INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und
Modellierungssprachen

Oettingenstraße 67 D–80538 München

Optimizing Multiple Queries against XML Streams

Tim Furche

Diplomarbeit

Beginn der Arbeit: 01.02.2003
Abgabe der Arbeit: 31.07.2003
Betreuer: Prof. Dr. François Bry
Dipl.-Ing. Dan Olteanu

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 31.07.2003 Tim Furche

Classification

according to: ACM Computing Classification System (1998 Version)

Categories and Subject Descriptors:

H.2.4 [**Database Management**]: Systems;
H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing — *Indexing Methods*;
H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval — *Information filtering*; *Search process*;
H.3.4 [**Information Storage and Retrieval**]: Systems and Software — *Performance evaluation (efficiency and effectiveness)*;

General Terms: Algorithms, Performance

Additional Key Words and Phrases: XML query evaluation, query merging, prefix merging, path sharing, stream processing, cost function, query optimization

Abstract

Processing and querying streams, XML streams in particular, has recently become a widely recognized area of interest both in research and in industry. In contrast to traditional query evaluation for databases, where multiple queries against the same data can be evaluated sequentially, for a streamed environment only the *simultaneous* execution of multiple queries is feasible, as the sequential evaluation requires multiple passes over the stream.

This work presents an overview of techniques for optimizing multiple queries posed against a stream of XML data. Building upon the SPEX query engine [79; 105], the problem how to find a cost-optimal query plan that allows the simultaneous evaluation of multiple queries against the same stream is presented and shown to be not only hard to solve but also hard to approximate, if arbitrary parts, and not only common prefixes as in previous approaches, can be shared among query plans. Several heuristics are investigated and compared, in particular with respect to their complexity. Furthermore, it is shown how to extend the SPEX query engine to support such query plans for multiple queries. This extension proves to be both natural and efficient. An extensive experimental evaluation shows that sharing arbitrary operators under a realistic cost function results in query plans that have consistently lower cost for reasonable sets of queries than query plans where only common prefixes are considered. In most cases, the relative improvement is higher than 50%. Although the time for generating such query plans is higher than for query plans where only common prefixes are shared, the increase in time is within an acceptable margin.

Zusammenfassung

Die Anfragebearbeitung auf Strömen, insbesondere Strömen von XML Daten, ist in den letzten Jahren weitgehend als interessante Herausforderung für Forschung wie Industrie anerkannt worden. Im Gegensatz zu traditionellen Anfrageauswertern für Datenbanken, die mehrere Anfragen auf denselben Daten sequenziell verarbeiten, ist diese Verarbeitungsform für Ströme nicht geeignet, da sie mehrere Durchläufe über den Strom erfordert. Für Ströme ist daher nur die gleichzeitige Auswertung mehrerer Anfragen akzeptabel.

Diese Arbeit gibt eine Überblicksdarstellung der Techniken zur Optimierung mehrerer simultan auf einem XML-Strom zu verarbeitenden Anfragen. Ausgehend von dem SPEX-Anfrageauswerter [79; 105] wird das Problem, einen kostenoptimalen Plan zur Anfrageauswertung zu finden, vorgestellt und gezeigt, daß es für dieses Problem nicht nur schwer ist, eine optimale Lösung zu finden, sondern selbst eine nur angenäherte Lösung nicht einfach zu finden ist, wenn nicht nur gemeinsame Präfixe wie in bisherigen Ansätzen, sondern beliebige Teile eines Plans von mehreren Pläne gemeinsam genutzt werden können. Mehrere heuristische Algorithmen werden untersucht und verglichen, insbesondere im Hinblick auf ihre Komplexität. Desweiteren wird gezeigt, wie der SPEX Anfrageauswerter erweitert werden kann, um mehrere Anfragen gleichzeitig verarbeiten zu können. Eine umfassende experimentelle Auswertung der Algorithmen zeigt, daß der vorgeschlagene Ansatz unter einer realistischen Kostenfunktionen zu deutlich geringeren Kosten für die generierten Pläne zur Anfrageauswertung führt, als wenn nur gemeinsame Präfixe betrachtet werden. In den meisten getesteten Fällen ergeben sich um mehr als 50% geringere Kosten. Weiterhin nimmt die benötigte Zeit zur Generierung solcher Pläne im Vergleich zu Plänen, in denen nur gemeinsame Präfixe genutzt werden, zwar zu, der Anstieg ist jedoch verhältnismäßig gering.

