1.
        ?author1 foaf:name ?name1.
        ?article2 dc:creator ?author2.
        ?author2 foaf:name ?name2.
        ?article1 swrc:journal ?journal.
        ?article2 swrc:journal ?journal
        FILTER (?name1<?name2) }

```

Q4

```

SELECT DISTINCT ?person ?name
WHERE { ?article rdf:type bench:Article.
        ?article dc:creator ?person.
        ?inproc rdf:type bench:Inproceedings.
        ?inproc dc:creator ?person2.
        ?person foaf:name ?name.
        ?person2 foaf:name ?name2
        FILTER(?name=?name2) }

```

Q5a

Figure 11. Examples of the benchmark queries.

Pattern reusing can result in the same data being sent along two different paths in the pipeline running at a different speed. Such paths may then converge in a join operation. When the faster path overfills the input buffer of the join box, the computation of all boxes on paths leading to the box is suspended. As a result, data for the slower path will never be produced and will not reach the join box, which results in a deadlock. We intend to examine the possibility of introducing a buffer box, which will be able to store and provide data on request. This way, the Bobox SPARQL implementation will be able to reuse graph patterns.

Q10 can be processed very fast because of our database representation. Therefore, only resulting triples are fetched directly from the database.

In contrary to Sesame, time of Q11 depends on the size of database. This is caused by the fact that we do not have any optimization for queries with LIMIT or OFFSET modifiers. In that case, the whole results set is produced which naturally slows down the evaluation.

Overall, the results of the benchmarks indicate very good potential of Bobox when used for implementation of RDF query engine. Our solution outperforms in all measurements Sesame, in most cases significantly and in most measurements Jena. The performance of Bobox and Virtuoso is comparable; Bobox outperforms Virtuoso namely in computing and data intensive queries.

VI. RELATED WORK

A. Parallel Frameworks and Libraries

The most similar to the Bobox run-time is the TBB library [26]. It was one of the first libraries that focused on task level parallelism. Compared to the Bobox, it is a low-level solution – it provides basic algorithms like parallel for cycle or linear pipeline and a very efficient task scheduler. The

developers are able to directly create tasks for the scheduler and create their own parallel algorithms. But the tasks are designed in a way that makes it very hard to create a non-linear pipeline similar to the one Bobox provides. Such pipeline may be necessary for complex data processing [27]. Bobox also provides more services for data passing and flow control.

The latest version of OpenMP [28] also provides a way to execute tasks in parallel, but it provides less features and less control than TBB. The OpenMP library is mainly focused on mathematical computations – it can execute simple loops in parallel really fast, it can also run blocks of code in parallel, but it is not well suited for parallel execution of a complex structure of blocks. Unlike TBB or Bobox, it is a language extension and not just a library; the compiler is well aware of the parallelization and optimize the code better, but it also enables OpenMP to provide features that cannot be done with just a library, like defining the way variables are shared among threads with a simple declaration. In TBB such variable has to be explicitly passed to an appropriate algorithm by the programmer. In Bobox, it must either be explicitly passed to the model or sent using an envelope at run-time.

Some of the architectural decisions could be implemented in a different manner. One way would be to create a thread for each box and via in the model instance. This would also ensure that each box or via is running at most once at any given time. However, this is considered a bad practice [29]. There are two main reasons for not using this architecture. First, it creates a large number of threads, usually much larger than the number of CPU cores. Although it forces the operating system to switch the threads running on a core, it may not impact the overall performance that badly, since it can be arranged that the idling threads (those assigned to a box or via that is not processing any data at the moment) are suspended and do not consume any CPU time. The second problem is that when data (envelopes) are transferred from one box to another, there is very little chance that it would still be hot in the cache, since the thread that corresponds to the second box is likely to be scheduled to a different CPU, that does not share its cache with the original one. The concept of tasks used by TBB and Bobox avoid these problems and the use of thread pool, fixed number of threads and explicit scheduling gives developers of the libraries better control of parallel execution.

Besides these low-level techniques of parallel data processing, the *MapReduce* approach gained significant attention. While it is often considered a step back [30], there are application areas where MapReduce may outperform a parallel database [31]. Although MapReduce was originally targeted to other environments, it was also studied in shared-memory settings (similar to Bobox) [32], [33]. Unlike MapReduce, Bobox is designed to support more complicated processing environment, namely nonlinear pipelines.

B. Parallel Databases

In a relational database management system, parallelism may be employed at various levels of its architecture:

- *Inter-transaction parallelism.* Running different transactions in parallel has been a standard practice for decades. Besides dealing with disk latency, it is also the easiest way to achieve a degree of parallelism in shared-memory or shared-disk environment. Although it is not considered a specific feature of parallel databases, it must be carefully considered in the design of parallel databases since parallel transactions compete for memory, cache, and bandwidth resources [34], [35].
- *Intra-transaction parallelism.* Queries of a transaction may be executed in parallel, provided they do not interfere among themselves and they do not interact with external world. Since these conditions are met rather rarely, this kind of parallelism is seldom exploited except for experiments [36].
- *Inter-operator parallelism.* Since individual operators of a physical query plan have well-defined interfaces and mostly independent behavior, they may be arranged to run in parallel relatively easily. On the other hand, the effect of such parallelism is limited because most of the cost of a query plan is often concentrated in one or a few of the operators [37].
- *Intra-operator parallelism.* Parallelizing the operation of a single physical operator is the central idea of parallel databases. From the architectural point of view, there are two different approaches:
 - a) *Partitioning* [38] – this technique essentially distributes the workload using the fact that many physical algebra operators are distributive with respect to union (or may be rewritten using such operators).
 - b) *Parallel algorithms* – implementing the operator using a parallel algorithm usually offers the freedom of control over the time and resource sharing and machine-specific means like atomic operations or SIMD instructions. However, designing, implementing, and tuning a parallel algorithm is an extremely complex task, often producing errors or varying performance results [39]. Moreover, the evolution of hardware may soon make a parallel implementation obsolete [40]. For these reasons, parallelizing frameworks are developed [41].

The central principle of Bobox allows parallelism among boxes but prohibits (thread-based) parallelism inside a box. This is similar to inter-transaction and inter-operator parallelism; however, a box does not necessarily correspond to a relational operator. In particular, Bobox allows the same approach to partitioning as in parallel databases, using transformation of the query plan.

Bobox does not allow parallel algorithms to be implemented inside a box (except of the use of SIMD instructions). Therefore, individual single-threaded parts of a parallel algorithm must be enclosed in their boxes and the complete algorithm must be built as a network of these boxes. This is certainly a limitation in the expressive power of the system; on the other hand, the communication and synchronization tasks are handled automatically by the Bobox framework.

VII. CONCLUSIONS AND FUTURE WORK

In the paper, we presented a parallel SPARQL processing engine that was built using the Bobox framework with a focus on efficient query processing: parsing, optimization, transformation and parallel execution. We also presented the parallelization of nested loops join algorithm to increase parallelism during the evaluation of time consuming queries. Despite the fact that this parallelization is very simple to be done using Bobox, the measurements show that it scales very well in a multiprocessor environment.

To test the performance, we performed multiple sets of experiments. We have chosen established frameworks for RDF data processing as the reference systems. The results seem very promising; using SP²Bench queries we have identified that our solution is able to process many queries significantly faster than other engines and to obtain results on larger datasets. Therefore, such a parallel approach to RDF data processing has a potential to provide better performance than current engines. On the other hand, we also identified several issues:

- We are working on improvements of our statistics used by the compiler to generate more optimal query plans.
- The pilot implementation of the compiler is not well optimized which is problem especially in Q1 and Q2.
- Our heuristics sometimes result in long chains of boxes. Streamed processing and fast merge joins minimize this disadvantage; however, it is better to have bushy query plans for efficient parallel evaluation.
- Also, some methods proposed in SP²Bench, such as graph pattern reuse, are not efficiently applicable in the current Bobox version.
- The query Q4 is very time consuming and does not benefit much from the fact that the system has multiple processors. Therefore, we must parallelize besides the nested loops join also merge join, which is the bottleneck of this query.
- Currently, we support only in-memory databases. In order to have engine usable for processing of really large RDF databases such as BTC Dataset (Billion triple challenge) [42], we must keep the database in external memory.

Because of these issues, we are convinced that there is still space for optimization in parallel RDF processing and we want to focus on them and improve our solution.

ACKNOWLEDGMENTS

The authors would like to thank the GAUK project no. 28910, 277911 and SVV-2012-265312, and GACR project no. 202/10/0761, which supported this paper.

REFERENCES

- [1] M. Cermak, J. Dokulil, Z. Falt, and F. Zavoral, "SPARQL Query Processing Using Bobox Framework," in *SEMAPRO 2011, The Fifth International Conference on Advances in Semantic Processing*. IARIA, 2011, pp. 104–109.
- [2] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," W3C Recommendation, 2008.
- [3] J. J. Carroll and G. Klyne, *Resource Description Framework: Concepts and Abstract Syntax*, W3C, 2004. [Online]. Available: <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>
- [4] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan, "Efficiently querying rdf data in triple stores," in *Proceeding of the 17th international conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 1053–1054.
- [5] J. Broekstra, A. Kampman, and F. v. Harmelen, "Sesame: A generic architecture for storing and querying RDF and RDF schema," in *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*. London, UK: Springer-Verlag, 2002, pp. 54–68.
- [6] "Jena – a semantic web framework for Java," <http://jena.sourceforge.net>. [Online]. Available: <http://jena.sourceforge.net>, retrieved 10/2012
- [7] "Virtuoso data server," <http://virtuoso.openlinksw.com>, retrieved 10/2012
- [8] A. Kiryakov, D. Ognyanov, and D. Manov, "Owlim a pragmatic semantic repository for owl," 2005, pp. 182–192.
- [9] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB Journal*, vol. 19, pp. 91–113, February 2010.
- [10] J. Huang, D. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, 2011.
- [11] Z. Falt, D. Bednarek, M. Cermak, and F. Zavoral, "On Parallel Evaluation of SPARQL Queries," in *DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*. IARIA, 2012, pp. 97–102.
- [12] D. Bednarek, J. Dokulil, J. Yaghob, and F. Zavoral, "Data-Flow Awareness in Parallel Data Processing," in *6th International Symposium on Intelligent Distributed Computing - IDC 2012*. Springer-Verlag, 2012.
- [13] "The Bobox Project - Parallelization Framework and Server for Data Processing," 2011, Technical Report 2011/1. [Online]. Available: <http://www.ksi.mff.cuni.cz/bobox>, retrieved 12/2012
- [14] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel, "An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario," in *ISWC, Karlsruhe*, 2008, pp. 82–97.
- [15] D. Bednarek, J. Dokulil, J. Yaghob, and F. Zavoral, "Bobox: Parallelization Framework for Data Processing," in *Advances in Information Technology and Applied Computing*, 2012.
- [16] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "Sp2bench: A sparql performance benchmark," *CoRR*, vol. abs/0806.4627, 2008.
- [17] Z. Falt, J. Bulanek, and J. Yaghob, "On Parallel Sorting of Data Streams," in *ADBIS 2012 - 16th East European Conference in Advances in Databases and Information Systems*, 2012.
- [18] J. Dokulil and J. Katreniakova, "Bobox model visualization," in *14th International Conference Information Visualisation*. London, UK: IEEE Computer Society, 2010, pp. 537–542.
- [19] D. Bednarek, J. Dokulil, J. Yaghob, and F. Zavoral, "Using Methods of Parallel Semi-structured Data Processing for Semantic Web," in *3rd International Conference on Advances in Semantic Processing, SEMAPRO*. IEEE Computer Society Press, 2009, pp. 44–49.
- [20] J. Galgonek, "Tequila - a query language for the semantic web," in *DATESO 2009*, ser. CEUR Workshop Proceedings, K. Richta, J. Pokorný, and V. Snášel, Eds., vol. 471. Czech Technical University in Prague, 2009, pp. 105–118.
- [21] M. Krulis and J. Yaghob, "Revision of relational joins for multi-core and many-core architectures," in *Proceedings of the Dateso 2011*. Pisek, Czech Rep.: FEECS, 2011.
- [22] Z. Falt and J. Yaghob, "Task Scheduling in Data Stream Processing," in *Proceedings of the Dateso 2011 Workshop*. Citeseer, 2011, pp. 85–96.
- [23] H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/rule based query rewrite optimization in starburst," *SIGMOD Rec.*, vol. 21, pp. 39–48, June 1992.
- [24] O. Hartig and R. Heese, "The SPARQL Query Graph Model for query optimization," in *The Semantic Web: Research and Applications*, ser. Lecture Notes in Computer Science, E. Franconi, M. Kifer, and W. May, Eds. Springer Berlin / Heidelberg, 2007, vol. 4519, pp. 564–578.
- [25] M. Cermak, J. Dokulil, and F. Zavoral, "SPARQL Compiler for Bobox," *Fourth International Conference on Advances in Semantic Processing*, pp. 100–105, 2010.
- [26] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in Intel Threading Building Blocks," *Intel Technology Journal*, vol. 11, no. 04, pp. 309–322, November 2007.
- [27] D. Bednárek, "Bulk evaluation of user-defined functions in XQuery," Ph.D. dissertation, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, 2009.

- [28] *OpenMP Application Program Interface, Version 3.0*, OpenMP Architecture Review Board, May 2008, <http://www.openmp.org/mp-documents/spec30.pdf>, retrieved 9/2011.
- [29] J. Reinders, *Intel Threading Building Blocks*. O'Reilly, 2007.
- [30] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "Mapreduce and parallel dbms: friends or foes?" *Commun. ACM*, vol. 53, pp. 64–71, 2010.
- [31] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," in *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*. USA: ACM, 2010, pp. 975–986.
- [32] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.
- [33] G. Kooor, "MR-J: A MapReduce framework for multi-core architectures," Ph.D. dissertation, University of Manchester, 2009.
- [34] F. Morvan and A. Hameurlain, "Dynamic memory allocation strategies for parallel query execution," in *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2002, pp. 897–901.
- [35] Z. Zhang, P. Trancoso, and J. Torrellas, "Memory system performance of a database in a shared-memory multiprocessor," 2007. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.1924>, retrieved 10/2012
- [36] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry, "Optimistic intra-transaction parallelism on chip multiprocessors," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 73–84.
- [37] A. N. Wilshut, J. Flokstra, and P. M. G. Apers, "Parallel evaluation of multi-join queries," in *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1995, pp. 115–126.
- [38] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Commun. ACM*, vol. 35, no. 6, pp. 85–98, 1992.
- [39] J. Aguilar-Saborit, V. Munteş-Mulero, C. Zuzarte, A. Zubiri, and J.-L. Larriba-Pey, "Dynamic out of core join processing in symmetric multiprocessors," in *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 28–35.
- [40] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: fast join implementation on modern multi-core cpus," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, 2009.
- [41] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye, "Automatic contention detection and amelioration for data-intensive operations," in *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*. New York, NY, USA: ACM, 2010, pp. 483–494.
- [42] "Billion triple challenge." [Online]. Available: <http://challenge.semanticweb.org>, retrieved 10/2012

Chapter 6.

Highly Scalable Sort-Merge Join Algorithm for RDF Querying

Zbyněk Falt, Miroslav Čermák, Filip Zavoral

Proceedings of the International Conference on Data Technologies and Applications,
SciTePress, pp. 293-300, 2013

DATA 2013

2nd INTERNATIONAL CONFERENCE ON
DATA MANAGEMENT TECHNOLOGIES AND APPLICATIONS

Proceedings

REYKJAVÍK, ICELAND
29 - 31 JULY, 2013

SPONSORED BY:

INSTICC

CO-ORGANIZED BY:



Highly Scalable Sort-merge Join Algorithm for RDF Querying

Zbyněk Falt, Miroslav Čermák and Filip Zavoral

Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
{falt, cermak, zavoral}@ksi.mff.cuni.cz

Keywords: Merge Join, Parallel, Bobox, RDF.

Abstract: In this paper, we introduce a highly scalable sort-merge join algorithm for RDF databases. The algorithm is designed especially for streaming systems; besides task and data parallelism, it also tries to exploit the pipeline parallelism in order to increase its scalability. Additionally, we focused on handling skewed data correctly and efficiently; the algorithm scales well regardless of the data distribution.

1 INTRODUCTION

Join is one of the most important database operation. The overall performance of data evaluation engines depends highly on the performance of particular join operations. Since the multiprocessor systems are widely available, there is a need for the parallelization of database operations, especially joins.

In our previous work, we focused on parallelization of SPARQL operations such as filter, nested-loops join, etc. (Cermak et al., 2011; Falt et al., 2012a). In this paper, we complete the portfolio of parallelized SPARQL operations by proposing an efficient algorithm for merge and sort-merge join.

The main target of our research is the area of streaming systems, since they seem to be appropriate for a certain class of data intensive problems (Bednarek et al., 2012b). Streaming systems naturally introduce task, data and pipeline parallelism (Gordon et al., 2006). Therefore, an efficient and scalable algorithm for these systems should take these properties into account.

Our contribution is the introduction of a highly scalable merge and sort-merge join algorithm. The algorithm also deals well with skewed data which may cause load imbalances during the parallel execution (DeWitt et al., 1992). We used SP²Bench (Schmidt et al., 2008) data generator and benchmark to show the behaviour of our algorithm in multiple test scenarios and to compare our RDF engine which uses this algorithm to other modern RDF engines such as Jena (Jena, 2013), Virtuoso (Virtuoso, 2013) and Sesame (Broekstra et al., 2002).

The rest of the paper is organized as follows. Section 2 examines relevant related work on merge joins,

Section 3 shortly describes Bobox framework that is used for a pilot implementation and evaluation of the algorithm. Most important is Section 4 containing a detailed description of the sort-merge join algorithm. Performance evaluation is described in Section 5, and Section 6 concludes the paper.

2 RELATED WORK

Parallel algorithms greatly improve the performance of the relational join in shared-nothing systems (Liu and Rundensteiner, 2005; Schneider and DeWitt, 1989) or shared memory systems (Cieslewicz et al., 2006; Lu et al., 1990).

Liu et al. (Liu and Rundensteiner, 2005) investigated the pipelined parallelism for multi-join queries. In comparison, we focus on exploiting the parallelism within a single join operation. Schneider et al. (Schneider and DeWitt, 1989) evaluated one sort-merge and three hash-based join algorithms in a shared-nothing system. In the presence of data skews, techniques such as bucket tuning (Schneider and DeWitt, 1989) and partition tuning (Hua and Lee, 1991) are used to balance loads among processor nodes.

Family of non-blocking algorithms, i.e. (Ming et al., 2004; Dittrich and Seeger, 2002) is introduced to deal with pipeline processing where blocking behaviour of network traffic makes the traditional join operators unsuitable (Schneider and DeWitt, 1989). The progressive-merge join (PMJ) algorithm (Dittrich and Seeger, 2002; Dittrich et al., 2003) is a non-blocking version of the traditional sort-merge join. For our parallel stream execution, we adopted the idea of producing join results as soon as first sorted data

are available, even when sorting is not yet finished.

(Albutiu et al., 2012) introduced a suite of new massive parallel sort-merge (MPSM) join algorithms based on partial partition-based sorting to avoid a hard-to-parallelize final merge step to create one complete sort order. MPSM are also NUMA¹-affine, as all sorting is carried on local memory partitions and it scales almost linearly with a number of used cores.

One of the specific areas of parallel join computations are semantic frameworks using SPARQL language. In (Groppe and Groppe, 2011) authors proposed parallel algorithms for join computations of SPARQL queries, with main focus on partitioning of the input data.

Although all the above mentioned papers deal with merge join parallelization, none of them focuses on streaming systems and exploiting data, task and pipeline parallelism and data skewness at once.

3 Bobox

Bobox is a parallelization framework which simplifies writing parallel, data intensive programs and serves as a testbed for the development of generic and especially data-oriented parallel algorithms (Falt et al., 2012c; Bednarek et al., 2012a).

It provides a run-time environment which is used to execute a non-linear pipeline (we denote it as the *execution plan*) in parallel. The execution plan consists of computational units (we denote them as the *boxes*) which are connected together by directed edges. The task of each box is to receive data from its incoming edges (i.e. from its *inputs*) and to send the resulting data to its outgoing edges (i.e. to its *outputs*). The user provides the execution units and the execution plan (i.e. the implementation of boxes and their mutual connections) and passes it to the framework which is responsible for the evaluation of the plan.

The only communication between boxes is done by sending *envelopes* (communication units containing data) along their outgoing edges. Each envelope consists of several columns and each column contains a certain number of data items. The data type of items in one column must be the same in all envelopes transferred along one particular edge; however, different columns in one envelope may have different data types. The data types of these columns are defined by the execution plan. Additionally, all columns in one envelope must have the same length; therefore, we can consider envelopes to be sequences of tuples.

¹Non-Uniform Memory Access

The total number of tuples in an envelope is chosen according to the size of cache memories in the system. Therefore, the communication may take place completely in cache memory. This increases the efficiency of processing of incoming envelopes by a box.

In addition to data envelopes, Bobox distinguishes so called poisoned envelopes. These envelopes do not contain any data and they just indicate the end of a stream.

Currently, only shared-memory architectures are supported; therefore, only shared pointers to the envelopes are transferred. This speeds up operations such as broadcast box (i.e., the box which resends its input to its outputs) significantly since they do not have to access data stored in envelopes.

Although the body of boxes must be strictly single-threaded, Bobox introduces three types of parallelism:

1. Task parallelism - independent streams are processed in parallel.
2. Pipeline parallelism - the producer of a stream runs in parallel with its consumer.
3. Data parallelism - independent parts of one streams are processed in parallel.

The first two types of parallelism are exploited implicitly during the evaluation of a plan. Therefore, even an application which does not contain any explicit parallelism may benefit from multiple processors in the system. Data parallelism must be explicitly stated in the execution plan by the user; however, it is still much easier to modify the execution plan than to write the parallel code by hand.

Due to the Bobox properties and especially its suitability for pipelined stream data processing we used the Bobox platform for a pilot implementation of the SPARQL processing engine.

4 ALGORITHMS

Contemporary merge join algorithms mentioned in Section 2 do not fit well into the streaming model of computation (Gordon et al., 2006). Therefore, we developed an algorithm which takes into account task, data and pipeline parallelism. The main idea of the algorithm is splitting the input streams into many smaller parts which can be processed in parallel.

The sort-merge join consists of two independent phases – sorting phase that sorts the input stream by join attributes and joining phase. We have utilized the highly scalable implementation of a stream sorting algorithm (Falt et al., 2012b); it is briefly described in Section 4.2

4.1 Merge Join

Merge join in general has two inputs – left and right. It assumes that both inputs are sorted by the join attribute in an ascending order. It reads its inputs and finds sequences of the same values of join attributes in the left and right input and then performs the cross product of these sequences. The pseudocode of the standard implementation of merge join is as follows:

```

while left.has_next  $\wedge$  right.has_next do
  left_tuple  $\leftarrow$  left.current
  right_tuple  $\leftarrow$  right.current
  if left_tuple = right_tuple then
    append left_tuple to left_seq
    left.move_next()
    while left.has_next  $\wedge$  left.current=left_tuple do
      append left.current to left_seq
      left.move_next()
    end while
    append right_tuple to right_seq
    right.move_next()
    while right.has_next $\wedge$ right.current=right_tuple do
      append right.current to right_seq
      right.move_next()
    end while
    output cross product(left_seq, right_seq)
  else if left_tuple < right_tuple then
    left.move_next()
  else
    right.move_next()
  end if
end while

```

If we take any value V of the join attribute, then all tuples less than V from both inputs can be processed independently on the tuples which are greater or equal to V . A common approach to merge join parallelization is splitting the inputs into multiple parts by $P - 1$ values V_i and process them in parallel in P worker threads (Groppe and Groppe, 2011).

However, there are two problems with the selection of appropriate values V_i :

1. The inputs of the join are data streams; therefore, we do not know how many input tuples are there until we receive all of them. Because of the same reason, we do not know the distribution of the input data in advance. Therefore, we cannot easily select V_i in order that the resulting parts have approximately the same size.
2. The distribution of data could be very non-uniform (Li et al., 2002); therefore, it might be impossible to utilize worker threads uniformly.

For the sake of simplicity, we first describe a simplified algorithm for joining inputs without duplicated join attribute values in Section 4.1.1. Then we extend the algorithm to take duplicities into account in Section 4.1.2.

4.1.1 Parallel Merge Join without Duplicities

In this section, we describe the algorithm which assumes that the input streams do not contain duplicated join attributes. The execution plan of this algorithm is depicted in Figure 1.

The algorithm makes use of the fact that the streams are represented as a flow of envelopes. The task of *preprocess* box is to transform the flow of input envelopes into the flow of pairs of envelopes. The tuples in these pairs can be joined independently (i.e., in parallel). *Dispatch* boxes dispatch these pairs among *join* boxes which perform the operation. When *join* box receives a pair of envelopes, it joins them and creates the substream of their results. Therefore, the outputs of *join* boxes are sequences of such substreams which subsequently should be consolidated in a round robin manner by *consolidate* box.

Now, we describe the idea and the algorithm of the *preprocess* box. Consider the first envelope *left_env* from the left input and the first envelope *right_env* from the right input. Denote the last tuple (the highest value) in *left_env* as *last_left* and the last tuple in *right_env* as *last_right*.

Now, one of these three cases occurs:

1. *last_left* is greater than *last_right*. In this case, we can split *left_env* into two parts. The first part contains tuples which are less or equal to *last_right* and the second part contains the rest. Now, the first part of *left_env* can be joined with the *right_env*.
2. *last_left* is less than *last_right*. In this case, we can do analogous operation as in the former case.
3. *last_left* is equal to *last_right*. This means, that the whole *left_env* and the whole *right_env* might be joined together.

The pseudocode of *preprocess* box is as follows:

```

left_env  $\leftarrow$  next envelope from left input
right_env  $\leftarrow$  next envelope from right input
while left_env  $\neq$  NIL  $\wedge$  right_env  $\neq$  NIL do
  last_left  $\leftarrow$  left_env[left_env.size - 1]
  last_right  $\leftarrow$  right_env[right_env.size - 1]
  if last_left > last_right then
    split left_env to left_first and left_second
    send right_env to the right output
    send left_first to the left output
    left_env  $\leftarrow$  left_second
    right_env  $\leftarrow$  next envelope from right input
  else if last_left < last_right then
    split right_env to right_first and right_second
    send right_first to the right output
    send left_env to the left output
    left_env  $\leftarrow$  next envelope from left input
    right_env  $\leftarrow$  right_second
  else
    send right_env to the right output
    send left_env to the left output

```

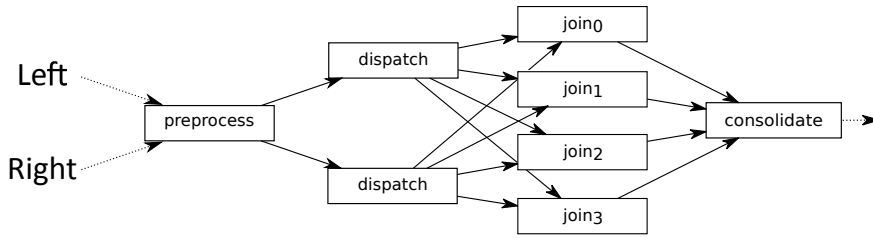


Figure 1: Execution plan of parallel merge join.

```

left_env ← next envelope from left input
right_env ← next envelope from right input
end if
end while
close the right output
close the left output

```

The boxes *preprocess*, *dispatch* and *consolidate* might seem to be bottlenecks of the algorithm. *Dispatch* and *consolidate* do not access data in envelopes, they just forward them from the input to the output. Since the envelope typically contains hundreds or thousands of tuples, these two boxes work in several orders of magnitude faster than *join* box.

On the other hand, *preprocess* box accesses data in the envelope since it has to find the position where to split the envelope. This can be done by a binary search which has time complexity $O(\log(L))$ where L is the number of tuples in the envelope. However, it does not access all tuples in the envelope; therefore, it is still much faster than *join* box.

4.1.2 Join with Duplicities

Without duplicities, *preprocess* box is able to generate pairs of envelopes which can be processed independently. However, the possibility of their existence complicates the algorithm. Consider a situation depicted in Figure 2.

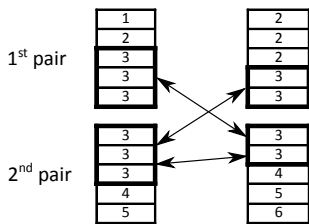


Figure 2: Duplicities of join attributes.

If *join* box receives the pair number 2, it needs to process also the pair number 1. The reason is, that it has to perform cross products of parts which are denoted in the figure.

Therefore, *join* boxes have to receive all pairs of envelopes for the case when there are sequences of the same tuples across multiple envelopes. This compli-

cates the algorithm of *join* box, since each join has to keep track of such sequences. When we processed an envelope (from either input), there is a possibility that its last tuple is a part of such sequence. Therefore, we have to keep already processed envelopes for the case they will be needed in the future. When the last tuple of the envelope changes, the new sequence begins and we can drop all stored envelopes except the last one.

The execution plan for the algorithm is the same as in the previous case, the only difference is that *dispatch* box does not forward its input envelopes in a round robin manner but it broadcasts them to all its outputs. Since a box receives and sends only shared pointers to the envelopes, the overhead of the broadcast operation is negligible in comparison to the join operation and therefore it does not limit the scalability.

Because of this modification, all boxes receive the same envelopes. Therefore, the algorithm should distinguish among them so that they generate the output in the same manner as in Section 4.1.1. Each *join* box gets its own unique index $P_i, 0 \leq P_i < P$. If we denote each pair of envelopes sequentially by non-negative integers j ; then *join* box with index P_i processes such pairs j for which it holds $j \bmod P = P_i$. This concept of parallelization is described in (Falt et al., 2012a) in more detail.

The complete pseudocode of *join* box is as follows:

```

left_env ← next envelope from left input
right_env ← next envelope from right input
j ← 0
left_seq ← empty
right_seq ← empty
while left_env ≠ NIL ∧ right_env ≠ NIL do
  if j mod P = P_i then
    do the join of left_env and right_env
  end if
  j ← j + 1
  if left_env.size > 0 then
    last_Left ← left_env[left_env.size - 1]
    if last_Left ≠ left_seq then
      left_seq ← last_Left
      drop all left envelopes except left_env
    else
      store left_env
    end if
  end if
end while

```

```

end if
if right_env.size > 0 then
  last_right ← right_env[right_env.size - 1]
  if last_right ≠ right_seq then
    right_seq ← last_right
    drop all right envelopes except right_env
  else
    store right_env
  end if
end if
left_env ← next envelope from left input
right_env ← next envelope from right input
end while

```

The performance evaluation in Section 5.1.3 shows that such concept of parallelization allows better scalability than other contemporary solutions.

4.2 Sort

If one or both input streams need to be sorted, we use the approach based on algorithm described in (Falt et al., 2012b). Basically, the sorting of the stream is divided into three phases:

1. Splitting the input stream into several equal sized substreams,
2. Sorting of the substreams in parallel,
3. Merging of the sorted substreams in parallel.

The algorithm scales very well; moreover, it starts to produce its output very shortly after the reception of the last tuple. Therefore, the consecutive merge join can start working as soon as possible which enables pipeline processing and increases scalability.

However, the memory becomes indispensable bottleneck when sorting tuples instead of scalars, since a tuple typically contains multiple items. Thus, the sorting of tuples needs more memory accesses especially when sorting in parallel.

Therefore, we replaced the merge algorithm (used in the second and the third phase) by a merge algorithm used in Funnelsort (Frigo et al., 1999). We used the implementation available on (Vinther, 2006). This algorithm utilizes cache memories as much as possible in order to decrease the number of accesses to the main memory. According to our experiments, this algorithm speeds up the merging phase by 20–30%.

5 EVALUATION

Since one of the main goals is efficient evaluation of SPARQL (Prud'hommeaux and Seaborne, 2008) queries, we used a standardized SP²Bench benchmark for the performance evaluation of our algorithm in a parallel environment. Moreover, in order to show

skewness resistance of our algorithm, we used additional synthetic queries.

All experiments were performed on a server running Redhat 6.0 Linux; server configuration is 2x Intel Xeon E5310, 1.60Ghz (L1: 32kB+32kB L2: 4MB shared) and 8GB RAM. Each processor has 4 cores; therefore, we used 8 worker threads for the evaluation of queries. The server was dedicated specially to the testing; no other applications were running during measurements.

5.1 Scalability of the Algorithm

In this set of experiments we examined the behaviour of the join algorithm in multiple scenarios. We used 5M dataset of SP²Bench.

We measured the performance of the queries in multiple settings. The setting ST uses just one worker thread and the execution plan uses operations without any intraoperator parallelization (i.e., joining and sorting was performed by one box). The setting MT1 uses also one worker thread; however, the execution plan uses operations with intraoperator parallelization (we use 8 worker boxes both for joining and sorting). The purpose of this setting is to show the overhead caused by the parallelization. The MT2, MT4 and MT8 are analogous to the setting MT1; however, they use 2, 4 and 8 worker threads respectively. These settings show the scalability of the algorithm.

5.1.1 Scalability of the Merge Join

The first experiment shows the scalability of the merge join algorithm when its inputs contain long sequences of tuples with the same join attribute (i.e., the join produces high number of tuples) and with the join condition with very high selectivity (i.e., the number of resulting tuples is relatively low). Since both inputs of the join are sorted by join attribute, this algorithm shows only the scalability of merge join and does not include eventual sorting.

For this experiment, we used this query E1:

```

SELECT ?article1 ?article2
WHERE {
  ?article1 swrc:journal ?journal .
  ?article2 swrc:journal ?journal
  FILTER (STR(?article1) = STR(?article2))
}

```

The query generates all pairs of articles which were published in the same journal and then selects the pairs which have the same URI (in fact, it returns all articles in the dataset). The execution plan of the query is depicted in Figure 3. The numbers in the bottom of boxes denote the numbers of tuples produced by the them.

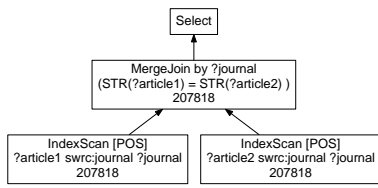


Figure 3: Query E1 execution plan.

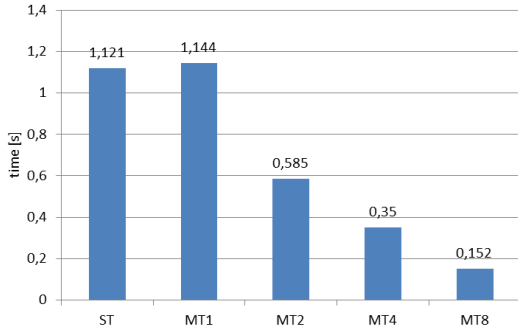


Figure 4: Results for query E1.

The settings MT1 is slightly slower than ST, since the query plan contains in fact more boxes (see Figure 1) which causes higher overhead with their management. Moreover, the *preprocess* box does useless job in this setting. However, when increasing the number of worker threads, the algorithm scales almost linearly with the number of threads.

5.1.2 Scalability of the Sort-merge Join

The scalability of the sort-merge join is shown in the following experiment. In the contrast to the previous experiment, the inputs of merge joins (the second phase of sort-merge join) need to be sorted.

For this experiment, we used this query E2:

```
SELECT ?article1 ?article2
WHERE {
  ?article1 swrc:journal ?journal .
  ?article2 swrc:journal ?journal .
  ?article1 dc:title ?title1 .
  ?article2 dc:title ?title2
  FILTER(?title1 < ?title2)
}
```

This plan generates a large number of tuples which have to be sorted before they can be finally joined with the second input. The execution plan is depicted in Figure 5.

We measured the runtime in the same settings as the previous experiment and the results are shown in Figure 6.

In this experiment, the difference between ST and MT1 setting is bigger than in the previous experiment. This is caused by the fact that the parallel sort algorithm has some overhead (see (Falt et al., 2012b)

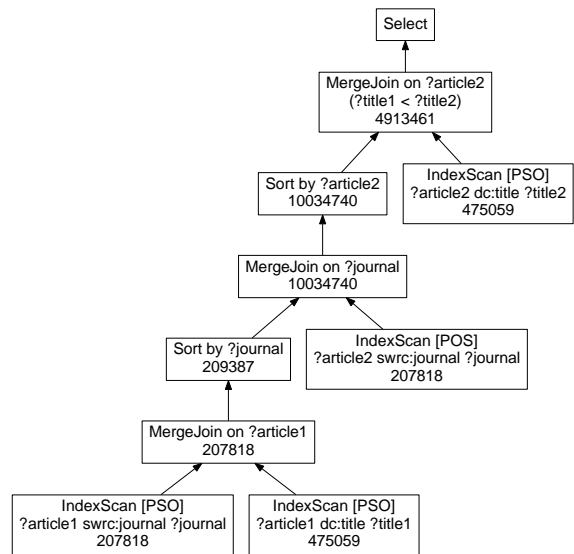


Figure 5: Query E2 execution plan.

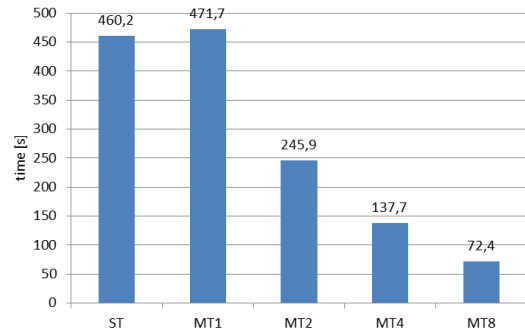


Figure 6: Results for query E2.

for more information). However, the more worker threads are used, the bigger speed-up we gain. The scalability is not as linear as in the previous experiment since the number of memory accesses during sorting is much higher than during merging. Therefore, the memory becomes the bottleneck with higher number of threads.

5.1.3 Data-skewness Resistance

To show the resistance of the algorithm to the non-uniform distribution of data, we used this query E3:

```
SELECT ?artcl1 ?artcl2 ?artcl3 ?artcl4
WHERE {
  ?artcl1 rdf:type bench:PhDThesis .
  ?artcl1 rdf:type ?type .
  ?artcl2 rdf:type ?type .
  ?artcl3 rdf:type ?type .
  ?artcl4 rdf:type ?type
}
```

The execution plan for this query is shown in Figure 7. The variable *?type* has just one value

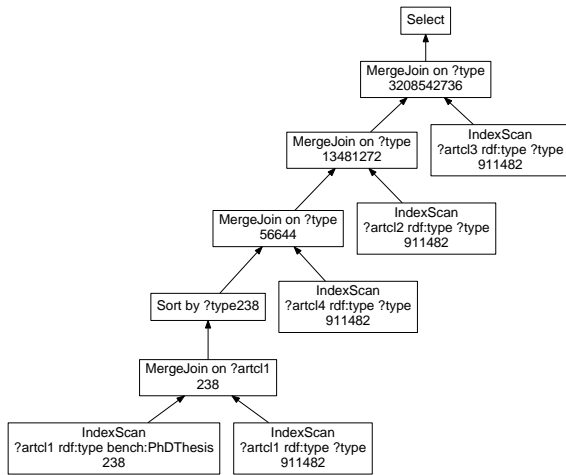


Figure 7: Query E3 execution plan.

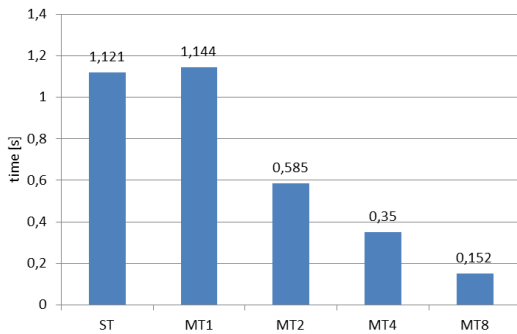


Figure 8: Results for query E3.

(bench:PhDThesis); therefore, all joins on that variable are impossible to be parallelized by partitioning their inputs. Despite this fact, our algorithm according to the results (Figure 8) scales very well and almost linearly.

5.2 Comparison to other Engines

The last set of experiments compares the Bobox SPARQL engine which uses new sort-merge join algorithm to other mainstream SPARQL engines, such as Sesame v2.0 (Broekstra et al., 2002), Jena v2.7.4 with TDB v0.9.4 (Jena, 2013) and Virtuoso v6.1.6.3127-multithreaded (Virtuoso, 2013). They follow client-server architecture and we provide a sum of the times of client and server processes. The Bobox engine was compiled as a single application. We omitted the time spent by loading dataset to be comparable with a server that has the data already prepared.

We evaluated queries multiple times over datasets 5M triples and we provide the average times. Each test run was also limited to 30 minutes (the same time-

out as in the original SP²Bench paper). All data were stored in-memory, as our primary interest is to compare the basic performance of the approaches rather than caching etc.

Table 1: Results of SP²Bench benchmark.

	ST	MT8	Jena	Virtuoso	Sesame
Q1	0.01	0.01	0.01	0.00	0.54
Q2	1.32	0.39	242.80	39.03	16.11
Q3a	0.01	0.01	20.84	7.00	2.09
Q3b	0.00	0.00	1.89	0.04	0.54
Q3c	0.00	0.00	1.31	0.03	0.55
Q4	43.69	6.48	TO	1740.84	TO
Q5a	3.08	0.77	TO	30.89	TO
Q5b	1.23	0.23	38.97	28.03	11.02
Q6	TO	1119.3	TO	61.53	TO
Q7	54.89	6.99	TO	23.06	TO
Q8	6.73	1.21	0.26	0.24	17.37
Q9	3.19	0.50	12.25	16.56	7.58
Q10	0.00	0.00	0.30	0.03	1.28
Q11	0.42	0.12	1.50	3.12	0.53

The results are shown in Table 1 (TO means timeout, i.e., 30 min). Queries Q1, Q3a, Q3b, Q3c and Q10 operate on few tuples and they all fit into several envelopes. Therefore, the parallelization is insignificant. However, the important feature is that despite the more complex execution plans in settings MT8, the run time is not higher than for non-parallelized version.

Queries Q8 and Q6 are slower than other frameworks, since our SPARQL compiler does not perform some optimizations useful for these queries.

The most important result is that queries Q2, Q3a, Q3b, Q3c, Q4, Q5a, Q5b, Q9 and Q11 significantly outperform other engines. All these queries benefit from extensive parallelization; therefore, much larger data can be processed in reasonable time. The significant slowdown of Virtuoso in Q4 is probably caused by extensive swapping, since the result set is too big.

6 CONCLUSIONS AND FUTURE WORK

In the paper, we proposed a new method of parallelization of sort-merge join operation for RDF data. Such algorithm is especially designed for streaming systems; moreover, the algorithm behaves well also with skewed data. The pilot implementation within the Bobox SPARQL engine significantly outperforms other RDF engines such as Jena, Virtuoso and Sesame in all relevant queries.

In our future research we want to focus on further optimizations such as the influence of granularity of data stream units (envelopes) on overall perfor-

mance. Additionally, the other research direction is to use these ideas for other than RDF processing, e.g., SQL.

ACKNOWLEDGEMENTS

The authors would like to thank the GACR 103/13/08195, GAUK 277911, GAUK 472313, and SVV-2013-267312 which supported this paper.

REFERENCES

- Albutiu, M.-C., Kemper, A., and Neumann, T. (2012). Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075.
- Bednarek, D., Dokulil, J., Yaghob, J., and Zavoral, F. (2012a). Bobox: Parallelization Framework for Data Processing. In *Advances in Information Technology and Applied Computing*.
- Bednarek, D., Dokulil, J., Yaghob, J., and Zavoral, F. (2012b). Data-Flow Awareness in Parallel Data Processing. In *6th International Symposium on Intelligent Distributed Computing - IDC 2012*. Springer-Verlag.
- Broekstra, J., Kampman, A., and Harmelen, F. v. (2002). Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 54–68, London, UK. Springer-Verlag.
- Cermak, M., Dokulil, J., Falt, Z., and Zavoral, F. (2011). SPARQL Query Processing Using Bobox Framework. In *SEMAPRO 2011, The Fifth International Conference on Advances in Semantic Processing*, pages 104–109. IARIA.
- Cieslewicz, J., Berry, J., Hendrickson, B., and Ross, K. A. (2006). Realizing parallelism in database operations: insights from a massively multithreaded architecture. In *Proceedings of the 2nd international workshop on Data management on new hardware*, DaMoN '06, New York, NY, USA. ACM.
- DeWitt, D. J., Naughton, J. F., Schneider, D. A., and Shadri, S. (1992). Practical skew handling in parallel joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 27–40, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Dittrich, J.-P. and Seeger, B. (2002). Progressive merge join: A generic and non-blocking sort-based join algorithm. In *VLDB*, pages 299–310.
- Dittrich, J.-P., Seeger, B., Taylor, D. S., and Widmayer, P. (2003). On producing join results early. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '03, pages 134–142, New York, NY, USA. ACM.
- Falt, Z., Bednarek, D., Cermak, M., and Zavoral, F. (2012a). On Parallel Evaluation of SPARQL Queries. In *DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 97–102. IARIA.
- Falt, Z., Bulanek, J., and Yaghob, J. (2012b). On Parallel Sorting of Data Streams. In *ADBIS 2012 - 16th East European Conference in Advances in Databases and Information Systems*.
- Falt, Z., Cermak, M., Dokulil, J., and Zavoral, F. (2012c). Parallel sparql query processing using bobox. *International Journal On Advances in Intelligent Systems*, 5(3 and 4):302–314.
- Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (1999). Cache-Oblivious Algorithms. In *FOCS*, pages 285–298.
- Gordon, M. I., Thies, W., and Amarasinghe, S. (2006). Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, 34(5):151–162.
- Groppe, J. and Groppe, S. (2011). Parallelizing join computations of sparql queries for large semantic web databases. In *Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11*, pages 1681–1686, New York, NY, USA. ACM.
- Hua, K. A. and Lee, C. (1991). Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 525–535, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Jena (2013). Jena – a semantic web framework for Java. Available at: <http://jena.apache.org/>, [Online; Accessed February 4, 2013].
- Li, W., Gao, D., and Snodgrass, R. T. (2002). Skew handling techniques in sort-merge join. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 169–180. ACM.
- Liu, B. and Rundensteiner, E. A. (2005). Revisiting pipelined parallelism in multi-join query processing. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 829–840. VLDB Endowment.
- Lu, H., Tan, K.-L., and Sahn, M.-C. (1990). Hash-based join algorithms for multiprocessor computers with shared memory. In *Proceedings of the sixteenth international conference on Very large databases*, pages 198–209, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Ming, M. M., Lu, M., and Aref, W. G. (2004). Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *In ICDE*, pages 251–263.
- Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL Query Language for RDF. W3C Recommendation.
- Schmidt, M., Hornung, T., Lausen, G., and Pinkel, C. (2008). Sp2bench: A sparql performance benchmark. *CoRR*, abs/0806.4627.
- Schneider, D. A. and DeWitt, D. J. (1989). A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *SIGMOD Rec.*, 18(2):110–121.
- Vinther, K. (2006). The Funnelsort Project. Available at: <http://kristoffer.vinther.name/projects/funnelsort/>, [Online; Accessed February 4, 2013].
- Virtuoso (2013). Virtuoso data server. Available at: <http://virtuoso.openlinksw.com>, [Online; Accessed February 4, 2013].

Chapter 7.

Resistance of Trust Management Systems against Malicious Collectives

Miroslav Novotný, Filip Zavoral

Proceedings of 2nd International Conference on Context-Aware Systems and Applications,
Springer Verlag, pp. 67-76, 2014

Phan Cong Vinh
Vangalur Alagar
Emil Vassev
Ashish Khare (Eds.)



128

Context-Aware Systems and Applications

Second International Conference, ICCASA 2013
Phu Quoc Island, Vietnam, November 25–26, 2013
Revised Selected Papers



 Springer

Resistance of Trust Management Systems Against Malicious Collectives

Miroslav Novotný and Filip Zavoral^(✉)

Faculty of Mathematics and Physics, Charles University in Prague,
118 00 Prague, Czech Republic
{novotny, zavoral}@ksi.mff.cuni.cz

Abstract. Malicious peers in Peer-to-peer networks can develop sophisticated strategies to bypass existing security mechanisms. The effectiveness of contemporary trust management systems is usually tested only against simple malicious strategies. In this paper, we propose a simulation framework for evaluation of resistance of trust management systems against different malicious strategies. We present results of five TMS that represent main contemporary approaches; the results indicate that most of the traditional trust managements are vulnerable to sophisticated malicious strategies.

Keywords: Trust management · Peer to peer networks

1 Introduction

One of the promising architectures of large-scale distributed systems is based on peer to peer architecture (P2P). However, providing proper protection to such systems is tricky. The P2P applications have to deal with treacherous peers that try to deliberately subvert their operation. The peers have to trust the remote party to work correctly. The process of getting this trust is, however, far from trivial.

Many trust management systems (TMS) have been developed to deal with treacherous peers in P2P networks. The main idea of these systems is sharing experience between honest peers and building reputations. Nevertheless, the group of cooperating malicious peers is often able to bypass their security mechanisms and cause a great deal of harm. The malicious collectives represent the main reasons why managing trust represents the biggest challenge in the current P2P networks.

In this paper, we investigate several TMSs and use the simple taxonomy to organize their major approaches. Using the simulation framework called P2PTrustSim we investigate different strategies used by malicious peers. Beside traditional strategies, we propose new, more sophisticated strategies and test them against five trust management systems. These systems have been chosen as the representatives of major approaches. Our goal was to verify the effectiveness of various TMSs under sophisticated malicious strategies. We have chosen five contemporary TMS: EigenTrust [1], PeerTrust [2], H-Trust [3], WTR [4], and BubbleTrust [5]. These TMS represent main contemporary approaches in Trust Management.

2 Malicious Strategies and Evaluation Criteria

In order to facilitate comparison of different TMSs and their behaviour under different malicious strategies we created a simulation framework [6] called P2PTrustSim. We used FreePastry, a modular, open-source implementation of the Pastry [12], P2P structured overlay network. Above the FreePastry, we created the peer simulation layer which implements various peers' behaviour.

2.1 Malicious Strategies

Most of the TMSs work well against straightforward malicious activities. However, the malicious peers can develop strategies to maintain their malicious business. Each peer can operate individually but the biggest threat is the collusion of malicious peers working together.

2.1.1 Individual Malicious Strategies

These strategies do not involve the cooperation between malicious peers.

False Meta-data - Malicious peers can insert false, attractive information into the meta-data describing their bogus resources to increase the demands for them.

Camouflage - The malicious peers that are aware of the presence of the TMS can provide a few honest resources. There can be many variants of this strategy, differing in the ratio of honest and bogus services or the period between changing behaviour. In some literature, the variant of this strategy is called Traitors [7, 8–10].

2.1.2 Collective Malicious Strategies

Malicious peers have a significantly higher chance to succeed if they work in a cooperative manner; this is considered as the biggest treat for P2P applications [11].

Full Collusion - All members of a malicious collective provide bogus resources and create false positive recommendations to all other members of the collective.

Spies - The malicious collective is divided into two groups: spies and malicious. The spies provide honest services to earn a high reputation and simultaneously provide false positive recommendations to the malicious part of the collective.

2.1.3 Newly Proposed Malicious Strategies

We analyzed published TMSs and known malicious tactics carefully and we suggest three new collective malicious strategies. Each strategy is designed for a particular type of TMS and tries to exploit its specific weakness.

Evaluator Collusion - If the TMS assesses credibility of the feedback source according to the truthfulness of its previous feedback, malicious peers can try to trick the TMS by using the services from peers outside the collective and evaluating them correctly. This feedback increases the credibility of malicious peers as recommenders and gives more weight to their feedback towards other members of collective.

Evaluator Spies - This strategy is a combination of Evaluator Collusion and Spies. The spies implement three techniques to maintain a credibility as a feedback source: they provide honest service, they use resources outside the collective and evaluate them correctly, and they create positive recommendations towards other spies.

Malicious Spies - This slight modification of the previous strategy is based on the idea that spies do not require a high reputation as resource providers. They can provide bogus resources and generate negative recommendations between each other. These recommendations are still truthful and should increase their credibility.

2.2 Evaluation Criteria

Each transaction within the framework is categorized on both sides (provider and consumer). The categories distinguish the type of the peer (honest or malicious), on which side of the transaction the peer was (provider or consumer), and the result of the transaction. The ulterior transactions represent honest transactions which malicious peers have to perform to fix their reputation. If no provider is sufficiently trustful, the transaction is refused and counted as ConsumeRefused. The originated peer typically tries to pick different service and repeat the transaction.

Let us suppose that all the malicious peers cooperate within a malicious collective in the network and all transactions from honest peers are honest. Our primary goal is to evaluate the success of each malicious strategy in different TMSs. Therefore, we defined four criteria:

MaliciousSuccessRatio (MSR) is a ratio between bogus transactions provided by malicious peers in the network with TMS and in the network without TMS (DummyTrust). It reflects the contribution of the given TMS and it is defined by the following formula:

$$\text{MaliciousSuccessRatio} = \frac{\text{TotalBogus}_{\text{withTMS}}}{\text{TotalBogus}_{\text{withoutTMS}}}$$

BogusRatio (BR) is a ratio between bogus and all services consumed by the honest peers. It is defined by the following formula:

$$\text{BogusRatio} = \frac{100 * \text{TotalBogus}}{\sum \text{ConsumeHonest} + \text{TotalBogus}}$$

MaliciousCost (MC) monitors the load associated with a malicious strategy. It is a ratio between extra transactions performed by the malicious peers to trick the TMS and the bogus transactions in the network. These extra transactions include faked and ulterior transactions and represent additional overhead for malicious peers which they try to minimize. We defined it by the following formula:

$$\text{MaliciousCost} = \frac{\text{TotalUlterior} + \text{TotalFaked}/2}{\text{TotalBogus}}$$

This metric gives us an idea of how much computational power and network utilization is required for a given malicious strategy.

The last criterion is a **MaliciousBenefit (MB)**. It represents how much beneficial transactions the malicious peers have to perform to pass one malicious service. It is defined by the following formula:

$$\text{MaliciousBenefit} = \frac{\text{TotalUlterior}}{\text{TotalBogus}}$$

The value above 1 means that there is benefit from the malicious collective which is bigger than the damage caused by the collective.

3 Simulation Results

We focused on two problems: the effectiveness of the strategies and the reaction of the TMSs to changes in peers' behaviours. The first problem was studied in a network that contains 200 peers and 80 peers are malicious; 40 % of nodes in the network are malicious, which represents a very dangerous environment. The honest peers wake up every 10 min and use one service from the network. The malicious peers also wake up every 10 min and perform a given number of faked or ulterior transactions. We ran 56 different simulations (7 TMSs each with 8 strategies). Each of the simulations represents 24 h. The data is counted in the last hour of the simulations when the TMSs are stabilized. Each simulation was repeated 20 times and average values are taken. The variation of results is expressed in the form of a relative standard deviation (RSD). The size of the network was designed with regards to simulation possibilities of the FreePastry and the load produced by our simulation. The results of other series of tests with the different settings were almost identical.

We set similar parameters for all TMSs. The most important parameter is the history period which determines how long the peers remember the information about previous transactions. We set this parameter to 30 cycles (5 h, in order to have a history period appropriate to the total simulation time) in all TMSs. The EigenTrust is not able to work correctly without pre-trusted peers, so we had to set 10 % honest peers as pre-trusted. Therefore, the EigenTrust has an advantage over other TMSs. Also, the numbers of ulterior and faked transactions are the same for all malicious strategies which use them.

3.1 Representative TMSs

The first simulations were performed in the network without TMS (DummyTrust) and in the network with the simplest version of TMS (SimpleTrust). We focused on the number of bogus transactions; these values will be used as a base for calculation of MaliciousSuccessRatio for other TMSs. The results are shown in Table 1.

As expected, the False Meta-data is the only useful strategy in DummyTrust. Other malicious mechanisms are useless or even counterproductive. The strategies Malicious Individual, Full Collusion, Evaluator Collusion and Malicious Spies have

Table 1. Number of bogus transactions in DummyTrust and SimpleTrust.

Strategy	DummyTrust		SimpleTrust		
	TotalBogus	RSD (%)	TotalBogus	RSD	Diff.
Simple Malicious Individual	262.20	8.95	247.15	6.25	6
Malicious Individual	435.65	3.05	387.70	4.12	11
Camouflage	310.85	4.02	281.20	3.54	10
Full Collusion	430.75	2.84	391.45	2.83	9
Evaluator Collusion	436.90	2.62	388.10	4.40	11
Spies	297.25	4.00	249.20	5.17	16
Evaluator Spies	301.30	4.31	244.80	7.00	19
Malicious Spies	433.65	2.54	386.65	3.17	11

almost the same results. All these strategies use False Meta-data, unlike The Simple Malicious Individual, which reaches fewer bogus transactions. The rest of the malicious strategies sacrifice a part of bogus transactions to circumvent TMSs, however these transactions have no effect in DummyTrust. The biggest variation in results has been measured in Simple Malicious Individual. In this strategy, honest peers completely rely on a random choice of communication partner.

The SimpleTrust has only slightly better results. The biggest improvement was measured in Evaluator Spies and Spies. These strategies are not suitable for simple TMSs. In fact, we have expected a bigger improvement. The limited factor is the size of the history period which was set to 30 cycles in all TMSs. Without cooperation with other peers, the information about peer's maliciousness is lost after 30 cycles and the delay between two transactions towards the same peer can be longer.

3.2 Efficiency Criterion

We measured the criteria described in Sect. 3. The most important of them is the MaliciousSuccessRatio (MSR); the measured values are in Tables 2 and 3 along with average numbers of bogus transactions and standard deviations. The MSR values above the threshold 0.5 are displayed in a bold font. We can see that only the BubbleTrust is resistant against all malicious strategies. There is at least one effective malicious strategy against all other TMSs. The EigenTrust, despite its advantage, is completely vulnerable to Spies and Evaluator Spies. These strategies are even able to perform more bogus transactions than it would be possible in a network without TMS. PeerTrust is resistant against only the simplest malicious strategies, on the other hand, malicious strategies like Evaluator Collusion and Evaluator Spies are 100 % effective. Also H-Trust does not work well, it is completely vulnerable to Evaluator Collusion and Evaluator Spies and the resistance against other strategies is not convincing either. WTR copes very well with individual strategies; especially the Camouflage is ineffective due to the risk factor. But the collective strategies can easily circumvent it. There are noticeable deviations in some malicious strategies. However, none of these deviations influence the MSR value that much that cross the limit 0.5. The next criterion is BogusRatio. Table 4 shows BogusRatio of each malicious strategy in all TMSs. In the worst case scenario, only 29 % of all transactions in the P2P network

Table 2. Malicious Success Ratio in BubbleTrust, EigenTrust and PeerTrust.

Strategy	BubbleTrust			EigenTrust			PeerTrust		
	Total	RSD	MSR	Total	RSD	MSR	Total	RSD	MSR
	Bogus	(%)		Bogus	(%)		Bogus	(%)	
Simple M Individual	6.2	66.2	0.0	64.2	21.6	0.2	17.5	20.3	0.1
Malicious Individual	1.5	89.9	0.0	137.9	15.2	0.3	0.0	0.0	0.0
Camouflage	1.55	84.96	0.00	87.85	24.82	0.28	200.60	6.84	0.65
Full Collusion	58.25	13.13	0.14	0.00	0.00	0.00	426.90	3.55	0.99
Evaluator Collusion	109.2	10.17	0.25	0.00	0.00	0.00	440.05	2.70	1.01
Spies	21.5	23.93	0.07	323.45	3.83	1.09	282.25	4.53	0.95
Evaluator Spies	48.3	11.44	0.16	295.50	28.08	0.98	300.00	4.65	1.00
Malicious Spies	53.5	11.21	0.12	0.55	–	0.00	297.95	4.87	0.69

Table 3. Malicious Success Ratio in HTrust and WTR.

Strategy	HTrust			WTR		
	Total	RSD (%)	MSR	Total	RSD (%)	MSR
	Bogus			Bogus		
Simple Malicious Individual	54.00	20.70	0.21	0.00	0.00	0.00
Malicious Individual	142.15	8.69	0.33	0.00	0.00	0.00
Camouflage	56.30	15.94	0.18	0.00	0.00	0.00
Full Collusion	138.05	8.23	0.32	435.45	2.26	1.01
Evaluator Collusion	411.00	4.30	0.94	436.70	3.73	1.00
Spies	108.10	8.74	0.36	293.30	5.57	0.99
Evaluator Spies	296.60	3.84	0.98	302.65	4.39	1.00
Malicious Spies	299.55	4.04	0.69	304.40	3.86	0.70

Table 4. BogusRatio for different malicious strategies and TMSs.

Strategy	EigenTrust (%)	H-Trust (%)	PeerTrust (%)	WTR (%)	BubbleTrust (%)
Simple M Individual	13	11	4	0	2
Malicious Individual	34	35	0	0	1
Camouflage	21	13	38	0	0
Full Collusion	0	34	72	73	18
Evaluator Collusion	0	70	73	73	29
Spies	55	22	48	49	5
Evaluator Spies	63	50	50	50	11
Malicious Spies	1	56	57	57	16

with the BubbleTrust can be bogus. Other TMS tolerate 63 % (EigenTrust), 70 % (H-Trust), 73 % (PeerTrust and WTR) bogus transactions.