Danksagung

Für die sehr gute Betreuung und eine angenehme Atmosphäre bei meiner Diplomarbeit möchte ich mich besonders bei meinen Betreuern François Bry und Dan Olteanu bedanken.

Contents

Chapter 1

Introduction

Processing of data streams or sequences of blocks of data (usually called elements of the stream) has triggered rising interest, both in research [131; 24; 144] and in industry [33; 94]. Stream processing differs from conventional methods of data processing in main memory or on data bases in the requirement to process data immediately on its arrival in the stream instead of creating an appropriate data structure that is on which the actual processing is performed afterwards. In particular, if the data to be processed changes fast, only small fragments of the data have to be processed repeatedly, the data arrives at a high rate, or the amount of data is too large to be efficiently stored and processed, stream processing provides clear advantages over traditional methods based on in-memory data structures or databases.

Indeed, in most applications for stream processing at least one of these characteristics can be observed, including some of the most exciting areas of application for streams:

— With increasing complexity of electronic and information systems, monitoring and analysis of these systems, performed on streams of data provided either by the system itself or by specific sensors information, more and more essential. Recent work in this area includes the analysis of network traffic [128; 44; 33] and monitoring of sensor networks [17; 88].

— Publish-subscribe systems provide capabilities to selectively disseminate information or publications for a large number of users or subscribers based on their profile or subscription [49]. These systems even find applications ranging from news distribution networks [117] over event notification systems alerting users of certain events [26] to the dissemination of relevant information for different needs in a military engagement [43].

— Another emerging application is the real-time in-

[131] Terry, D. B., et al. 1992. Continuous queries over append-only databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data.* ACM Press, 321–330.

[24] Carney, D., et al. 2002. Monitoring streams: A new class of data management applications. In *Proc. of the International Conference on Very Large Databases (VLDB).*

[144] Yu, P. S., Ed. 2003. *IEEE Transactions on Knowledge and Data Engineering: special section on online analysis and querying of continuous data streams.* Vol. 15. IEEE Computer Society.

[33] Cisco Systems. 2000. Cisco IOS netflow – technology data sheet.

[94] Megginson, D. and Brownell, D. 2002. SAX: The simple API for XML.

[128] Sullivan, M. and Heybey, A. 1998. Tribeca: A system for managing large databases of network traffic. In *Proc. of the USENIX Annual Technical Conference.*

[44] Duffield, N. G. and Grossglauser, M. 2001. Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Networking (TON) 9,* 3, 280–292.

[17] Bonnet, P., et al. 2001. Towards sensor database systems. In *Proc. of the International Conference on Mobile Data Management (ICMDM).* 3–14.

[88] Madden, S. and Franklin, M. J. 2002. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of the International Conference on Data Engineering (ICDE).*

[49] Franklin, M. J., Ed. 1996. *Special Issue on Data Dissemination.* Data Engineering Bulletin, vol. 19, 3. IEEE Computer Society.

[117] Ramakrishnan, S. and Dayal, V. 1998. The pointcast network. In *Proc. of the ACM SIGMOD International Conference on Management of Data.* ACM Press, 520.

[26] Carzaniga, A., et al. 2001. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS) 19,* 3, 332–383.

[43] Douglass, R., et al. 1997. Battlefield awareness and data dissemination (BADD for the warfighter. In *Proc. of the*

tegration of fast “news feeds” from diverse sources, similar to Google News (http://news.google.com) or NewsIsFree (http://www.newsisfree.com/). In the context of web services, the automatic on-the-fly syndication of streams generated by heterogeneous services is viewed as an emmerging challenge.

— With the advent of MPEG-7 [92] it is expected that an increasing number of multimedia streams accompanied by detailed meta-data information have to be efficiently filtered, transformed, and routed from sources to consumers.

— Pipeline processing allows several independent processors to be used in a pipeline, where the output stream of one processor is the output stream of the next one, therefore allowing each processor to perform its task as soon as the previous one delivers new data. Pipeline processing is of particular importance where large amounts of data have to be processed by diverse components, e.g. in astronomic data analysis [95; 65].

All these areas of applications share an inherent heterogeneity of information sources in respect to the data delivered, the parameters of the provided service, or simply the administrative responsibility. For example, monitors on devices from various vendors in network analysis might generate monitoring data with differing resolution, precision and in varying intervals, the various news agencies used as sources for syndication as well as dissemination will likely focus on different stories or aspects of the same story. These heterogeneous information sources require some agreed upon means for information encoding as provided by XML [20] dialects for the various areas such as NITF (News Industry Text Format, http://www.nitf.org) for news, MPEG-7 for metadata on multimedia streams, or SensorML [18] for sensor information and configuration. Although in some of these applications, most notably for sensor data, one commonly expects rather flat data consisting in simple attribute-value pairs—and even monitoring data sometimes requires more sophisticated modeling constructs, e.g., when representing the fused information from entire sensor networks or partial results of sensor analysis [19]—, others such as syndication of news or meta-data processing with MPEG-7 require the richer hierarchical relations provided by XML.

♦

Summing up, a clear need for the efficient processing of XML streams can be identified in all the above application areas. In particular, all the described applications require some capability to query the incoming stream: Sensor networks have to correlate and monitor incoming data based on specifications of interesting events (e.g., “send an alert, if the value of sensor A is above a certain limit for more than 5 minutes”). Publish-subscribe systems route data based on the subscriptions of their users, essentially filtering the stream of data with large numbers of queries, such as “send all publications containing a certain keyword both in the title and the first paragraph”. Syndication systems correlate information from different sources and often, again based on user profiles, select only potentially interesting data. Finally multimedia streams have to be filtered and routed based on queries over the meta-data, it is even possible to select only parts of a multimedia stream, e.g. “return only those scenes in the movie where a certain speaker uses a certain word”.

As noted above, one of the advantages of stream processing is the progressive delivery of result in a single pass over the stream. One-pass processing is required in particular on large or unbounded streams or if the results have to be delivered continuously even before all data has arrived, as it is common in monitoring and event notification systems. This manner of processing imposes three important challenges to a query engine:

(1) It is not feasible to look back in a stream (except possibly within a certain small window). Hence,

SPIE, B. R. Suresh, Ed. Vol. 3080. SPIE – The International Society for Optical Engineering, 18–24.

[92] Martínez, J. M., Ed. 2002. Mpeg-7 overview. Tech. Rep. N4980, ISO/IEC JTC1/SC29/WG11.

[95] Mehringer, D. M., et al., Eds. 1999. *Astronomical Data Analysis Software and Systems VIII: Data Pipelines.* ASP (Astronomical Society of the Pacific) Conference Series, vol. 172.

[65] Harnden, F. R., et al., Eds. 2001. *Astronomical Data Analysis Software and Systems X: Science Data Pipelines.* ASP (Astronomical Society of the Pacific) Conference Series, vol. 238.

[20] Bray, T., et al., Eds. 2000. Extensible markup language (XML) 1.0 (second edition). Recommendation, World Wide Web Consortium.

[18] Botts, M., Ed. 2002. Sensor model language (SensorML) for in-situ and remote sensors specification. discussion paper 02-026r4, Open GIS Consortium.

[19] Botts, M. and Reichardt, M. 2003. Sensor web enablement. white paper, Open GIS Consortium.

queries containing such backward navigation either have to be rewritten, as proposed in [107], or disallowed.

(2) As a stream is always processed sequentially, traditional optimization techniques, such as database indices, that are tailored to minimize the amount of access to data and the result of intermediary results, are not applicable to streams. Here, the optimization objective is to minimize the number of operations per element in the stream rather than the number of elements visited. This shift in the optimization objective is reflected in the notion of a stream index employed in the XML Toolkit [8] that does not provide efficient access paths to data relevant for a query but rather allows to skip elements not relevant for a query reducing the number of operations for each such element to one.