Table 5 shows MaliciousCost of malicious strategies which use ulterior or faked transactions. Other strategies (Simple Malicious Individual and Malicious Individual) have no additional cost. MaliciousCost of the strategies with no measurable MSR is infinite and the cells contain 'N/A'.

Table 5. MaliciousCost for different malicious strategies and TMSs.

Strategy	EigenTrust	H-Trust	PeerTrust	WTR	BubbleTrust
Camouflage	0.17	0.19	0.09	N/A	0.16
Full Collusion	N/A	8.58	2.78	2.72	20.31
Evaluator Collusion	N/A	9.67	9.06	9.12	36.47
Spies	1.95	5.79	2.23	2.13	29.87
Evaluator Spies	5.87	5.74	5.72	5.69	36.23
Malicious Spies	N/A	5.65	5.68	5.56	31.63

Table 6. MaliciousBenefit for different malicious strategies and TMSs.

Strategy	EigenTrust	H-Trust	PeerTrust	WTR	BubbleTrust
Camouflage	0.17	0.19	0.09	N/A	0.16
Evaluator Collusion	N/A	6.79	6.36	6.41	25.64
Spies	0.10	0.24	0.11	0.09	1.96
Evaluator Spies	2.85	2.73	2.75	2.74	17.73
Malicious Spies	N/A	2.67	2.68	2.63	14.95

The attacker most likely uses a strategy which has the best price/performance ratio. For instance, in the PeerTrust the most successful strategy is Evaluator Collusion but it is very expensive (above 9), better choice is Full Collusion with success ratio 0.99 and cost only 2.78. The Camouflage strategy is relatively efficient; although it has low a success ratio in the most TMSs, it is compensated by its very low price. In the BubbleTrust, all strategies have cost above 20 (except Camouflage) and the most expensive strategy (Evaluator Collusion) has almost 37. This is significantly higher value than the other TMSs have.

Table 6 shows MaliciousBenefit of malicious strategies which have some beneficial transactions. Again, MaliciousBenefit of the strategies with no measurable MSR is infinite and the cells contain 'N/A'.

The strategies like Evaluator Collusion, Evaluator Spies and Malicious Spies have always more beneficial transactions than bogus ones. Strictly speaking, the designation of the collective as malicious is no longer suitable. The attackers, whose primary goal is to destroy the network functionality for other peers, probably would not choose malicious strategy with a high MaliciousBenefit. But attackers desired to spread their malicious services at any cost do not bother with MaliciousBenefit.

3.3 Influence of Simulation Settings

We have tried different simulation settings. We have adjusted the number of nodes in the network while preserving the ratio of malicious nodes. We have made the following observation: increasing the number of nodes does not affect the MaliciousSuccess-Ratio. The reason is that each TMS can handle only a limited number of nodes in the calculation of ratings. A similar limitation can be found in all TMSs. The information from nodes which are very distant in a trust chain is negligible. On the other hand, the results change if we decrease the number of nodes. This change can be in both

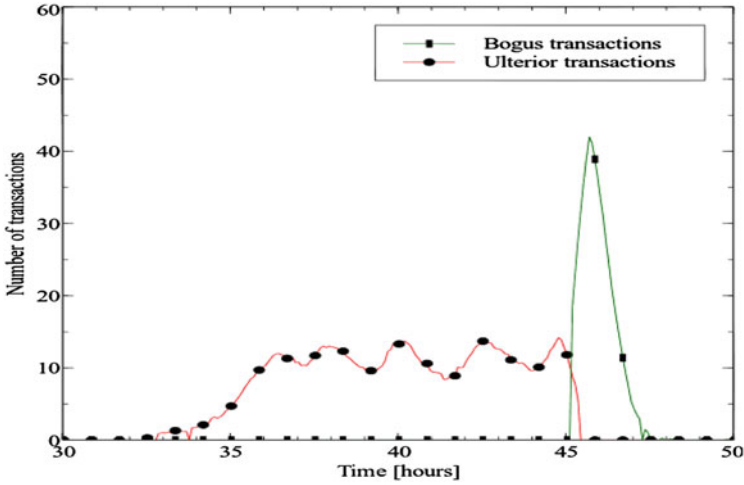


Fig. 1. Rehabilitation after treason in BubbleTrust.

directions dependent on the TMS and the malicious strategy. In this case the TMS has to rely on information from a smaller number of nodes than it expects (Fig. 1).

Next, we have altered the ratio of malicious nodes. Figure 2 shows the results for BubbleTrust. As we can see, the malicious success increases with the ratio of malicious nodes in the network. BubbleTrust resists relatively well even in the network with more than 50 % of malicious nodes. In our tests we stayed at 40 % because it is very unlikely that the overlay network beneath the P2P application can handle the situation in which half of the peers are malicious. The defence techniques described in

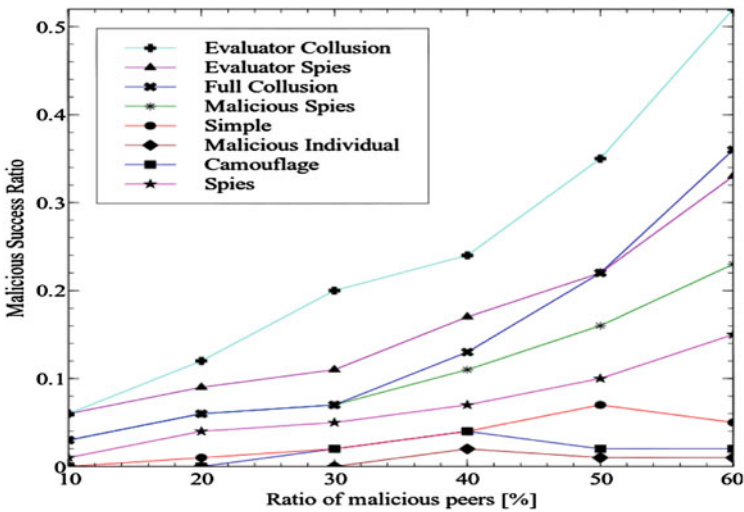


Fig. 2. Ratio of malicious peers on Malicious Success Ratio in BubbleTrust.

2.1 assume that only a small fraction of nodes is malicious. In fact, 40 % already causes big problems.

3.4 Result Summary

H-index calculation used in H-Trust proved to be vulnerable to traitors. It takes too long to detect traitors and malicious peers are rehabilitated too quickly. The system WTR permits the highest number of bogus transactions from all tested TMSs, but it is followed closely by PeerTrust and HTrust. EigenTrust has better results than H-Trust, WTR and PeerTrust but it has advantage in the form of pre-trusted peers.

Our tests proved that it is very difficult to resist against the sophisticated malicious techniques. Especially the calculation of the evaluator rating is susceptible to rigging. The previously published TMSs do not pay as much attention to the evaluator rating as they pay to the provider rating. This must be changed if the TMS should be resistant against the Evaluator Collusion or the Evaluator Spies.

The best TMS in our comparison is BubbleTrust. It has the shortest treason detection time, the longest rehabilitation time and allows only 28 % of bogus transaction under the most successful malicious strategy. As far as we know, it is the only one TMS using global experience as feedback verification.

4 Conclusion

In this paper, using simulation framework called P2PTrustSim we compared trust management systems against different malicious strategies. We also proposed several efficiency criteria which can be evaluated using this framework. We analysed known malicious tactics and suggested three new collective malicious strategies against the most representative systems for each type of TMS. We can expect that malicious peers working in a collective will try to use the most effective strategy against TMS currently used in the network. Therefore, the quality of TMSs has to be assessed according to the most successful malicious strategy. Nevertheless, other properties have to be taken into account too; e.g. the cost connected with the malicious strategy can exceed the potential benefit for malicious peers. The results indicate that only the BubbleTrust is resistant against all considered malicious strategies; it is, therefore, the best choice for deployment in the secured P2P networks.

Acknowledgment. This work was supported in part by grants 204/13/08195 and SVV-2013-267312.

References

1. Kamvar, S.D., Schlosser, M.T., Garcia-Molina, H.: The EigenTrust algorithm for reputation management in P2P networks. In: WWW'03: Proceedings of the 12th International Conference on World Wide Web, pp. 640–651. ACM Press (2003)
2. Xiong, L., Ling, L.: PeerTrust: supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Trans. Knowl. Data Eng.* **16**, 843–857 (2004)

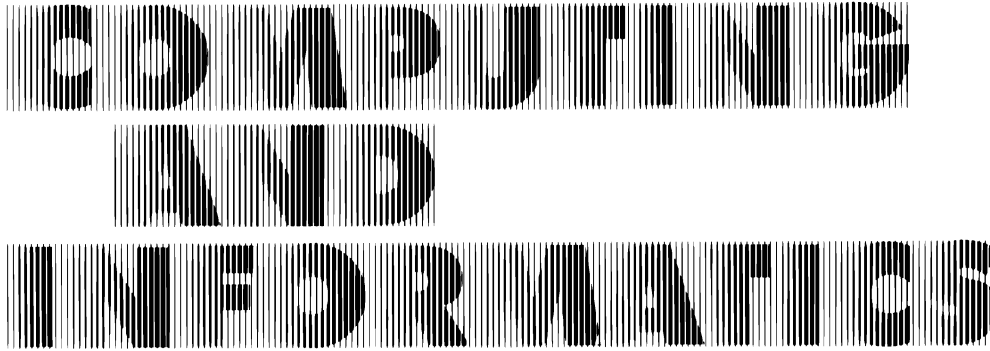
3. Huanyu, Z., Xiaolin, L.: H-Trust: a group trust management system for peer-to-peer desktop grid. *J. Comput. Sci. Technol.* **24**, 447–462 (2009)
4. Bonnaire, X., Rosas, E.: WTR: a reputation metric for distributed hash tables based on a risk and credibility factor. *J. Comput. Sci. Technol.* **24**, 844–854 (2009)
5. Novotny, M., Zavoral, F.: BubbleTrust: a reliable trust management for large P2P networks. In: Meghanathan, N., Boumerdassi, S., Chaki, N., Nagamalai, D. (eds.) CNSA 2010. CCIS, vol. 89, pp. 359–373. Springer, Heidelberg (2010)
6. Novotny, M., Zavoral, F.: Resistance against malicious collectives in BubbleTrust. In: The 12th International Conference on Parallel and Distributed Computing, Gwangju, Korea (2011)
7. Hoffman, K., Zage, D., Nita-Rotaru, C.: A survey of attack and defense techniques for reputation systems. *ACM Comput. Surv.* **42**(1), 1–31 (2009)
8. Marti, S., Garcia-Molina, H.: Taxonomy of trust: categorizing P2P reputation systems. *Comput. Netw.* **50**, 472–484 (2006)
9. Selvaraj, C., Anand, S.: Peer profile based trust model for P2P systems using genetic algorithm. *Peer-to-Peer Netw. Appl.* **4**, 1–12 (2011)
10. Suryanarayana, G., Taylor, R.N.: A survey of trust management and resource discovery technologies in peer-to-peer applications. Technical report, UC Irvine (2004)
11. Bonnaire, X., Rosas, E.: A critical analysis of latest advances in building trusted P2P networks using reputation systems. In: Weske, M., Hacid, M.-S., Godart, C. (eds.) WISE Workshops 2007. LNCS, vol. 4832, pp. 130–141. Springer, Heidelberg (2007)
12. Druschel, P., Rowstron, A.: PAST: a large-scale, persistent peer-to-peer storage utility. In: Proceedings of the Eighth Workshop Hot Topics in Operating Systems (2001)

Chapter 8.

Metro-NG: Computer-Aided Scheduling and Collision Detection

David Bednárek, Jakub Yaghob, Filip Zavoral

Computing and Informatics, Vol. 33, Num. 6, pp. 1-27, 2014



Volume
34
2015
Number
2

formerly: COMPUTERS AND ARTIFICIAL INTELLIGENCE

CONTENTS

- | | | |
|------------------------------------------------------|-----|------------------------------------------------------------------------------------------------------------|
| D. BEDNÁREK, J. YAGHOB,
F. ZAVORAL | 277 | Metro-NG: Computer-Aided Scheduling and Collision Detection |
| O. ZAMAZAL, V. SVÁTEK | 305 | PatOMat – Versatile Framework for Pattern-Based Ontology Transformation |
| V. MARINKOVIĆ | 337 | Proof Simplification in the Framework of Coherent Logic |
| J. ZENG, F. WANG, Y. SUN | 367 | A Natural Hand Gesture System for People with Brachial Plexus Injuries |
| D. BRODIĆ, Č. A. MALUCKOV,
L. PENG | 383 | Statistics Oriented Preprocessing of Document Image |
| M. M. ATANAK, A. DOĞAN | 402 | Improving Real-Time Data Dissemination Performance by Multi Path Data Scheduling in Data Grids |
| A. SZYMCZAK, M. PASZYŃSKI,
D. PARDO, A. PASZYŃSKA | 425 | Petri Nets Modeling of Dead-End Refinement Problems in a 3D Anisotropic hp-Adaptive Finite Element Method |
| Z. FARZANYAR,
M. KANGAVARI | 458 | Distributed Frequent Item Sets Mining over P2P Networks |
| B. KRYZA, J. KITOWSKI | 473 | Comparison of Information Representation Formalisms for Scalable File Agnostic Information Infrastructures |
| C. WEN, T.-Z. HUANG,
T. SOGABE | 495 | An Extension of Two Conjugate Direction Methods to Markov Chain Problems |

ISSN 1335-9150

METRO-NG: COMPUTER-AIDED SCHEDULING AND COLLISION DETECTION

David BEDNÁREK, Jakub YAGHOB, Filip ZAVORAL

Charles University in Prague

Faculty of Mathematics and Physics

Malostranské nám. 25

118 00 Prague, Czech Republic

e-mail: {bednarek, yaghob, zavoral}@ksi.mff.cuni.cz

Abstract. In this paper, we propose a formal model of the objects involved in a class of scheduling problems, namely in the classroom scheduling in universities which allow a certain degree of liberty in their curricula. Using the formal model, we present efficient algorithms for the detection of collisions of the involved objects and for the inference of a tree-like navigational structure in an interactive scheduling software allowing a selection of the most descriptive view of the scheduling objects. These algorithms were used in a real-world application called MetroNG; a visual interactive tool that is based on more than 10 years of experience we have in the field. It is currently used by the largest universities and colleges in the Czech Republic. The efficiency and usability of MetroNG suggests that our approach may be applied in many areas where multi-dimensionally structured data are presented in an interactive application.

Keywords: Scheduling tools, collision detection, visualization

1 INTRODUCTION

Every university is faced with the problem of creating an acceptable schedule of their educational events within available resources of time, space, and personnel.

The main problem when creating a schedule of university lectures and seminars lies in checking and preventing collisions – one teacher should not teach two lectures at the same time, students should not visit more than one lesson at the same time, etc. Moreover, there is a great number of soft or hard constraints, e.g. a teacher

cannot teach in a particular time period or students are not happy to have lessons continually 12 hours in one day. Therefore, creating ‘good’ schedule is a difficult task which requires some IT support.

The problem could be described as a constraint satisfaction problem (CSP) [14]. The formal constraints and the quality criteria or the objective functions should be specified in order to use an appropriate solver [10].

Each CSP solver requires the exact formal specification of all constraints. Besides individual constraints, the members of academia have their individual ideas of ‘good’ schedules; therefore, the cost or objective function shall be adjusted individually per each scheduling object. These constraints and objective functions are fuzzy and unclear by nature and ordinary users of the system are not able to specify them in the necessary formal manner. Thus, the use of the CSP solver requires skilled personnel to formalize all requirements of the users into constraints and objective functions.

An alternative approach to a complete formalization of the problem for a scheduling algorithm is creating the schedule by humans, using a software application to display and efficiently customize the (partial) schedule.

Although the manual schedule creation itself requires more human effort than an algorithmic solution, the difference may be balanced or even outweighed by the cost of formalization of the constraints. In addition, human-based scheduling offers the following advantages:

- humans can effectively work with incompletely defined constraints
- humans can intuitively recognize erroneously entered constraints
- humans can communicate individually to the subjects in case of ambiguity or to negotiate a relaxation of constraints
- in case of inevitably conflicted schedule, humans can find the ‘least bad’ solution instead of giving-up completely
- in case of complaints, humans can explain the rationale behind the schedule.

The importance of individual advantages and disadvantages of automated and manual approaches depends on many factors, including the organization structure of the university, the work flow of the scheduling, the level of the academic liberty, etc. According to our experience, the manual approach is still viable and favorable in some universities, despite the recent progress in the CSP and other algorithmic scheduling techniques.

A mixed approach is also possible: using a CSP solver with a weaker and smaller set of constraints as the first step and then the manual adjustment for the final tuning of the schedule.

Human-based scheduling requires strong software support capable at least of these tasks:

- display several views of the schedule (e.g., a schedule of the lecture rooms, the schedule of particular student groups, etc.)

- automatically or manually choose an appropriate level of detail of particular views; in other words, the application must present some *navigation structures* to allow efficient exploration of the scheduling space
- display all possible collisions (e.g., more lectures intended for the same student group in overlapping time slots, a lecture of a particular teacher conflicting with his constraints, etc.).

The article presents a formal model for the complex collision detection system which is used to identify situations where a group of students should attend two classes scheduled for the same time. Next, we present an incremental algorithm used to extract the navigation structures from the formal model. Using this model, we developed an application called MetroNG¹ for supporting the whole process of creating a complex university and curricula schedule. Besides the sophisticated student collision detection, the application has other unique features such as the combination of several views, handling of transfer collisions and various types of scheduled events besides regular ‘once-a-week’ classes.

The rest of the article is organized as follows: Next section contains related work, Section 3 describes the issues addressed in the article. The most crucial part of the article is contained in Sections 4 and 5 that describe a theoretical model of the entities and their collisions and the respective algorithms. Section 6 describes the MetroNG application that is based on the previously proposed formal methods. The final section summarizes and concludes the article.

2 RELATED WORK

There is a lot of research on classroom or curriculum scheduling (also called timetabling in some sources). The proposed solutions range from simple applications to large automatic or semi-automatic CSP solvers.

The scheduling problems are NP-complete in general [42], as far as their computational complexity is concerned. Therefore, various optimization methods have been proposed for solving the scheduling problem [17, 18, 19, 20]. Other methods are heuristic orderings [22], case-based reasoning [21], genetic/evolutionary algorithms [25, 26, 27], ant systems [30], local search techniques [23, 24], particle swarm optimization [28, 29], tabu search [31, 32], metaheuristics [33] and hyperheuristics [34, 35].

Recent definitions of the course scheduling problem can be found in [33, 36]. Universities have increasingly relied upon the automation of this task to produce efficient timetables that satisfy these constraints [36]. Many recent papers have been published on specific techniques [37, 38, 39] dealing with the university course scheduling problem.

¹ <https://www.erudio.cz/?stranka=sw.metrong>

High school timetabling involves weekly scheduling for all lecturers of the high school, where the schedule is regular, the number of events is quite small, there are no collisions, the constraints are simple, etc. The problem consists in assigning lectures to timeslots while other constraints of several different kinds are satisfied. These constraints may include both hard constraints that must be respected and soft constraints used to evaluate the solution's quality [40].

Some recent papers have been published on these specific techniques [41, 42, 43, 11, 12, 13] and corresponding software solutions [6, 7, 8] oriented to high-school timetabling were described. These tools are not sufficient for most university environments, mostly because they are not capable to capture the more complex structure of the groups of students.

The system [9] uses individual students as the subjects of scheduling. Of course, this approach is viable only in environments where the set of individual students for each lecture is known prior to the scheduling phase. In addition, this fine-grained approach is extremely demanding in terms of computing resources.

Systems combining automatic scheduling with human interaction described as interactive or semiautomatic scheduling were described in [15, 16].

The solution proposed in this paper is based on a strong theoretical background defined in Section 4. The problem of modeling events and their visitors is similar to some problems in the area of *concept analysis* – for instance, the role of test context in the Contextual Attribute Logic [3, 4] is similar to our group base (see Section 4.2). Nevertheless, our intent is visualization and not reasoning; therefore, our formal model is different.

3 PROBLEM DESCRIPTION

The scheduling problem essentially consists of a set of *events*. Each event is associated to a set of participants – teachers and students. While the teachers are easily identified by their enumeration, the description of the associated set of students is difficult: The enumeration of individual students is usually impossible because they are not known at the moment of scheduling. Instead of the enumeration, the student participants of an event are described in terms of student *groups*.

The schedule consists of the mapping of events to space and time, i.e. assigning a room and time slot to each event. The time slots are usually but not necessarily recurrent. The schedule shall satisfy a number of hard and soft constraints; most importantly it shall avoid *collisions*. However, since finding a collision-free schedule is not necessarily feasible, the system must be able to manipulate schedules containing collisions.

The main purpose of the scheduling application is to display the schedule and allow the users to modify it. The visualization part must cope with the huge amount of information present in the schedule. An average schedule of a college or university contains hundreds or thousands of events, spans several buildings with tens or hundreds of lecture rooms, and handles hundreds of teachers and many thousands

of students. Since it is too much information to fit in one view, the visualization must be highly interactive and provide the user with only that part of information that he or she is interested in at the moment.

Although the modification of the schedule is basically a simple drag-and-drop action, it is a part of a larger, very complex problem. While the whole decision making is left to the user (the user picks the time and room for each event), it is not sufficient to give the users the ability to schedule the events. To do the task efficiently, they need a lot of information beyond “what, when and where”. The most important piece of information is whether they are creating a collision-free schedule.

3.1 Student Groups

In the physical world, student groups consist of individual students who, for whatever reason, visit a particular event. In our setting, we do not know the individual students in the moment of scheduling; instead, the group must be described by using the properties of the expected participants.

The set of students visiting a particular lecture may be characterized using *attributes* like study program, specialization, year, etc. Some of them may be assigned to the students by an authority, others (like proficiency level) may represent a fiction created by the scheduling personnel.

For the collision checking, the system must be able to determine whether two student groups intersect. This test conceptually corresponds to testing intersection of two sets; however, these sets are not described by enumerating their contents. Instead, the intersection test shall be based on the attributes of the student groups.

Moreover, the set of real students does not completely fill the space of all attribute combinations. For instance, the group of ‘second-year’ students may or may not intersect with the group of ‘beginners’, depending on the current (or estimated) state of the student population. In our system, the student population is modeled by a *group base* – a set of student *prototypes*.

3.2 Visualization of Groups

Student attributes form a multi-dimensional space which cannot be visualized directly. Some dimensions may be mutually dependent (e.g., study program and specialization), other pairs are considered independent (e.g., language skills vs. specialization).

When creating a schedule, a portion of this multi-dimensional space must be accessible in an appropriate level of detail. A straightforward approach to selecting a portion of the space would be using a form to enter the filtering rules for each attribute. However, to control such a form is tedious; moreover, the user is often unable to predict whether a particular filter leads to a human-readable presentation.

To allow exploration of the attribute space, we propose using a *navigational tree* to perform the navigation and level-of-detail setting by expanding and collapsing

a virtual tree. Besides avoiding annoying forms, this approach also allows to show several portions of the space at once, using different levels of detail in different parts of the space. In this sense, the tree replaces a filter in the form of disjunction of conjunctive clauses.

While this idea is simple, creating the tree is not easy. First, the degree of individual nodes of the tree shall be kept as low as possible in order to make the navigation easy for the user. In other words, while the user must be able to reach any desired view in the tree, the number of paths to the view shall be minimized. Second, the size of the completely expanded tree is exponential with respect to the size of group base; therefore, the tree shall be constructed incrementally.

3.3 Collisions

A collision is a state when two events are scheduled in such a way, that someone who is supposed to attend both events would be unable to do so. The two classes of participants – teachers and students – form two classes of collisions.

The collision may occur either directly, when two events intersect in time, or by transfer, when the time distance between the two events is too short to allow transfer between the two locations. In addition, two events may collide if scheduled to the same room at the same time.

While room and teacher-related collisions are relatively easy to detect, the detection of student-related collision involves a complex determination of the intersection of the groups, as described in Subsection 3.1. Also the visualization of the space of students must be arranged so that the user can easily determine the danger of collisions.

4 FORMAL MODEL

Instead of modeling individual students, the system works with *prototypes*; each prototype represents a group of students assumed to have the same preferences and/or prescriptions with respect to the lectures/events visited. The set of prototypes, called *group base*, is defined by human operators of the scheduling system, as accurate as possible and/or necessary.

The group base concept is similar to the test context in the Contextual Attribute Logic [3, 4] since the problem of modeling events and their visitors is similar to some problems in the area of *concept analysis* [2].

Given a group base, the visitors of an event could be defined by using a set of prototypes. However, it would be impractical for a human operator because such sets may be quite large. Instead of enumerating prototypes, student groups may be defined by using tuples of attributes as described in the following paragraphs.

Prototypes are distinguished by *attributes* carrying values (*tags*) from finite *domains*. Some attributes correspond to properties defined in the real world and assigned to real students, like the field of study, the year, or an administratively

assigned group number. Others are defined solely for the purpose of scheduling, representing for instance assumed future specialization.

4.1 Attributes and Tuples

Each instance of the scheduling system defines a finite set \mathcal{A} of attributes, divided into single-valued \mathcal{A}_s and multi-valued \mathcal{A}_m attributes. Each attribute $a \in \mathcal{A}$ is associated to a finite domain \mathcal{D}_a ; members of these domains are called *tags*. These domains are specific for a particular school and they develop slowly over time to reflect the changes in the curricula. For the simplicity of notation, we assume that the attribute domains are pairwise disjoint; the union of domains is marked $\mathcal{D} = \bigcup\{\mathcal{D}_a \mid a \in \mathcal{A}\}$.

Example 1. Throughout this paper, we will show several examples based on the following set of attributes:

$$\begin{aligned}\mathcal{A}_s &= \{D, P, R\}, \\ \mathcal{A}_m &= \{S\}.\end{aligned}$$

The corresponding domains are:

$$\begin{aligned}\mathcal{D}_D &= \{B, N\}, \\ \mathcal{D}_P &= \{I, M\}, \\ \mathcal{D}_R &= \{1, 2, 4, 5\}, \\ \mathcal{D}_S &= \{IPR, IOI, ISS\}.\end{aligned}$$

Although our formalism distinguishes single-valued and multi-valued attributes, many definitions become simplified when the values of both kinds of attributes are viewed as sets. In this approach, the value of an attribute is a set of tags, i.e., a subset of the associated attribute domain. For a single-valued attribute, the cardinality of the attribute value is restricted to at most 1. The ability to assign the empty-set value to an attribute loosely corresponds to the concept of null value in relational databases; however, the treatment of empty values in predicates is different from the treatment of null values in databases.

All tuples in our system have the same schema, i.e., all tuples contain all attributes defined in the system (nevertheless, the value of an attribute may be empty in a tuple).

Definition 1 (Tuple). A *tuple* is a mapping $t : \mathcal{A} \rightarrow \mathbf{P}(\mathcal{D})$ such that $t(a) \subseteq \mathcal{D}_a$ and $a \in \mathcal{A}_s \Rightarrow \mathbf{card}(t(a)) \leq 1$. The universe of all such tuples is denoted \mathcal{T} .

Each tuple associates a value (a set of tags) to each attribute. In most cases, a tuple serves as a means to define a set of certain individuals by declaring which tags shall be present in these individuals. The more tags a tuple contains, the less individuals fit to the tuple; if an attribute of a tuple has the empty value, it does

not restrict the set of individuals anyhow. This approach, reflected in the definitions below (and different from traditional null-value handling in relational systems), is motivated by the required conservative-approximation rule: “Possible collision is a collision.”

Definition 2 (Tuple meet). The *meet* (denoted $t_1 \sqcap t_2$) of a pair of tuples $t_1, t_2 \in \mathcal{T}$ is the mapping t_3 such that $t_3(a) = t_1(a) \cup t_2(a)$ for each $a \in \mathcal{A}$, provided t_3 is a correct tuple, i.e. $t_3 \in \mathcal{T}$.

The name “meet” was borrowed from the lattice theory; however, it should be stressed that the tuple meet is not a semilattice because the meet may not exist. Also note that the sense of the operation is reverted with respect to the lattice of tag sets; i.e. tuple meet corresponds to union of tag sets. This reversion corresponds to the fact that adding attributes diminishes the set of individuals conforming to these attributes. Similarly to lattice theory, the tuple meet operation is associated with a partial ordering, named *tuple inclusion*.

Definition 3 (Tuple inclusion). A tuple $t_1 \in \mathcal{T}$ is considered *included* in a tuple $t_2 \in \mathcal{T}$ (denoted $t_1 \sqsubseteq t_2$), if for each attribute $a \in \mathcal{A}$, $t_2(a) \subseteq t_1(a)$.

The rationale behind this definition is that a tuple defines a set of individuals by defining a set of required tags of these individuals. The tuple inclusion relation corresponds to the set inclusion relation on the associated sets of individuals. If t_1 requires more tags than t_2 , then any individual satisfying t_1 also satisfies t_2 . Note that tuple inclusion is a partial order on \mathcal{T} .

Example 2. The following mapping

$$t_1 = \{D \mapsto \{B\}, P \mapsto \emptyset, R \mapsto \{1\}, S \mapsto \{IPR, IOI\}\}$$

is a correct tuple with respect to the attribute set defined in the previous example. In the following examples, we will use the traditional positional notation; in addition, we will omit braces containing a single tag. Thus, t_1 will be written as $(B, \emptyset, 1, \{IPR, IOI\})$. For example,

$$t_1 \sqcap (B, I, \emptyset, ISS) = (B, I, 1, \{IPR, IOI, ISS\})$$

while $t_1 \sqcap (B, \emptyset, 2, ISS)$ is not defined because the attribute R is single-valued. Furthermore,

$$(B, I, 1, \{IPR, IOI, ISS\}) \sqsubseteq t_1$$

while (B, I, \emptyset, ISS) and t_1 are incomparable.

Definition 4 (Tuple collision). A pair of tuples $t_1, t_2 \in \mathcal{T}$ are said to *collide* (denoted $t_1 \triangle t_2$), if there exists a tuple $t_3 \in \mathcal{T}$ such that $t_3 \sqsubseteq t_1$ and $t_3 \sqsubseteq t_2$.

Such a tuple t_3 must satisfy the condition $t_1(a) \cup t_2(a) \subseteq t_3(a)$ for each $a \in \mathcal{A}$. The existence of such a tuple may be prevented by the condition $a \in \mathcal{A}_s \Rightarrow \text{card}(t(a)) \leq 1$ from the definition of tuple. If such tuples exists, the tuple meet $t_1 \sqcap t_2$ is one of them. Therefore, the following lemma holds:

Lemma 1 (Tuple collision). A pair of tuples $t_1, t_2 \in \mathcal{T}$ *collides* ($t_1 \Delta t_2$) if and only if

$$t_1(a) \neq \emptyset \wedge t_2(a) \neq \emptyset \Rightarrow t_1(a) = t_2(a)$$

for each single-valued attribute $a \in \mathcal{A}_s$.

Example 3. The tuples $(B, I, \emptyset, \emptyset)$ and $(\emptyset, I, 4, \emptyset)$ collide while $(B, M, \emptyset, \emptyset)$ and $(\emptyset, I, 4, \emptyset)$ do not.

4.2 Group Base

Group base is the device used to specify which individuals exist by the means of prototypes. Each prototype is modeled by a tuple; a *group base* is simply a set of tuples, as shown in the following definition.

Definition 5 (Group base). A *group base* is a set $\mathcal{G} \subseteq \mathcal{T}$. A group base \mathcal{G} is *well-formed* if $t_1 \Delta t_2 \Rightarrow t_1 = t_2$ for each $t_1, t_2 \in \mathcal{G}$.

Each scheduling problem defines a group base and all (student-based) scheduling constraints are related to this group base. We assume that there are no other individuals than those corresponding to the prototypes enumerated in the group base. Consequently, any group of individuals is completely described by a set of prototypes from the group base. Instead of enumerating these prototypes directly, a tuple inclusion offers the ability to define a set of prototypes using a single tuple, as shown in the following definition of *trace*. Of course, not every set of prototypes is a trace of a tuple; however, every set of prototypes may be defined as a union of traces, i.e., any group of individuals may be described using a set of tuples. In real-life cases, these tuples offer a significantly shorter description than the enumeration of prototypes.

Definition 6 (Trace). The set

$$\mathbb{T}_{\mathcal{G}}(t) = \{g \in \mathcal{G} \mid g \sqsubseteq t\}$$

is called the *trace* of a tuple t with respect to a group base \mathcal{G} .

The following lemma shows how the tuple meet operation corresponds to operations on prototype sets.

Lemma 2 (Trace of tuple meet). For each pair of tuples $t_1, t_2 \in \mathcal{T}$ such that $t_1 \sqcap t_2$ exists, $\mathbb{T}_{\mathcal{G}}(t_1 \sqcap t_2) = \mathbb{T}_{\mathcal{G}}(t_1) \cap \mathbb{T}_{\mathcal{G}}(t_2)$.

The following important notions are defined with respect to a given group base \mathcal{G} : \mathcal{G} -relative inclusion and collision. In these definitions, a group base functions as a restriction on the set of individuals existing in the system; thus, the inclusion relation is weakened and the collision relation strengthened by the \mathcal{G} -relativization, as shown in Lemma 3.

Definition 7 (Relative inclusion). For a pair of tuples $t_1, t_2 \in \mathcal{T}$, t_1 is considered \mathcal{G} -included in t_2 with respect to a group base \mathcal{G} (denoted $t_1 \sqsubseteq_{\mathcal{G}} t_2$), if $\mathsf{T}_{\mathcal{G}}(t_1) \subseteq \mathsf{T}_{\mathcal{G}}(t_2)$.

Definition 8 (Relative collision). Tuples $t_1, t_2 \in \mathcal{T}$ are said to \mathcal{G} -collide with respect to a group base \mathcal{G} (denoted $t_1 \Delta_{\mathcal{G}} t_2$), if $\mathsf{T}_{\mathcal{G}}(t_1) \cap \mathsf{T}_{\mathcal{G}}(t_2) \neq \emptyset$.

Lemma 3. For each pair of tuples $t_1, t_2 \in \mathcal{T}$

$$t_1 \sqsubseteq t_2 \Rightarrow t_1 \sqsubseteq_{\mathcal{G}} t_2, \quad t_1 \Delta t_2 \Rightarrow t_1 \Delta_{\mathcal{G}} t_2.$$

D	P	R	S
B	I	1	IPR
B	I	1	IOI
B	I	1	ISS
B	I	2	IPR
B	I	2	IOI
B	M	1	
B	M	2	
N	I	4	
N	I	5	

Table 1. Example: A group base

Example 4. Let \mathcal{G} be the group base shown in Table 1. Let $t_3 = (N, \emptyset, 4, \emptyset)$, $t_4 = (N, I, \emptyset, \emptyset)$. Their traces with respect to \mathcal{G} are

$$\mathsf{T}_{\mathcal{G}}(t_3) = \{(N, I, 4, \emptyset)\}; \mathsf{T}_{\mathcal{G}}(t_4) = \{(N, I, 4, \emptyset), (N, I, 5, \emptyset)\}$$

and they are, by definition, ordered by \mathcal{G} -inclusion ($t_3 \sqsubseteq_{\mathcal{G}} t_4$) although they are incomparable by inclusion ($t_3 \not\sqsubseteq t_4$). Furthermore, the tuples $(B, I, \emptyset, \emptyset)$ and $(\emptyset, I, 4, \emptyset)$ do not \mathcal{G} -collide although they do collide (see Example 3).

4.3 Normalization

The relativization with respect to a group base may cause that two different tuples correspond to the same group of individuals, which means that the two tuples are equivalent with respect to the group base.

Definition 9 (Relative equivalence). Tuples $t_1, t_2 \in \mathcal{T}$ are considered \mathcal{G} -equivalent (denoted $t_1 \approx_{\mathcal{G}} t_2$), if $t_1 \sqsubseteq_{\mathcal{G}} t_2 \wedge t_2 \sqsubseteq_{\mathcal{G}} t_1$.

The \mathcal{G} -inclusion is induced by set-inclusion on traces; therefore, it is a partial order on \mathcal{T} . Consequently, the \mathcal{G} -equivalence is an equivalence.

We will show that \mathcal{G} -equivalence classes on tuples can be uniquely represented by *normalized* tuples.

Definition 10 (Implied attribute). An attribute $a \in \mathcal{A}$ is called *implied* with respect to a tuple $t \in \mathcal{T}$ and a group base \mathcal{G} (denoted $t \vdash_{\mathcal{G}} a$), if there exists a tag $v \in \mathcal{D}_a$ such that $t \approx_{\mathcal{G}} (t \sqcap \{a \mapsto \{v\}\})$. The attribute is called *non-trivially implied* if $v \notin t(a)$.

Definition 11 (Normalized tuple). A tuple $t \in \mathcal{T}$ is called *normalized* with respect to a group base \mathcal{G} , if $\mathsf{T}_{\mathcal{G}}(t) \neq \emptyset$ and there exists no attribute being non-trivially implied with respect to t .

Lemma 4 (Tuple equivalence). If tuples $t_1, t_2 \in \mathcal{T}$ are normalized with respect to a group base \mathcal{G} then

$$t_1 \approx_{\mathcal{G}} t_2 \Rightarrow t_1 = t_2.$$

Each tuple whose trace is non-empty can be normalized by adding implied attributes and their values until a normalized tuple is reached. In addition, the lemma essentially states that normalized tuples are unique representatives of equivalence classes induced by the \mathcal{G} -equivalence, with the exception of the class of tuples whose trace is empty.

Normalized tuples are also representatives of their traces. Not every subset of a group base \mathcal{G} is a trace of a tuple; however, every subset of \mathcal{G} can be represented by a union of traces of some set of tuples. Thus, sets of normalized tuples can represent subsets of \mathcal{G} .

4.4 Events and Visitors

In our scheduling problem, a subset of \mathcal{G} represents the visitors of an event. However, this subset may be quite large for human perception; therefore, an alternative representation using normalized tuples is understood more easily because it can be usually significantly smaller. On the other hand, for processing in software, the traces can be stored in bitmaps and handled using Boolean operators.

In addition, representation by tuples usually works in better accordance with reality when the group base \mathcal{G} is changed due to evolution of curricula. Therefore, tuples are more suitable for persistent representation than subsets of \mathcal{G} .

These observations led to the following design principles:

- When presented to users, representation using normalized tuples is always used.
- In the database and in SQL-based applications (web interface), normalized tuples are used.
- In C++ applications, tuples are converted to bitmaps representing traces and handled using vectorized bit operations.

Definition 12 (Events). Let \mathcal{G} be a group base; \mathcal{E} be a set of events. A mapping $\beta : \mathcal{E} \rightarrow \mathbf{P}(\mathcal{T})$ is called *event binding*. An event binding is called *normalized* if, for each event $e \in \mathcal{E}$, each tuple in $\beta(e)$ is normalized with respect to \mathcal{G} and $t_1 \sqsubseteq_{\mathcal{G}} t_2 \Rightarrow t_1 = t_2$ for each $t_1, t_2 \in \beta(e)$.

The visualization algorithm uses a precomputed relation $\mathbf{M}_{\mathcal{G}} \subseteq \mathcal{E} \times \mathcal{G}$ stored as a binary matrix according to the following definition.

Definition 13 (Concern matrix).

$$\mathbf{M}_{\mathcal{G}} = \{ \langle e, t \rangle \mid (\exists t_e \in \beta(e)) t \in \mathbf{T}_{\mathcal{G}}(t_e) \}.$$

Definition 14 (Event collision). Events $e_1, e_2 \in \mathcal{E}$ are said to \mathcal{G} -*collide* if there are tuples $t_1 \in \beta(e_1)$ and $t_2 \in \beta(e_2)$ such that $t_1 \Delta_{\mathcal{G}} t_2$.

4.5 Visualization of the Tuple Hierarchy

As mentioned in Section 3.1, the partially-ordered set of groups is displayed using a tree. In this section, we will define the tree more formally.

Definition 15 (Attribute tree). *Attribute tree* is a labeled unranked rooted tree

$$\mathcal{U} = (V_m, V_p, r, \pi, L_m, L_p, \tau)$$

whose nodes are of three kinds – *meta*-nodes V_m , *plain* nodes V_p and the root node r . The mapping

$$\pi : (V_m \rightarrow (V_p \cup \{r\})) \cup (V_p \rightarrow V_m)$$

defines the structure of the tree by defining the parent $\pi(n)$ of every node $n \neq r$. The mapping

$$L_m : V_m \rightarrow \mathcal{A}$$

assigns attributes to *meta*-nodes and the mapping

$$L_p : V_p \rightarrow \mathcal{D}$$

assigns tags to plain nodes. Each node n is also assigned a tuple $\tau(n)$ defined recursively as

$$\begin{aligned} \tau(r) &= \emptyset, \\ (\forall n \in V_m) \tau(n) &= \tau(\pi(n)), \\ (\forall n \in V_p) \tau(n) &= \tau(\pi(n)) \sqcap \{L_m(\pi(n)) \mapsto \{L_p(n)\}\}. \end{aligned}$$

Definition 16 (Distinctive attribute tree). An attribute tree is called *distinctive* with respect to a group base \mathcal{G} if

$$(\forall n \in V_p) \mathbf{T}_{\mathcal{G}}(\tau(n)) \neq \emptyset \wedge \tau(n) \not\approx_{\mathcal{G}} \tau(\pi(n)).$$

Definition 17 (Functional dependency). An attribute $b \in \mathcal{A}$ is called *functionally dependent* on an attribute $a \in \mathcal{A}$ with respect to a tuple $t \in \mathcal{T}$ and a group base \mathcal{G} (denoted $t \vdash_{\mathcal{G}} a \rightsquigarrow b$), if, for each tag $v \in \mathcal{D}_a$, the attribute b is implied with respect to the tuple $(t \sqcap \{a \mapsto \{v\}\})$ and the group base \mathcal{G} . The functional dependency is called *non-trivial* if $b \neq a$ and $t(b) = \emptyset$.

Definition 18 (Non-skipping attribute tree). An attribute tree is called *non-skipping* with respect to a group base \mathcal{G} if

$$(\forall n \in V_m) \tau(\pi(n)) \vdash_{\mathcal{G}} a \rightsquigarrow L_m(n) \Rightarrow a = L_m(n).$$

An attribute tree is used to display (a part of) the partial order defined by the \mathcal{G} -inclusion on the set of normalized tuples. In this sense, the role of the attribute tree is similar to Hasse diagrams [5]; however, the interactive environment allows to incrementally unroll the lattice DAG to a tree. Note that the completely unrolled tree may have an exponential number of nodes with respect to the original DAG; therefore, the interactive incremental approach is crucial.

Attribute trees are constructed incrementally from the group base \mathcal{G} , whenever the user expands a node n . The construction algorithm scans the trace $T_{\mathcal{G}}(\tau(n))$ to determine which attributes are functionally dependent on $\tau(n)$. These attributes are excluded; in addition, the application may exclude some attributes based on site-specific configuration. The remaining attributes, if any, generate a meta-node children of n . When expanding a meta-node, all corresponding attribute values in $T_{\mathcal{G}}(\tau(n))$ are used.

An attribute tree contains interleaved layers of meta-nodes and value nodes; each pair of layers corresponds to adding an attribute/value pair to the corresponding tuple. The two layers allow the user to select the required attribute first; then, values from the corresponding domain are selected. In a distinctive attribute tree, the children of a meta-node n_m correspond to a disjoint (in the sense of \mathcal{G} -collision) cover of the parent value node $\pi(n_m)$. Consequently, a sibling of n_m contains the same set of groups; arranged, however, in a different manner.

4.6 Visualization of Event Binding

When displaying the schedule relevant to a tuple t , *concern relations* are used to determine the associated set of events. *Covering concern* collects all events which are bound to all individuals from t (i.e. all basic tuples included in t) while *intersecting concern* contains events bound to some individuals from t . Naturally, the covering concern is a subset of the intersecting concern.

Definition 19 (Concern relations). The mappings $\text{cover}_{\mathcal{G}}, \text{intersect}_{\mathcal{G}} : \mathcal{T} \rightarrow \mathbf{P}(\mathcal{E})$ are called *covering concern* and *intersecting concern* and defined as

$$\begin{aligned} \text{cover}_{\mathcal{G}}(t) &= \{e \in \mathcal{E} \mid (\exists t_e \in \beta(e)) t \sqsubseteq_{\mathcal{G}} t_e\}, \\ \text{intersect}_{\mathcal{G}}(t) &= \{e \in \mathcal{E} \mid (\exists t_e \in \beta(e)) t \Delta_{\mathcal{G}} t_e\}. \end{aligned}$$

For each node n of the attribute tree, the application displays all events from the covering concern which are not in the covering concern of the parent, i.e., the event set $\text{cover}_{\mathcal{G}}(\tau(n)) \setminus \text{cover}_{\mathcal{G}}(\tau(\pi(n)))$, horizontally positioned at their scheduled time. Note that for an internal node a covering event is displayed using a rectangle whose height corresponds to the visual height of the node. In addition, events from $\text{intersect}_{\mathcal{G}}(\tau(n)) \setminus \text{cover}_{\mathcal{G}}(\tau(n))$ are displayed as grayed areas to indicate their presence.

When two covering events $e_1, e_2 \in \text{cover}_{\mathcal{G}}(\tau(n))$ intersect at the time axis, it is obvious that there is a collision and the intersecting area of their rectangles is painted in red to indicate the problem. Similarly, if a covering event intersects with an intersecting event, it also means a collision; however, this kind of collision is weaker because it affects only a part of individuals from $\tau(n)$. On the other hand, if two intersecting events $e_1, e_2 \in \text{intersect}_{\mathcal{G}}(\tau(n))$ intersect at the time axis, it may or may not indicate a problem – in this case, the pair of events must be examined using the definition of event collision (see Section 4.4).

Covering events may be computed using bit-vector operations on columns $\mathbf{M}_{\mathcal{G}}^T(t)$ of the concern matrix, according to the following lemma.

Lemma 5 (Concern relations).

$$\begin{aligned} \text{cover}_{\mathcal{G}}(t_n) &= \bigcap \left\{ \mathbf{M}_{\mathcal{G}}^T(t) \mid t \in \mathbb{T}_{\mathcal{G}}(t_n) \right\}, \\ \text{intersect}_{\mathcal{G}}(t_n) &= \bigcup \left\{ \mathbf{M}_{\mathcal{G}}^T(t) \mid t \in \mathbb{T}_{\mathcal{G}}(t_n) \right\}. \end{aligned}$$

Events found using the bit-vector arithmetics are then ordered by their position on the time axis and their mutual collisions are checked during a single scan along the time axis, as shown in the following section.

5 ALGORITHMS

In order to use the formal model practically, we have developed three algorithms that compute the useful knowledge from the underlying data. The most important algorithm is the detection of collisions among events in a given context. Other two tightly coupled algorithms, the extraction algorithm and the generator of functional dependencies, are used for user navigation in the attribute trees. The algorithms were then used in the application MetroNG [1], see Section 6 for its description.

5.1 Collision Detection

The basic task solved by the collision detector is the enumeration of all collisions among all events in a given context, i.e., all events in the intersecting concern $\text{intersect}_{\mathcal{G}}(t_C)$ (see Definition 19) of a given tuple $t_C \in \mathcal{T}$. The corresponding algorithm is shown in Algorithm 1.

The collision detection algorithm starts by the enumeration of all events in the intersecting concern (line 2) – based on the Lemma 5, this can be done in $O(|\mathcal{E}| \cdot |\mathcal{G}|)$

Algorithm 1 Collision Detection Algorithm**Input:** $t_C \in \mathcal{T}$ **Output:** $C \subset \mathbf{P}(\mathcal{E})$ – the set of collisions