(3) If several queries, such as monitoring conditions or subscriber profiles, have to be evaluated against the same stream, it is not feasible to evaluate the queries sequentially as in traditional databases but they have to be processed simultaneously

In this work, the last issue is further investigated: The requirement to execute multiple queries concurrently against the same stream combined with the second observation, that the number of operations per element is crucial for fast processing of streams, there is an evident need for novel methods to optimize multiple queries for a single processing against a stream. Indeed, all proposals [4; 28; 41] of query engines for publish-subscribe systems are tailored to process large amounts of queries, but due to their application area restricted to the filtering of small self-contained documents from a stream of such documents . Among the general query engines recently proposed [56; 60; 111; 14; 105] only the [56; 60] consider the optimization of multiple queries. All these approaches have in common, that the optimization is based on common prefixes in the queries (for an in-depth comparison see Chapter 3). The approach presented here improves on these by considering not only prefixes but rather any kind of shared subparts techniques for multi-query optimization on XML streams in several points:

(1) A framework for multi-query optimization together with a formal representation of (logical) query plans on XML streams is presented in Chapter 4.

(2) Based on this representation, the general problem to derive the (cost-) *minimum common super-plan* for executing a set of queries is defined precisely and its complexity and approximability properties are investigated in Chapter 5.

(3) Several heuristic algorithms to construct the minimum common super-plan for a set of query plans are given and compared in respect to their complexity in Chapter 6.

(4) A cost model for navigational query languages against XML streams is proposed, based on experience with the SPEX processor (cf. [105]) in Chapter 8.

(5) The SPEX evaluation model is extended to allow the execution of multiple queries according to a query plan generated by the proposed algorithms in Chapter 7.

(6) An extensive experimental evaluation of the proposed algorithms based on the cost model introduced proves the feasibility of the proposed methods if the queries to be processed are known before processing (in contrast to being updated during processing) in Chapter 9.

---

[107] Olteanu, D., et al. 2002. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*. Lecture Notes on Computer Science (LNCS), vol. 2490. Springer Verlag, 109–125.

[8] Avila-Campillo, I., et al. 2002. XMLTK: An XML toolkit for scalable XML stream processing. In *Proc. of the Workshop on Programming Language Technologies for XML (PLAN-X)*.

[4] Altinel, M. and Franklin, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the International Conference on Very Large Databases (VLDB)*.

[28] Chan, C.-Y., et al. 2002a. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal (Special Issue on XML Data Management)*.

[41] Diao, Y., et al. 2002. Path sharing and predicate evaluation for high-performance XML filtering. Submitted for publication, `www.cs.berkeley.edu/~diaoyl/publications/yfilter-public.ps`.

[56] Green, T. J., et al. 2003. Processing XML streams with deterministic automata. In *Proc. of the International Conference on Database Technology (ICDT)*. 173–189.

[60] Gupta, A. K. and Suciu, D. 2003. Stream processing of XPath queries with predicates. In *Proc. of the Proc. of the ACM SIGMOD International Conference on Management of Data.*

[111] Peng, F. and Chawathe, S. S. 2003a. XPath queries on streaming data. In *Proc. of the Proc. of the ACM SIGMOD International Conference on Management of Data.*

[14] Barton, C., et al. 2003. Streaming XPath processing with forward and backward axes. In *Proc. of the International Conference on Data Engineering (ICDE).*

[105] Olteanu, D., et al. 2003. An evaluation of regular path expressions with qualifiers against XML streams. In *Proc. of the International Conference on Data Engineering (ICDE).*