```

1:  $A := \emptyset$ 
2: for all  $e \in \text{intersect}_{\mathcal{G}}(t_C)$  do
3:    $A := A \cup \{ \langle e.T_{\text{begin}}, 0, e \rangle, \langle e.T_{\text{end}}, 1, e \rangle \}$ 
4: end for
5: sort  $A$  using lexicographical order on triplets
6:  $E := \emptyset$ 
7: for all  $\langle T, f, e \rangle \in A$  do
8:   if  $f = 0$  then
9:     for all  $e_1 \in E$  do
10:      if  $M_{\mathcal{G}}(e) \cap M_{\mathcal{G}}(e_1) \neq \emptyset$  then
11:         $C := C \cup \{ \{e, e_1\} \}$ 
12:      end if
13:    end for
14:     $E := E \cup \{e\}$ 
15:  else
16:     $E := E \setminus \{e\}$ 
17:  end if
18: end for

```

time. For every event, its begin and end times $e.T_{\text{begin}}$, $e.T_{\text{end}}$ are added to the time axis A and later sorted (line 3 and 5).

The core part of the algorithm examines all intervals on the time axis (line 7), keeping track of active events E at each moment of time. Whenever a new event e starts, it is compared to every previous event e_1 (line 10), using bit operations on rows of the concern to detect event collision according to Definition 14. The worst-case time complexity of the core part is $O(|\mathcal{E}| \cdot |\mathcal{E}| \cdot |\mathcal{G}|)$ because $|A| \leq 2|\mathcal{E}|$ and $|E| \leq |\mathcal{E}|$.

Although the time complexity of the algorithm is, in principle, cubic, its real performance cost is negligible even for an interactive application, because the above-mentioned worst-case limits for the sizes $|A|$ and $|E|$ are highly overestimated with respect to real data. Furthermore, the bit operations may be computed extremely quickly in current computer architectures.

5.2 Extraction Algorithm

The non-skipping attribute trees defined in Definition 18) are used for navigation in structures of event visitors, namely student groups and teachers. When the user selects the appropriate level-of-detail, he/she incrementally expands the branches from the root of the tree until the desired nodes are reached.

Unfortunately, the trees may be very large for a common user (hundreds or thousands of nodes); their presentation in a fully expanded tree is unacceptable. In order to expand only the proper parts of the tree, the extraction algorithm is used.

When a leaf of a non-skipping attribute tree is expanded (by the user), the extraction algorithm computes the set of children of the selected node. If the set is non-empty, the node is no longer a leaf but an internal node – from the theoretical point of view, the previously displayed non-skipping attribute tree is replaced by a new, larger one. In each step, the extraction algorithm computes all feasible nodes, i.e., all possible expansions of the trace.

The algorithm takes the whole tuple set and the expanded tree branch as an input; it incrementally computes a set of successors (attribute types) for the branch. Two principal data structures are used within this algorithm:

- \mathcal{G} : a group base defined by an instance administrator.
- f_D : boolean matrix indexed by attribute types containing functional dependencies between pairs of attributes with respect to the tree branch t_B being currently unrolled. $f_D(x, y) = true \Leftrightarrow t_B \vdash y \rightsquigarrow x$. This matrix is computed on each run of the algorithm when the next tree level is incrementally unrolled.

The algorithm is divided into two main parts – computing next level and generating functional dependencies.

5.2.1 Next Level

The Algorithm 2 contains the main *NextLevel* function. The idea is that based on a functional dependency matrix computed from the tree branch all dependent attribute types are excluded from the set of candidates.

Algorithm 2 NextLevel function

Input: $t_B \in \mathcal{T}$

Output: $A_R \subseteq \mathcal{A}$

- 1: $A_C := \{a \in \mathcal{A} \mid t_B(a) = \emptyset\}$
 - 2: $f_D := \text{GenFunDep}(t_B, A_C)$
 - 3: $A_R := A_C$
 - 4: **for all** $x \in A_C$ **do**
 - 5: **if** $f_D(x, x)$ **then**
 - 6: $A_R := A_R \setminus \{x\}$
 - 7: **end if**
 - 8: **for all** $y \in A_C \mid f_D(x, y)$ **do**
 - 9: $A_R := A_R \setminus \{y\}$
 - 10: **end for**
 - 11: **end for**
-

Detailed description:

- 1: A_C is a set containing all possible candidates for the next tree level, initialized by all unused attributes, i.e., the attributes that are not contained in the tree branch.
- 2: The functional dependency of all attribute pairs is computed.
- 4–11: All unused functionally dependent attributes are removed.
- 5–7: Empty and constant attributes effectively behave like functionally self-dependent, they are removed from A_C .
- 8–10: Each attribute y that is functionally dependent on any unused attribute x is removed since x must be unrolled prior to y .

The remaining attributes in A_C define the set of possible attributes for the next level.

5.2.2 Functional Dependencies

The key operation of the *NextLevel* function is the computation of functional dependencies. The corresponding Algorithm 3 works as follows:

- 1–5: The *lv* and *ch* maps indexed by \mathcal{A} are used for constantness detection, their elements contain last attribute values and number of distinct values respectively. Initially, no attribute values exist, no attribute shall be displayed (it will be changed later).
- 3–4: First, the dependency matrix is filled. The diagonal true values represent self dependence, i.e., the absence of values of such attribute.
- 6–21: Each matching tuple in the tuple set is detected for constantness, non-emptiness and potential functional dependency.
- 6–8: Within each tuple in the tuple set matching the tree branch, all attributes that were not used in the tree branch are tested.
- 9–10: Number of distinct values of such attribute (or, more exactly, value changes) within the matching tuple set is detected.
- 12–13: If the unused attribute is contained in the tuple, the functional self-dependency is cleared.
- 14–18: Now we have one particular attribute in one tuple; each distinct non-empty unused (not contained in the tree branch) attribute is set to be a functionally dependent candidate since there may be a relation between these attributes. These relations will be checked later. The dependency cannot be detected in one pass since there may be independent attributes ($x_1y_1, x_1y_2, x_2y_1, x_2y_2$); in this case all possible combinations should be generated.
- 22–26: Each constant attribute (its value was never changed in the set of matching tuples) is excluded from candidates – there is nothing to select from.

Algorithm 3 Function GenFunDep

Input: $\mathcal{G} \subseteq \mathcal{T}; t_B \in \mathcal{T}$ **Output:** $f_D : \mathcal{A} \times \mathcal{A} \rightarrow \text{Boolean}$

```

1:  $lv : \mathcal{A} \rightarrow \mathcal{D}; lv := \emptyset$ 
2:  $ch : \mathcal{A} \rightarrow \mathbf{N}; (\forall a) ch(a) := 0$ 
3: for all  $\langle x, y \rangle \in \mathcal{A} \times \mathcal{A}$  do
4:    $f_D(x, y) := (x = y)$ 
5: end for
6: for all  $i \in \mathbb{T}_{\mathcal{G}}(t_B)$  do
7:   for all  $x \in \mathcal{A} \setminus \text{dom}(t_B)$  do
8:     if  $i(ax) \neq lv(x)$  then
9:        $ch(x) := ch(x) + 1$ 
10:       $lv(x) := i(x)$ 
11:    end if
12:    if  $x \in \text{dom}(i)$  then
13:       $f_D(x, x) := \text{false}$ 
14:      for all  $y \in \mathcal{A} \setminus \text{dom}(t_B)$  do
15:        if  $x \neq y \wedge y \in \text{dom}(i)$  then
16:           $f_D(x, y) := \text{true}$ 
17:        end if
18:      end for
19:    end if
20:  end for
21: end for
22: for all  $x \in \mathcal{A}$  do
23:   if  $ch(x) \leq 1$  then
24:      $f_D(x, x) := \text{true}$ 
25:   end if
26: end for
27: for all  $i \in \mathbb{T}_{\mathcal{G}}(t_B)$  do
28:   for all  $j \in \mathbb{T}_{\mathcal{G}}(t_B) \wedge j < i$  do
29:     for all  $x \in \mathcal{A} \setminus \text{dom}(t_B)$  do
30:       if  $x \in \text{dom}(i) \wedge x \in \text{dom}(j) \wedge i(x) = j(x)$  then
31:         for all  $y \in \mathcal{A} \setminus \text{dom}(t_B)$  do
32:           if  $x \neq y \wedge (y \notin \text{dom}(i) \vee y \notin \text{dom}(j) \vee i(y) \neq j(y))$  then
33:              $f_D(y, x) := \text{false}$ 
34:           end if
35:         end for
36:       end if
37:     end for
38:   end for
39: end for

```

- 27–39: The main part of the functional dependency detection – dependency candidates are checked.
- 27–29: Each pair of the matching tuples is compared and functional dependency disablers are detected.
- 30: Each unused attribute x having equal value in both tuples is considered.
- 31–32: If another unused attribute y is either empty in one of the tuples or their values are different ...
- 33–...: then x is not functionally dependent on y since there exist distinct values of y for one value of x .

5.2.3 Examples

This section contains examples of particular data and their processing. The examples use the group base defined in Table 1. This data set is a very simplified real-world excerpt. Nevertheless, using this data the important properties of the extraction algorithm can be demonstrated.

There are 4 attributes used – ‘D’, ‘P’, ‘R’ and ‘S’. Their real-world meaning is the education level (Bachelor/Master), the field of study (Informatics/Mathematics), the year of study and specialization (Programming/Computer Science/Software Systems), although it is irrelevant for the algorithm.

Example 5. Branch tree = {B, -, -, -}

	D	P	R	S		D	P	R	S
D	1	0	0	0	D	1	0	0	0
P	0	0	1	1	P	0	0	0	1
R	0	1	0	1	R	0	0	0	0
S	0	1	1	0	S	0	0	0	0

Table 2. Dependency matrix for {B, -, -, -}

Table 2 contains the computed dependency matrix in two versions – the left matrix is produced by the initialization phase of the algorithm, the right matrix is a final output of the algorithm.

In this example the branch tree consists of one attribute D with value of ‘B’. The first phase detects the following facts:

- D is excluded since it is already contained in the branch tree
- P, R and S attributes are not used and nonempty; they may have functional dependencies

The second phase erases those functional dependency candidates that have no data dependency in matching tuples. E.g., the tuple pair $i = \{B, I, 1, -\}$, $j = \{B, I, -, -\}$ and attribute pair $ax = P$, $ay = R$ induce that P is not functionally dependent on R since there exist distinct values of R for one value of P, so that

$dp[P, R]$ is erased. Similarly the tuple pair $i = \{B, I, 1, IPR\}$, $j = \{B, I, -, -\}$ induces that P is not functionally dependent on S.

Finally, $S \rightsquigarrow P$, $D \rightsquigarrow D$, the D and S attributes are excluded from the next level; the resultset (set of allowed attributes) = $\{P, R\}$.

Example 6. Branch tree = $\{N, -, -, -\}$

Table 3 displays the result of using another value of the attribute D. There are no ‘M’ values of the attribute P and no values of the attribute S at all in the matching tuple set. It excludes the S attribute from the resultset in the first phase and the dependency $R \rightsquigarrow P$ in the second phase. Since the attribute P is constant within the matching tuple set, the final resultset = $\{R\}$.

	D	P	R	S		D	P	R	S
D	1	0	0	0	D	1	0	0	0
P	0	0	1	0	P	0	0	1	0
R	0	1	0	0	R	0	0	0	0
S	0	1	1	1	S	0	0	0	1

Table 3. Dependency matrix for $\{N, -, -, -\}$

Example 7. Branch tree = $\{-, -, -, -\}$

If the branch tree is empty, the whole tuple set is processed for detection of the first level attributes. During the first phase, all diagonal values are erased since no attribute is used. All nondiagonal values are set since all pairs of attributes have a nonempty value in at least one tuple; all attributes are functional dependency candidates.

The second phase removes all dependency disablers. The resulting matrix shows three remaining dependencies: $R \rightsquigarrow D$, $S \rightsquigarrow D$, $S \rightsquigarrow P$. The final resultset is $\{D, P\}$; these attributes may be used at the first level.

	D	P	R	S		D	P	R	S
D	0	1	1	1	D	0	0	1	1
P	1	0	1	1	P	0	0	0	1
R	1	1	0	1	R	0	0	0	0
S	1	1	1	0	S	0	0	0	0

Table 4. Dependency matrix for the empty tree

6 VISUALIZATION

The theoretical background and the algorithms described in the previous sections have been implemented in the MetroNG system, which consists of two complimentary applications. The web interface offers access to the schedule namely for students and teachers; this web application will not be discussed further in this article. The

second application is the MetroNG client application intended for creators of the schedule.

6.1 Application Modes

MetroNG supports several application modes; each mode de facto displays several dimensions of the data and it is tailored for a specific class of events. The most common type of events is a regular event being held every week of the semester at the same time. The regular modes display these regular events using a days-of-the-week time dimension as the X axis. These modes are used most of the time and an example is displayed in Figure 1. Week-oriented modes are used for irregular events. These modes allow the users to display schedule for each week individually.

Another type of events are block-oriented lectures; all working days in one week are dedicated to a single lecture for a particular group of students. This type of events is commonly used for practices and clinical education. The block mode supports this type of events by using the weeks of the semester as the X axis; while each week is displayed as a single column.

Moreover, there are three special-purpose application modes used for maintenance work and a lot of other events in the system, e.g., room reservations not related to lectures, students' busy time, teachers' preferences, etc.

6.2 Display Areas and Axes

Figure 1 displays the principal areas of the main application window. The text-oriented part on the left contains a sorted and filtered list of events together with their most important data fields. It is possible to directly schedule or reschedule these events by dragging them to the graphical grid.

The graphical part displays events in a grid-like way. All views share the same horizontal axis, but each view has a different vertical axis, usually rooms, student groups, and teachers. Although these areas are used in all regular modes, their content is dependent on the horizontal axis bound to a particular mode. These grid areas are complemented by a detail area at the bottom that displays additional information on any object including overlapping or colliding events.

The areas are bound to vertical and horizontal axes; there are 10 different linear and tree-shaped axes in MetroNG and their combinations make possible to display the data in the most useful way for particular tasks.

The tree axes hierarchically organize the main scheduling entities – e.g. rooms, students, teachers, and courses. The extraction algorithm (Section 5.2) is useful especially in the students' axis since the structure is often very complex, irregular and the data are too extensive to be displayed at once in an unstructured or regular way. A small slice of the real-world students' axis is depicted in Figure 2. Nevertheless, the algorithms are used for other tree axes in the same way; there is no special adjustment for student groups. The most valuable data (e.g., a classroom for the

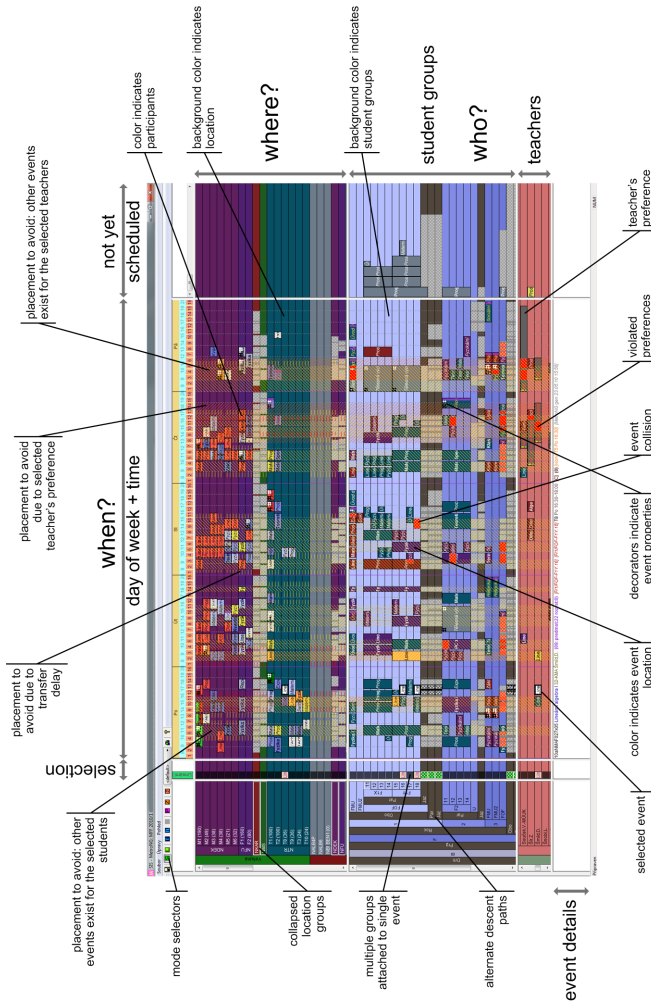


Figure 1. Principal MetroNG Areas

students in particular time, possible conflicts of location, etc.) during the whole scheduling process is then displayed in the intersections of these axes.

6.3 Collisions and Decorators

One of the most valuable feature of MetroNG is the detection of collisions and the prevention of collisions during the scheduling process. Both types of the collisions are implemented using the Collision Detection algorithm (Section 5.1). Existing collisions are displayed as hatched rectangles, so that the user can immediately

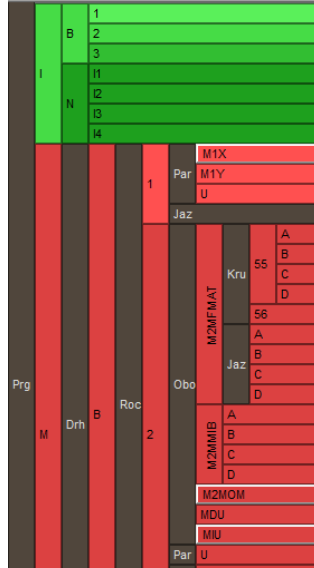


Figure 2. Students' axis

see an existing collisions (see Figure 3 and Figure 4). The figure also shows other *decorators* used to display important information about the events (e.g., status flags, event lock, etc).

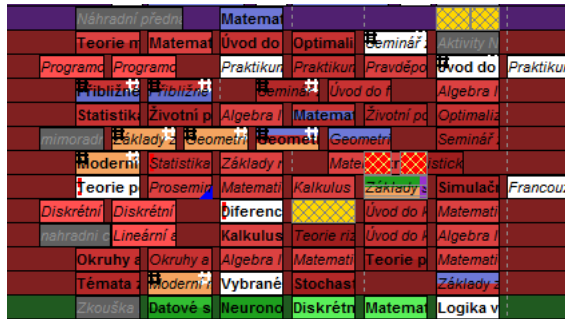


Figure 3. Event Decorators

Other important decorator is the color used as the background of the rectangle. Its meaning depends on the area where the rectangle is displayed. In the students area, the color of the rectangles is the same as the color of the building they are scheduled to. In the room area, there is the same color as the student groups that attend the event. Note that in Figure 1 in the top (room) section of the view the color of the background (color of the building) very often

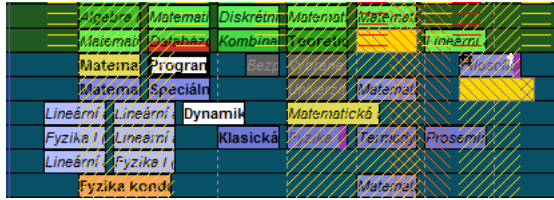


Figure 4. Collisions

matches the color of the rectangle. This is an intentional side effect of this principle.

So far, we described graphical elements that help the user see the current state of the schedule. However, there is another important group – features that allow the user to see possible effects of his or her actions (of changing the schedule). For an example see Figure 4. When an event is selected, some parts of the view are covered with hatching. It displays the areas where there is a potential collision between the selected event and other events – that is, if the event is scheduled to that time and room, it would create a particular collision.

7 CONCLUSIONS

The main contributions of this paper can be summed up as follows:

- formal model of events and visitors
- algorithms that efficiently detect event collisions and compute a structure of user navigation in such multidimensional data without explicit specification of the navigation structure
- the MetroNG application that implements the formal model and proposed algorithms; the application supports the whole process of creating a complex university and curricula schedule.

The formal model for the complex collision detection system is used to identify situations where a group of students should attend two classes scheduled for the same time. Using the model and algorithms, there is no need for the users to explicitly describe the navigation structure of the underlying data; the structure is computed automatically from the used data-set (group base) and its current state (tuple traces). The algorithms are successfully implemented in the MetroNG scheduling tool which is currently used in real-life by some of the largest universities in the Czech Republic for modeling complicated large-size schedules – e.g., at the Charles University in Prague, there are 53 000 students, 6 000 teachers, 650 study programs and 40 000 scheduling events. The real-life experience of the users shows that the presented methods are able to solve their task efficiently.

In our future work, we intend to formalize the user requirements and the quality of the resulting output and compare our solution to automatic schedulers.

Acknowledgment

This work was supported by the Grant Agency of the Czech Republic, grant number SVV-2013-267312, GACR 204/13/08195 and PRVOUK P46.

REFERENCES

- [1] BEDNAREK, D.—DOKULIL, J.—YAGHOB, J.—ZAVORAL, F.: MetroNG: Multimodal Interactive Scheduling Interface. Proceedings of the International Conference on Advanced Visual Interfaces, ACM, 2010, pp. 317–320.
- [2] CARPINETO, C.—ROMANO, G.: Concept Data Analysis: Theory and Applications. Wiley, 2004, ISBN 978-0-470-85055-8.
- [3] GANTER, B.—WILLE, R.: Contextual Attribute Logic. In: Tepfenhart, W., Cyre, W. (Eds.), Conceptual Structures: Standards and Practices. LNAI, 1999, Vol. 1640, pp. 377–388.
- [4] GANTER, B.: Contextual Attribute Logic of Many-Valued Attributes. In: Carbonell, J. G., Siekmann, J. (Eds.), Formal Concept Analysis. LNCS, 2005, Vol. 3626, pp. 101–103.
- [5] BIRKHOFF, G.: Lattice Theory. American Mathematical Soc., 1984.
- [6] SEHWAN, Y.—JONGDAE, J.—DAE RYONG, K.: Self Conflict Resolving Interactive Web-Based Class Scheduling System. Academy of Information and Management Sciences Journal, Vol. 8, 2005, No. 2, pp. 69–78.
- [7] Visual Classroom Scheduler, Visual Scheduling Systems, 2001, <http://www.vss.com.au/index.asp>.
- [8] Class Scheduler, Cyber Matrix, 2009, http://www.cybermatrix.com/class_scheduler.html.
- [9] Lantiv Timetabler, Lantiv, 2009, <http://www.lantiv.com/>.
- [10] POTHITOS, N.—STAMATOPOULOS, P.—ZERVOUDAKIS, K.: Course Scheduling in an Adjustable Constraint Propagation Schema. 2012 IEEE 24th International Conference on Tools with Artificial Intelligence (ICTAI), 2012, Vol. 1, pp. 335–343, DOI: 10.1109/ICTAI.2012.53.
- [11] DASGUPTA, P.—KHAZANCHI, D.: Adaptive Decision Support for Academic Course Scheduling Using Intelligent Software Agents. International Journal of Technology in Teaching and Learning, Vol. 1, 2005, No. 2, pp. 63–78.
- [12] SALTZMAN, R.: An Optimization Model for Scheduling Classes in a Business School Department. Journal of Operations Management, Vol. 7, 2009, No. 1, pp. 84–92.
- [13] KINGSTON, J. H.: The KTS High School Timetabling System. The 6th International Conference on Practice and Theory of Automated Timetabling (PATAT), 2006, pp. 181–195.
- [14] TSANG, E.: A Glimpse of Constraint Satisfaction. Artificial Intelligence Review, Vol. 13, 1999, No. 3, pp. 215–227.

- [15] CARTER, M. W.: A Comprehensive Course Timetabling and Student Scheduling System at the University of Waterloo. Practice and Theory of Automated Timetabling III. LNCS, 2001, Vol. 2079, pp. 64–82.
- [16] MATHAISEL, D. F. X.—COMM, C. L.: Course and Classroom Scheduling: An Interactive Computer Graphics Approach. Journal of Systems and Software, Vol. 15, 1991, Issue 2, pp. 149–157, ISSN 0164–1212.
- [17] ABDENNADHER, S.—MARTE, M.: University Course Timetabling Using Constraint Handling Rules. Applied Artificial Intelligence, Vol. 14, 2000, No. 4, pp. 311–325.
- [18] BURKE, E.—BYKOV, Y.—PETROVIC, S.: A Multicriteria Approach to Examination Timetabling. LNCS, 2001, Vol. 2079, pp. 118–131.
- [19] DIMOPOULOU, M.—MILIOTIS, P.: Implementation of a University Course and Examination Timetabling System. European Journal of Operational Research, Vol. 130, 2001, No. 1, pp. 202–213.
- [20] RUDOVÁ, H.—MURRAY, K.: University Course Timetabling with Soft Constraints. LNCS, 2003, Vol. 2740, pp. 310–328.
- [21] BURKE, E. K.—MACCARTHY, B.—PETROVIC, S.—QU, R.: Case-Based Reasoning in Course Timetabling: An Attribute Graph Approach. LNCS, 2001, Vol. 2080, pp. 90–104.
- [22] BURKE, E. K.—NEWALL, J. P.: Solving Examination Timetabling Problems Through Adaption of Heuristic Orderings. Annals of Operations Research, Vol. 129, 2004, No. 1-4, pp. 107–134.
- [23] SCHAEFER, A.—MEISELS, A.: Solving Employee Timetabling Problems by Generalized Local Search. LNCS, 2000, Vol. 1792, pp. 380–389.
- [24] BURKE, E.—BYKOV, Y.—NEWALL, J.—PETROVIC, S.: A Time-predefined Local Search Approach to Exam Timetabling Problems. IIE Transactions, Vol. 36, 2004, No. 6, pp. 509–528.
- [25] ROSS, P.—HART, E.—CORNE, D.: Genetic Algorithms and Timetabling. Natural Computing Series, Advances in Evolutionary Computing: Theory and Applications, 2003, pp. 755–777.
- [26] BELIGIANNIS, G. N.—MOSCHOPOULOS, C. N.—KAPERONIS, G. P.—LIKOTHANASSIS, S. D.: Applying Evolutionary Computation to the School Timetabling Problem: The Greek Case. Computers and Operations Research, Vol. 35, 2008, No. 4, pp. 1265–1280.
- [27] NEDJAH, N.—DE MACEDO MOURELLE, L.: Evolutionary Time Scheduling. International Conference on Information Technology, Coding and Computing (ITCC '04), Vol. 2, 2004, pp. 357–361.
- [28] QAROUNI-FARD, D.—NAJAFI-ARDABILI, A.—MOEINZADEH, M.-H.: Finding Feasible Timetables with Particle Swarm Optimization. 4th International Conference on Innovations in Information Technology, 2007, pp. 387–391.
- [29] CHU, S.-C.—CHEN, Y.-T.—HO, J.-H.: Timetabling Scheduling Using Particle Swarm Optimization. 1st International Conference on Innovative Computing, Information and Control, 2006, pp. 324–327.
- [30] SOCHA, K.—KNOWLES, J.—SAMPOLS, M.: A MAX-MIN Ant System for the University Course Timetabling Problem. LNCS, 2002, Vol. 2463, pp. 1–13.

- [31] DI GASPERO, L.—SCHAERF, A.: Tabu Search Techniques for Examination Timetabling. LNCS, 2001, Vol. 2079, pp. 104–117.
- [32] BURKE, E. K.—KENDALL, G.—SOUBEIGA, E.: A Tabu-Search Hyperheuristic for Timetabling and Rostering. *Journal of Heuristics*, Vol. 9, 2003, No. 6, pp. 451–470.
- [33] ROSSI-DORIA, O.—SAMPELS, M.—BIRATTARI, M.—CHIARANDINI, M.—DORIGO, M.—GAMBARDELLA, L. M. et al.: A Comparison of the Performance of Different Metaheuristics on the Timetabling Problem. LNCS, 2003, Vol. 2740, pp. 329–351.
- [34] BILGIN, B.—ÖZCAN, E.—KORKMAZ, E. E.: An Experimental Study on Hyper-Heuristics and Exam Timetabling. 6th International Conference on the Practice and Theory of Automated Timetabling, 2006, pp. 123–140.
- [35] BURKE, E. K.—MCCOLLUM, B.—MEISELS, A.—PETROVIC, S.—QU, R.: A Graph-Based Hyper Heuristic for Educational Timetabling Problems. *European Journal of Operational Research*, Vol. 176, 2007, pp. 177–192.
- [36] BURKE, E. K.—PETROVIC, S.: Recent Research Directions in Automated Timetabling. *European Journal of Operational Research*, Vol. 140, 2002, pp. 266–280.
- [37] ADRIAEN, M.—DE CAUSMAECKER, P.—DEMEESTER, P.—BERGHE, G. V.: Tackling the University Course Timetabling Problem with an Aggregation Approach. 6th International Conference on the Practice and Theory of Automated Timetabling, 2006, pp. 330–335.
- [38] MALIM, M. R.—KHADER, A. T.—MUSTAFA, A.: Artificial Immune Algorithms for University Timetabling. 6th International Conference on the Practice and Theory of Automated Timetabling, 2006, pp. 234–245.
- [39] PERZINA, R.: Solving the University Timetabling Problem with Optimized Enrolment of Students by a Parallel Self-Adaptive Genetic Algorithm. 6th International Conference on the Practice and Theory of Automated Timetabling, 2006, pp. 264–280.
- [40] TEN EIKELDER, H. M. M.—WILLEMEN, R. J.: Some Complexity Aspects of Secondary School Timetabling Problems. LNCS, 2001, Vol. 2079, pp. 18–27.
- [41] DE HAAN, P.—LANDMAN, R.—POST, G.—RUIZENAAR, H.: A Four-Phase Approach to a Timetabling Problem in Secondary Schools. 6th International Conference on the Practice and Theory of Automated Timetabling, 2006, pp. 423–425.
- [42] JACOBSEN, F.—BORTFELDT, A.—GEHRING, H.: Timetabling at German Secondary Schools: Tabu Search Versus Constraint Programming. 6th International Conference on the Practice and Theory of Automated Timetabling, 2006, pp. 439–442.
- [43] KINGSTON, J. H.: The KTS School Timetabling System. 6th International Conference on the Practice and Theory of Automated Timetabling, 2006, pp. 181–195.
- [44] COOPER, T. B.—KINGSTON, J. H.: The Complexity of Timetable Construction Problems. TR No. 495, Basser Department of Computer Science, The University of Sidney, 1995.



David BEDNÁREK received his Ph. D. in informatics from Charles University in Prague in 2009. His research interests include programming languages, compiler construction, parallel programming, and database systems. He was involved in many national and international research projects. He served as member of the program committee of several international conferences.



Jakub YAGHOB is currently associated with the Charles University in Prague, Faculty of Mathematics and Physics. He graduated consequently at the Charles University in Prague (1991 – M.Sc., 2003 – Ph.D.). He is responsible supervisor for numerous development grants of the Ministry of Education and FRV grants (e.g. Environment for teaching parallel programming). He was also working on numerous GAR grants (Highly Scalable Parallel and Distributed Methods of Data Processing in e-Science amongst the others). He is a member of a few international program committees of various conferences and workshops taking

the role of the program or organization committee chair several times.



Filip ZAVORAL is the vice-head of the Department of Software Engineering, Charles University in Prague, Czech Republic. His research interests include distributed and parallel technologies, cloud computing and efficient data processing. He was involved in many national and international research projects. He has co-authored more than 60 research publications such as: journal papers, conference proceedings papers, book chapters, and editorials of journal special issues. He is member of the editorial board of several international journals and served as member of the program or organizing committee of many international conferences.

Chapter 9.

Locality Aware Task Scheduling in Parallel Data Stream Processing

Zbyněk Falt, Martin Kruliš, David Bednárek, Jakub Yaghob, Filip Zavoral

Proceedings of the 8th International Symposium on Intelligent Distributed Computing - IDC'2014, Springer Verlag, pp. 331-342, 2014

Studies in Computational Intelligence 570

David Camacho
Lars Braubach
Salvatore Venticinque
Costin Badica *Editors*

Intelligent Distributed Computing VIII

 Springer

Chapter 1

Locality Aware Task Scheduling in Parallel Data Stream Processing

Zbyněk Falt, Martin Kruliš, David Bednárek, Jakub Yaghob, Filip Zavoral

Abstract Parallel data processing and parallel streaming systems become quite popular. They are employed in various domains such as real-time signal processing, OLAP database systems, or high performance data extraction. One of the key components of these systems is the task scheduler which plans and executes tasks spawned by the system on available CPU cores. The multiprocessor systems and CPU architecture of the day become quite complex, which makes the task scheduling a challenging problem. In this paper, we propose a novel task scheduling strategy for parallel data stream systems, that reflects many technical issues of the current hardware. We were able to achieve up to $3\times$ speed up on a NUMA system and up to 10% speed up on an older SMP system with respect to the unoptimized version of the scheduler. The basic ideas implemented in our scheduler may be adopted for task schedulers that focus on other priorities or employ different constraints.

1.1 Introduction

Parallel processing is becoming increasingly important in high performance systems, since the hardware architectures have embraced concurrent execution to increase their computational power. Unfortunately, parallel programming is much more difficult and error prone, since the programmers are used to think and express their intentions in serial manner. Many different paradigms and concepts have been devised to simplify the design of concurrent processing.

One of these approaches is stream data processing. It was originally designed for systems that process data which are generated in real-time and

Parallel Architectures/Applications/Algorithms Research Group
Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic e-mail:
falt,krulis,bednarek,yaghob,zavoral@ksi.mff.cuni.cz

need to be processed immediately, but they also have been adopted in database systems and parallel systems, since they simplify the application design and naturally reveal opportunities for parallelization.

A streaming application is usually expressed as an oriented graph (also denoted as *execution plan*), where the vertices are processing stages (operators or filters) that process the data and the edges prescribe how the data are passed on between these stages. The main advantage from the perspective of parallel processing is that each stage contains serial code (which is easy to design) and multiple stages may be executed concurrently.

One of the systems that implements this idea is Bobox [5]. The main objective of Bobox is to process semantic and semi-structured data effectively [6]. It currently supports the SPARQL [18] query language and partially the XQuery [8] and the TriQuery [4] language. One of the most challenging problems of this system is to effectively and efficiently execute the work of the operators on the available CPU cores.

In this paper, we propose a novel locality aware task scheduling strategy (called LAS) for data streaming systems. This strategy incorporates important hardware factors such as cache hierarchies and non-uniform memory architectures (NUMA). We have implemented this strategy in the Bobox task scheduler and achieved significant speedup on modern host systems. Although our performance analysis was conducted using Bobox, the scheduler itself can be easily adopted for other streaming systems as well.

The paper is organized as follows. Section 1.2 revise the most important facts regarding state-of-the-art CPU architectures and NUMA systems. Our LAS scheduler is described in Section 1.3. Section 1.4 presents the experimental results that evaluate the benefits of our innovations. The related work is revised in Section 1.5 and Section 1.6 concludes the paper.

1.2 CPU Fundamentals and Task Scheduling

In this section, we revise fundamental facts regarding the architectures of modern multi-core CPUs and NUMA systems. We also put these facts in the perspective of task scheduling which is often employed to achieve parallel data processing in complex systems.

1.2.1 CPU Architecture

The CPU architectures became quite complex in the past few decades. We will focus solely on the properties, which directly affect the parallel execution of tasks that cooperate via shared memory. A generic schema of modern multi-core CPU is presented in Figure 1.1.

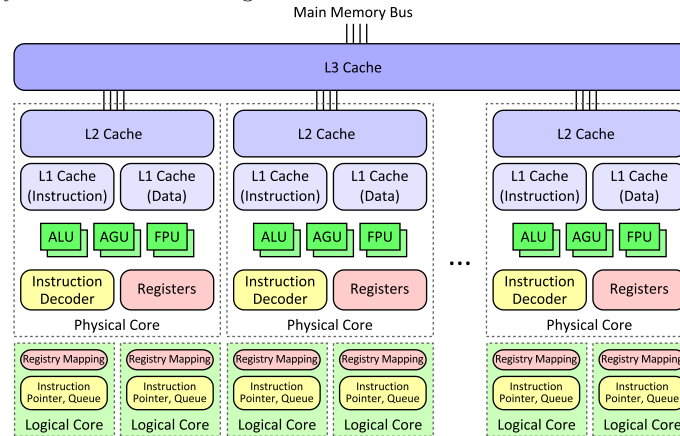


Fig. 1.1: A schema of multi-core CPU

The CPU comprises several physical cores which are quite independent. These cores usually share only the memory controller and sometimes certain levels of cache. The physical cores are often divided into two logical cores by means of hyper threading (Intel) or dual-core modules (AMD) technology. The logical cores share also some computational units and also the lowest level of cache. In the remainder of the paper, we will use the term CPU core to denote logical cores – i.e., the lowest computational unit of the CPU which processes one thread at a time.