♦

The remainder of this work is structured as follows: In Chapter 2 the proposed approach for multi-query optimization is compared to conventional query optimization techniques for databases and tuple streams detailing the use of query plans for XML streams. A more systematic survey of related work on multi-query optimization follows in Chapter 3, focusing on previous work on processing XML streams. Based on the query plans informally introduced in Chapter 2, Chapter 4 establishes a formal representation of query plans together with an extensive use-case describing structure and generation of query plans for SPEX. The problem how to construct a common query plan with minimal cost for multiple queries is formalized in Chapter 5. Chapter 5 also investigates the properties of the problem in respect to complexity and approximability. Several heuristic algorithms for the generation process are described and compared with respect to complexity in Chapter 6. Based on experience with the SPEX engine, Chapter 8 establishes cost functions suitable for a streamed environment, where statistics for the data are unlikely to be available. Extending the use-case on the generation of query plans for SPEX from Chapter 2, Chapter 7 presents the evaluation of the query plans created by the algorithms from the previous chapter. Using the cost functions from Chapter 8 and the just introduced evaluation of query plans with the help of SPEX, an extensive experimental evaluation of the algorithms is performed in Chapter 9. Chapter 10 details the implementation of the query optimization framework, the algorithms, the cost functions, and extensions of the SPEX engine. Finally, Chapter 11 concludes with a short outlook on future work on the topic.

Chapter 2

Challenges for Query Optimization on Semi-structured Streams

The question, in which respect goals and challenges of query optimization for semi-structured streams differ from those encountered in traditional query optimization, is to be investigated in this chapter. Preceded by a short recall of query optimization in relational databases, the impact of the characteristics of a stream, in particular a semi-structured stream, on query optimization is analyzed. This analysis motivates several important shifts to the focus of query optimization in face of semi-structured streams.

Contents

To emphasize the novelties of query optimization for semi-structured data streams, a short review of traditional query optimization techniques (for relational databases) is indicated.

2.1 Traditional Query Optimization

Query optimization is an important part of query processing. Query processing is often divided into two steps: query compilation and query execution. During *query compilation* a given query is parsed and compiled into a query plan that is afterwards executed by the execution engine against the actual data. Following, [51] query compilation can be further divided into three phases (cf. Figure 2.1):

(1) The query, specified in an appropriate query language, is parsed into a query parse tree.

(2) From the parse tree, a *logical query plan* is generated and subsequently optimized by various transformation and rewriting rules. The translation from the query tree to the logical query plan as well as the rules used for optimization depend on the *logical query algebra* [55] of the data model or the database system. The logical algebra is closely related to the data model of the query language (or languages) supported. Relational database systems usually employ the relational algebra as logical query language.

[51] Garcia-Molina, H., et al. 2001. *Database systems: the complete book*, 1st ed. Prentice Hall, Upper Saddle River, New Jersey.

[55] Graefe, G. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys 25*, 2, 73–170.

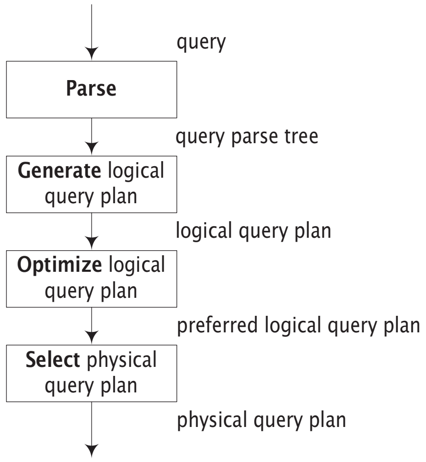


Figure 2.1: Query compilation

(3) Following construction and optimization of the logical query plan, both mostly independent of the physical aspects of the query execution engine, the logical query plan is translated into a *physical query plan* specified in the physical algebra [55] of the database system. The physical algebra, and consequently the physical query plan, differs from the logical algebra in that it provides a finer granularity of operators. Hence, a physical algebra contains for many operators of the logical algebra several physical operators with various trade-offs among the operators. For example, the relational join operator can be realized in numerous ways, such as nested-loop, merge or hash-based join algorithms. Therefore, the translation of a logical into a physical query plan often involves a selection among many alternative plans, usually based on a cost estimation.

As this work focuses on optimization of logical query plans for semi-structured streams, in the following the premises and objectives for traditional optimization on logical query plans are examined. Some consideration on the generation of a physical query plan for the SPEX evaluation engine is given in Chapter 7, presenting an application for the methods proposed in the following chapters.