Multiprocessor configurations combine several CPUs into one system. Older CPUs, which does not have integrated memory controllers are connected in a similar way as the physical cores in the CPU die. This configuration is called symmetric multiprocessing (SMP). Newer CPUs have memory controllers integrated, thus each CPU manages part of the system memory. This configuration is called non-uniform memory architecture (NUMA), since the memory latency depends on whether it is directly connected to the CPU who uses it, or whether it needs to be accessed via a controller of another CPU.

The organization of the cores within the CPU and organization of the CPU chips within a NUMA system forms a hierarchy. Logical cores of one physical core or physical cores that share memory cache are considered close, while cores on two different CPU chips (i.e., in two different NUMA nodes) are considered distant. This hierarchy plays significant role in task planning. Related or cooperating tasks should be scheduled on cores that are close by, since they will likely benefit from cache sharing. Completely unrelated tasks using different portions of the main memory should be scheduled on different NUMA nodes, so they can keep their intermediate data in different caches and in different memory nodes.

1.2.2 Task Scheduling

In order to achieve parallelism on modern CPUs, the work needs to be divided into portions that can be processed concurrently. Traditional division into threads is too coarse and tedious, hence most of the parallel systems deal with tasks. The *task* comprises both the data and the procedure that process the data. Tasks are scheduled and processed by available CPU cores. It has been established [19] that the tasks can be effectively employed in the implementation of more complex parallel patterns such as parallel loops, reduction, pipeline, or data stream processing.

In this work, we focus solely on systems where the tasks are generated dynamically by other tasks or by external events (e.g., user requests). Such systems must employ dynamic scheduling, which can cope with the ever changing situation. The dynamic scheduler manages the tasks which are ready to run and when to assign them to the available CPU cores as they become available.

Furthermore, task schedulers often employ some form of restrictions for implicit synchronization like task dependencies. When a task is spawned, it may not be ready to execute immediately. In such case, the task scheduler needs to manage *waiting* tasks along with the *ready* tasks. When a waiting task conditions for execution are met, the scheduler change its state to *ready* and eventually assigns it to an available CPU core. However, we are focusing on improving efficiency of the task scheduling, thus we will not consider the waiting tasks nor any mechanisms for automatic testing the task readiness. Henceforth, we use the term *task spawning* for introducing a ready tasks to the scheduler.

1.3 Locality Aware Task Scheduler

The task scheduler manages tasks in the system and process them on the available computational units. Different task schedulers may be used for different systems. In our work, we address the problems of parallel data processing, such as problems of database management systems. Hence, our objective is to design a task scheduler that reflects three important issues:

- Even though the overall work is orchestrated by some form of an execution plan, the interpretation of the plan is data dependent, thus the tasks are spawned dynamically.
- The tasks should be planed with respect to overall throughput of the system, since they usually work on a complex problem which needs to be solved as whole.
- The available hardware resources should be utilized efficiently.

The last issue is becoming increasingly important as the CPU architectures are getting more complex with each new generation. Planning the tasks in

a way that considers which data are hot in caches or that better organizes the work among NUMA nodes is the key to achieving much better overall performance.

In order to achieve better results, we have improved the definition of a task, so the programmer of the system that employs our scheduler can pass on some explicit information which can be used for scheduling. First of all, we distinguish two types of tasks [7] and this type is specified when the task is spawned:

- The *immediate task* represent work that immediately relates to the task being currently processed. This type of tasks is expected to be executed as soon as possible and preferably close to the task that spawned them to utilize data which are still hot in the cache.
- The *deferred task* represent work that is not closely related to the task being currently processed. This type of tasks is also expected to generate more sub tasks eventually.

Furthermore, every task (both immediate and deferred) is attached to a *request* which corresponds to a larger portion of work that is divided into tasks to achieve parallelism (e.g., it can be related to a database query). Requests are uniquely identified by a *request ID*, which is a sequentially assigned number. A task inherits its request ID from the task which spawned it.

1.3.1 Task Scheduling Strategy

The initialization process of the task scheduler scans the host system and detects the configuration and properties of the CPUs. CPU cores which share at least one level of cache are bundled together in logical *core groups* and a *thread pool* is created for each group. The thread pool has one thread for each CPU core in the corresponding group and the threads have their affinity set to this core group. The thread can easily determine the associated CPU core using appropriate operating system functions.

Each core group maintains one queue of immediate tasks per core and one shared queue of deferred tasks. The deferred queue is in fact more complex data structure than a simple queue, which maintains the task of each request separately. It also provides quick access and extraction of the youngest and the oldest tasks from the oldest and second oldest request.

The main paradigm employed in the LAS scheduling strategy is to emphasize data locality awareness. Therefore, when the scheduler assigns another work to a thread, it attempts to select a task which is as close as possible to the previous work done by that thread. For this purpose, we define the distance between two cores within one group and the distance between two core groups. Distance of cores within one group is equal to the lowest level of cache these two cores are sharing. Distance of two core groups is equal to

the distance of their corresponding NUMA nodes (and zero for groups that share a NUMA node).

The distance between cores and core groups determine our scheduling strategy. When a thread completes a task it executes the scheduling algorithm to fetch another task to execute. The first applicable rule of the following list is taken:

1. The youngest task from the queue of immediate tasks of the current core.
2. Other cores of the same group are scanned (in the increasing distance) and the first non-empty immediate queue is found. If such queue exists, its oldest task is taken.
3. The youngest deferred task of the oldest request from the deferred task queue of the current group is taken. This rule ensures that all threads of one core group work on the same (the oldest) request if possible.
4. Other core groups are scanned (in increasing distance) and the first non-empty deferred queue is found. If such queue exists, the oldest deferred task of the **second** oldest request is taken. If the queue has tasks of only one request, its oldest task is taken instead. This strategy assumes that a core group is heavily engaged in the processing of the oldest request and it would not be wise to disrupt this work when another request is available. However, this algorithm does not prevent the situation that the whole system cooperates on one common request.
5. Immediate queues of cores from other groups which are located on the same NUMA node¹ are scanned. If non-empty queue is found, its oldest task is taken. The immediate queues are scanned in round robin manner and the thread remembers the last non-empty queue found. When this rule is applied again, the scan is resumed where it previously ended. This rule enforces that all immediate tasks are processed on the same NUMA node where they were spawned.

If all steps fail (i.e., there is no available task to execute), the thread is suspended, so it will not consume system resources. The whole algorithm and the thread group hierarchy is depicted in Figure 1.2.

When a thread spawns a tasks, it first determines on which CPU core it runs. The immediate tasks are inserted to the immediate queue of the core. The deferred tasks are inserted in the shared queue of deferred tasks of the corresponding core group.

¹ Note that no such group may exist when exactly one group is assigned to each NUMA node.

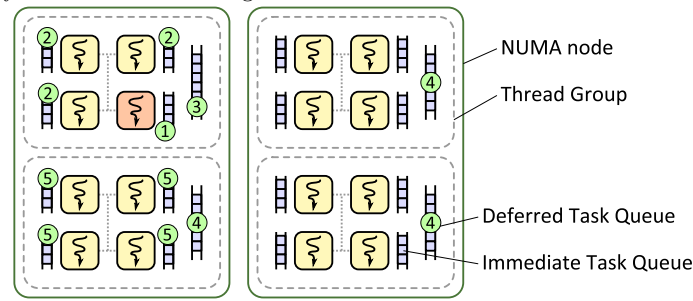


Fig. 1.2: The schema of a task scheduler

1.3.2 Resuming Suspended Threads

When a thread does not have another task to process, the thread suspends itself on a synchronization primitive². Each group has one such synchronization primitive and the suspended threads are added to its waiting queue.

When a new task is spawned, the spawning thread attempts to wake one of the suspended threads. First, it tries to wake a thread in the same thread pool (the same group of cores). If the whole group is working, it scans all other groups and attempts to wake a thread there. The groups are scanned in an increasing distance from the original group and the search finishes when a group with suspended thread is found or all groups are scanned.

The immediate tasks and deferred tasks are handled slightly differently in this case. When an immediate task is spawned, the search for a group with suspended thread ends at the boundary of the NUMA node. There is no need to wake threads on other NUMA nodes since the scheduling rules prevent the immediate tasks to travel between NUMA nodes. When deferred tasks is spawned, all groups are tested.

1.4 Experiments

We performed several experiments to prove that the LAS scheduling algorithm significantly improved performance of the system. For the testing, we used our parallel implementation of the in-memory SPARQL engine [14] and the SP²Bench benchmark [21] and its 5m testing dataset. The implementation of the engine is able to generate parallel execution plans without significant serial bottlenecks, i.e., all worker threads are utilized during their evaluation. Additionally, the SP²Bench benchmark contains several queries which generate various and really complex query execution plans. This is profitable

² Current implementation uses standard semaphore and atomic operations that handles related metadata.

since this variety shows the behaviour of the task scheduler under various circumstances.

We selected queries Q2, Q4, Q5a, Q6, Q7 Q8, Q9 and Q11 from the benchmark, since they take reasonable time to evaluate and their query execution plans are complex enough. Other queries are evaluated so fast that the results are negligible.

We used two hardware configurations for the experiments:

- A server with two Intel Xeon E5310 processors, both running at 1.60Ghz. This type of processor has 4 cores and two shared 4MB L2 caches. First two cores share the first L2 cache, second two cores share the other. Additionally, each core has its own L1 cache (32kB + 32kB). This configuration represents non-trivial SMP system and our scheduling strategy creates 4 thread pools for this configuration.
- A NUMA server with four Intel Xeon E7-4820 processors, all running at 2.0Ghz. This type of processor has physical 8 cores with Hyper-Threading Technology, i.e., the processor has 16 logical cores in total. Each physical core has its own L1 cache (32kB + 32kB), L2 cache (256kB) and all cores share one L3 cache (18MB). This configuration represents non-trivial NUMA system and our scheduling strategy creates 4 thread pools as well.

We performed two different experiments for each hardware configuration:

- We run the selected query just once. This experiment demonstrate the situation when there is a lot of various data dependencies among the tasks, since all tasks belong to one query.
- We run the selected query multiple times in parallel (16 times in all measurements). This experiment demonstrate the situation when there is a lot of tasks which do not have any dependency on each other, i.e., each thread can process its own instance of the query without any cooperation with the others.

Finally, we performed two different measurements for each experiment:

- We used the scheduler which implements the LAS scheduling strategy described in the Section 1.3.
- We used the scheduler which implements the strategy which is close to the strategy used in TBB or in our previous work [7]. Each thread keeps its local queue of immediate task and all threads share one queue of deferred tasks. Thread executes the first existing task in this order: the latest immediate task from its local queue, the oldest deferred task from the shared queue and the oldest immediate task from the local queue of another thread. In other words, immediate tasks are handled in the same manner as the spawned tasks in TBB [2] and deferred tasks are handled in the same manner as the enqueued tasks in TBB. We denote this scheduler as *NLS*.

1.4.1 SMP system

The results are shown in Figure 1.3 for single query and in Figure 1.4 for multiple parallel queries. Notice that for Q6 only 1m dataset was used so that the evaluation takes reasonable time. In multiple queries, we used 1m dataset for Q4 and Q8 in order to avoid swapping of the operating memory. Additionally, we used only 250k dataset for Q6 from the same reason as in the single query.

1.4.1.1 Single query

As expected, on SMP system the NLS scheduling algorithm performs quite well. However, queries Q4 and Q6 benefits from the LAS and especially query Q2 is almost $2\times$ faster. This query consist of one long pipeline, therefore, it is especially sensitive to the data locality. Other queries contains such pipelines as well, however, these pipelines are typically split to multiple independent parts because of sorting operators which break up the pipeline processing and cause that the execution plans are evaluated by parts, i.e., that all threads cooperate close to each other.

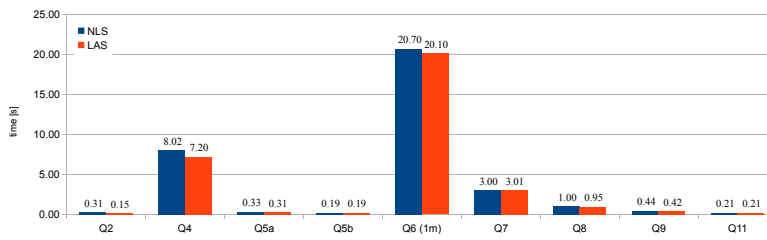


Fig. 1.3: Single query on SMP

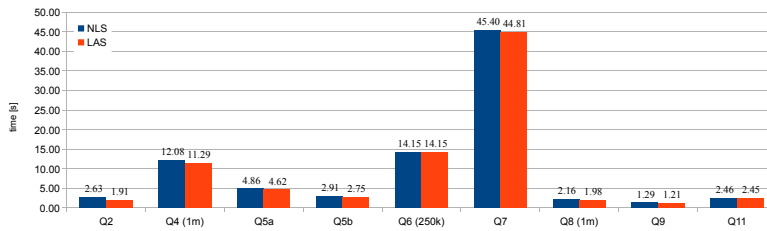


Fig. 1.4: Multiple queries on SMP

1.4.1.2 Multiple queries

The second experiment shows that the LAS performs better than NLS and in more cases than in the first experiment. The main reason is that the LAS better separates individual requests, i.e., the requests do not force out each other from cache memory.

1.4.2 NUMA system

Both experiments on the NUMA system (see Figure 1.5 and Figure 1.6) proves that taking the NUMA factor into account is very important in modern systems. The main problem is that accessing memory of another node slows down both communicating nodes and the system bus.

The LAS tries to keep one request on one NUMA node as much as it is possible. If it is not possible, it tries to keep different branches of execution plans on different NUMA nodes which minimizes data interference between the NUMA nodes. The NLS does not distinguish among the NUMA nodes, therefore, the relationship between a thread and the memory being accessed is almost arbitrarily. This is significant especially in the multiple queries.

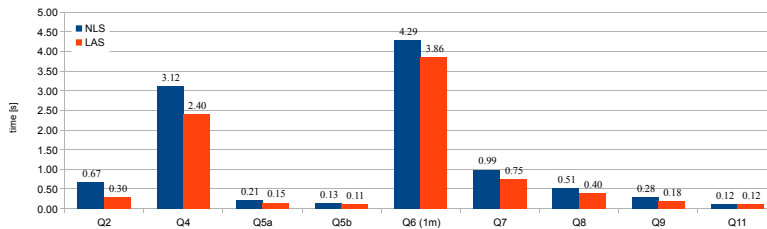


Fig. 1.5: Single query on NUMA

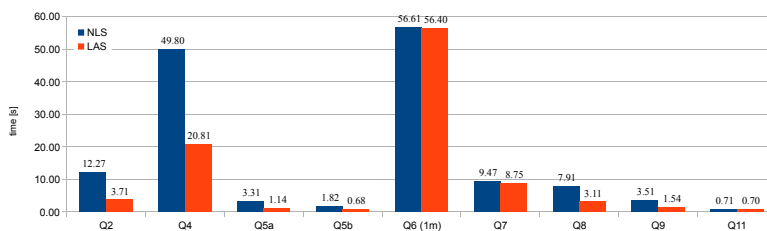


Fig. 1.6: Multiple queries on NUMA

All performed experiments proved that the locality awareness has a significant impact on the performance of the parallel streaming system running on either SMP or NUMA systems.

1.5 Related Work

As we already mentioned in Section 1.2, modern architectures became complex and complicated. Thus, optimizing the performance of applications through elaborate task scheduling strategies is a challenging task and a very hot topic in current research. The fact that finding an optimal scheduling plan is NP-hard problem causes that all scheduling strategies just try to find a suboptimal solution using heuristics and approximation techniques [22].

In streaming systems, there are several aspects of scheduling optimization, such as memory usage [3], cache-efficiency [12], response time, throughput, etc. or their mutual combinations [15, 20].

In this work, we relaxed many aspects and just tried to maximize data locality in order to increase the performance of the system. This allowed us to adopt techniques used in non-streaming systems. Our previous work [7] was the first step. We showed that data flow awareness (i.e., using immediate and deferred tasks) in streaming systems increase data locality; however, the scheduling algorithm lacks the support of NUMA and non-trivial SMP systems.

These issues are successfully solved in several works, e.g., in popular parallel frameworks such as OpenMP [13, 9] or Intel Threading Building Blocks [17, 2].

The division of tasks to immediate and deferred tasks ensures that threads work with data hot in cache if they have its own immediate tasks. However, it showed up, that the bottleneck of the system is the task stealing since the tasks were stolen from a randomly chosen thread. However, this is an issue of the cited papers as well.

The task stealing optimization is researched thoroughly in work by Chen et. al. [11, 10]. In fact, the algorithm CATS/CAB from this work is similar to our LAS algorithm described in Section 1.3; however, there are several differences between these two algorithms. LAS partitions physical processors more precisely according to the structure of shared caches, whilst CATS/CAB creates always one group per physical processor (socket). Furthermore, LAS algorithm for task stealing within a group also considers the cache hierarchy, which is beneficial when the cores in one group share more than last level of cache. Additionally, the LAS sets affinity of threads together for the whole group. This has two advantages – first, we can freely add and remove threads to the thread pool which enables support of IO operations [16], second, this strategy copes better with Hyper-Threading Technology, since it does not restrict the operating system from its own load balancing strategy [1]. Finally, we optimize the situation when the system processes multiple independent requests.

1.6 Conclusions

In this paper, we presented a novel task scheduling strategy that takes advantages on current CPU architectures and both SMP and NUMA multiprocessor systems. Our scheduler can effectively improve the data locality and thus the cache reusability when employed on parallel data stream processing systems. We have implemented a prototype of the scheduler and integrated it into the Bobox framework, which allows creation and evaluation of the execution plans. When applied on a SPARQL benchmark that process RDF data, the system achieved up to 10% speed up on double-processor SMP system and up to 3× speed up on four processor NUMA system for selected queries with respect to previous version of the scheduler.

In the future work, we would like to extend our scheduler to other domains of task processing. We would like to improve generic frameworks that also use tasks to achieve parallelism, but which process different types of datasets (not only streaming data). Furthermore, we would like to extend the scheduler to support work offloading to parallel accelerators such as GPUs and Xeon Phi cards, where the data transfers between the host system and the parallel device need to be considered.

Acknowledgements

This work was supported by the Czech Science Foundation (GACR), projects P103-13-08195S and P103-14-14292P, and by Specific Research project SVV-2014-260100.

References

1. Impact of Load Imbalance on Processors with Hyper-Threading Technology. <http://software.intel.com/en-us/articles/impact-of-load-imbalance-on-processors-with-hyper-threading-technology>, 2011. [Online; accessed 03-18-2014].
2. Intel Threading Building Blocks Reference Manual. <http://software.intel.com/en-us/node/506130>, 2014. [Online; accessed 03-18-2014].
3. B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The International Journal on Very Large Data Bases*, 13(4):333–353, 2004.
4. D. Bednárek and J. Dokulil. TriQuery: Modifying XQuery for RDF and Relational Data. In *2010 Workshops on Database and Expert Systems Applications*, pages 342–346. IEEE, 2010.
5. D. Bednárek, J. Dokulil, J. Yaghob, and F. Zavoral. The Bobox Project - A Parallel Native Repository for Semi-structured Data and the Semantic Web. *ITAT - IX. Informačné technológie - aplikácie a teória*, pages 44–59, 2009.

6. D. Bednárek, J. Dokulil, J. Yaghob, and F. Zavoral. Using methods of parallel semi-structured data processing for semantic web. *Advances in Semantic Processing, International Conference on*, pages 44–49, 2009.
7. D. Bednárek, J. Dokulil, J. Yaghob, and F. Zavoral. Data-flow awareness in parallel data processing. In *Intelligent Distributed Computing VI*, pages 149–154. Springer, 2013.
8. S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML query language. *W3C working draft*, 15, 2002.
9. F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier. Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 79–92. Springer, 2009.
10. Q. Chen, M. Guo, and Z. Huang. CATS: Cache Aware Task-stealing Based on Online Profiling in Multi-socket Multi-core Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing, ICS '12*, pages 163–172, New York, NY, USA, 2012. ACM.
11. Q. Chen, Z. Huang, M. Guo, and J. Zhou. Cab: Cache aware bi-tier task-stealing in multi-socket multi-core architecture. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 722–732. IEEE, 2011.
12. J. Cieslewicz, W. Mee, and K. Ross. Cache-conscious buffering for database operators with state. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 43–51. ACM, 2009.
13. A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, pages 100–110. Springer-Verlag, 2008.
14. Z. Falt, M. Čermak, and F. Zavoral. Highly Scalable Sort-Merge Join Algorithm for RDF Querying. In *Proceedings of the 2nd International Conference on Data Management Technologies and Applications*, 2013.
15. Q. Jiang and S. Chakravarthy. Scheduling strategies for processing continuous queries over streams. *Key Technologies for Data Management*, pages 16–30, 2004.
16. M. Kruliš, Z. Falt, D. Bednárek, and J. Yaghob. Task scheduling in hybrid CPU-GPU systems. *Informačné Technológie-Aplikácie a Teória*, page 17.
17. A. Kukanov and M. Voss. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4):309–322, 2007.
18. E. Prud’Hommeaux, A. Seaborne, et al. SPARQL query language for RDF. *W3C working draft*, 4, 2006.
19. J. Reinders. *Intel Threading building blocks*. O’Reilly, 2007.
20. A. A. Safaei and M. S. Haghjoo. Parallel processing of continuous queries over data streams. *Distrib. Parallel Databases*, 28:93–118, December 2010.
21. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP²Bench: a SPARQL performance benchmark. In *Data Engineering, 2009. ICDE’09. IEEE 25th International Conference on*, pages 222–233. IEEE, 2009.
22. O. Sinnen. *Task scheduling for parallel systems*, volume 60. John Wiley & Sons, 2007.

Chapter 10.

Conclusions and Future Work

The collection of papers included in this thesis shows a consecutive advance of the research areas of interest of the author - distributed computing, parallelism, and processing of large and semistructured data. We show that integration of these research disciplines may address common issues worth solving and their research may significantly influence each other.

The papers selected for this thesis cover a wide range of types of scientific results – formal models, algorithms and data structures, low-level system functionalities, and software tools and prototypes. We believe that such diversity of scopes significantly helps to deeply understand researched topics and their mutual relationships.

Some of the topics addressed in this thesis are further studied in our current research projects. Our experience with methods of processing of large data is being utilized in a grant project focused on efficient processing of big scientific data sets. The infrastructure for parallel processing developed mainly for studying parallel algorithms in past several years is being extended to a distributed computing system supporting a wide range of emerging modern architectures, such as Xeon Phi processors or Epiphany nodes. Our intention is to combine methods of large scale distributed processing with promising hardware technologies and apply the results to achieve more efficient data processing.

In our future research, we are planning to focus on modern parallel architectures and study the impact of their increasing ubiquity on various areas of computer science and on everyday practice in the area of computing and computer science. New methods, algorithms, programming paradigms, and other outcomes of these areas seem to be inevitable for efficient exploitation of contemporary hardware and architectural achievements.

Finally, we are quite confident that other interesting related topics will appear and that the recent results will become applicable in different or unexpected research areas, as we have witnessed in the past years.

References

- [atz10] Luigi Atzori, Antonio Iera, Giacomo Morabito: The Internet of Things: A Survey, *Comput. Netw.* Vol. 54/15, pp. 2787-2805, Elsevier, 2010
- [bed12] Davis Bednárek, Jiří Dokulil, Jakub Yaghob, Filip Zavoral: Data-Flow Awareness in Parallel Data Processing, in *Intelligent Distributed Computing VI*, Calabria, Springer, ISBN: 978-3-642-32523-6, ISSN: 1860-949X, pp. 149-154, 2012
- [box07] Don Box, Anders Hejlsberg : LINQ: .NET Language-Integrated Query, MSDN, 2007
- [dew92] DeWitt, D. J., Naughton, J. F., Schneider, D. A., and Seshadri, S.: Practical skew handling in parallel joins. In *Proceedings of the 18th International Conference on Very Large Data Bases, VLDB '92*, pp 27–40, San Francisco, USA. Morgan Kaufmann Publishers Inc., 1992
- [dok07] Jiří Dokulil, Jaroslav Tykal, Jakub Yaghob, Filip Zavoral: Semantic Web Infrastructure, in *IEEE International Conference on Semantic Computing*, Irvine, California, IEEE Computer Society, ISBN: 978-0-7695-2997-4, pp. 209-215, 2007
- [dvo07] Jana Dvořáková: Automatic Streaming Processing of XSLT Transformations Based on Tree Transducers. In *Proceedings of IDC 2007, Studies in Computational Intelligence*. Springer, 2007
- [gil02] Seth Gilbert, Nancy Lynch: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33(2):51–59, 2002
- [mar05] Wim Martens, Frank Neven: On the complexity of typechecking top-down XML transformations. *Theor. Comput. Sci.* 336/1, pp. 153-180, 2005
- [nov11] Miroslav Novotný, Filip Zavoral: Resistance against Malicious Collectives in BubbleTrust, in *PDCAT 2011 - 12th International Conference on Parallel and Distributed Computing*, Gwangju, Korea, IEEE Comp Soc, pp. 56-61, 2011
- [sch08] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel: SP2Bench: A SPARQL performance benchmark, *CoRR*, vol. abs/0806.4627, 2008
- [ste12] Maarten van Steen, Guillaume Pierre, Spyros Voulgaris: Challenges in very large distributed systems, *J Internet Serv Appl* 3:59–66, 2012
- [sto05] Mike Stonebraker et al., C-Store: A column-oriented DBMS, *Proceedings of the 31st VLDB Conference*, Trondheim, Norway, 2005