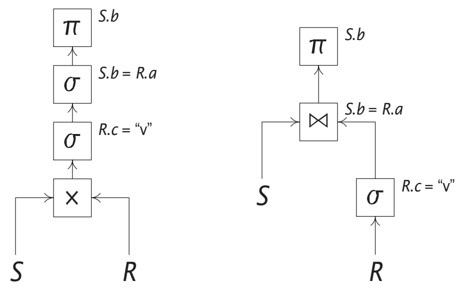
2.1.1 Optimizing Logical Query Plans

As briefly mentioned above, the primary objective for query optimization on (relational) databases is to minimize access to data on secondary storage, as that access is usually by orders of magnitude more expensive than in-memory operations, and to minimize intermediary results. To meet these objectives, the initial logical query plan similar to the parse trees of a query, is revised by applying rewriting and transformation rules to the plan that change the estimated cost of the query plan but not its semantics. The best results in this optimization phase can be obtained if the selection and the order of application of these rewriting rules is determined based on the estimated cost of the resulting query plan, e.g., based on statistics over the data accessed by the query or on properties of the operators of the relational algebra. More in-depth descriptions of query optimization and cost estimation for relational databases can be found among many others in [55; 51].

To illustrate the kind of optimizations usually considered on a logical query plan consider Figure 2.2. Figure 2.2(a) gives the initial query plan derived from the SQL query `SELECT S.b FROM S, R WHERE S.b = R.a AND R.c = "v"`, selecting the b attribute from relation S (denoted $S.b$), if there is a tuple in the cross product of S with another relation R, such that $S.b$ is equal to $R.a$ and $R.c$ has the value “v”. In relational algebra this query is denoted as $\pi_{S.b}(\sigma_{S.b=R.a}(\sigma_{R.c=\text{“}v\text{”}}(R \times S)))$. The later expression serves as basis for the initial query plan. Obviously this query plan can be improved quite notably, if one observes that a part of the selection expression contains only attributes from relation R. Pushing the selection to R and changing the cross product into a join leads to the query plan shown in Figure 2.2(b), a considerable improvement regardless of the concrete data the query will be processed against as the initial plan constructed first the entire cross product with a size $|R| \cdot |S|$ and evaluated then the selection on all constructed tuples, whereas the optimized query performs the selection directly on R, thereby also reducing the number of tuples that have to be considered for the join. If there is an index on the $R.c$ attribute, the advantage of the second query plan is even higher. In this simple case the decision for the second query plan is independent of the concrete data, but there are many other cases where optimization has to rely on statistics about the data to be queried, e.g., when considering the order of selections on a single relation.

[55] Graefe, G. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys 25*, 2, 73–170.

[51] Garcia-Molina, H., et al. 2001. *Database systems: the complete book*, 1st ed. Prentice Hall, Upper Saddle River, New Jersey.



(a) Initial query plan (b) Optimized query plan

Figure 2.2: Two query plans for the same query

Similar to this example, optimization on logical query plans for relation data often reduces to reordering of operators and in some cases to replacing operators (or sequences of operators) by ones that are expected to be evaluated more efficiently. In particular, operator reordering is possible due to the an important property that is shared by all basic operators of the relational algebra: they filter or enhance the original data, but always retain either a super- or a subset (with respect to the tuples or attributes) of the input. This property leads to a high flexibility in the positioning of operators. For example, a selection can be pushed to nearly any position in the query plan with the single restriction not to be pushed after a projection that removes the attributes the selection is based on. This flexibility in operator placement is considered one of the strengths of the relational data model, as it allows extensive optimization without effect on the correctness of the query plan.

It is worth mentioning, that these considerations on operator reordering for relational databases apply even stronger for query optimization on streams of tuples or attribute-value pairs, as many such systems emphasize on filtering of the incoming data and do not provide more expensive, for unbounded or very large streams infeasible, operators such as cross products or joins.

♦

Naturally, the operators used in querying semi-structured data differ considerably from the operators used to access relational data. In particular, operator reordering is far less promising on semi-structured data streams. To illustrate this issue, it is important to identify the differences between the logical algebra used in this paper for semi-structured streams and the relational algebra used as logical algebra in relational database systems. Therefore, the following section introduces the semi-structured data model together with a query algebra and a query language on such data.

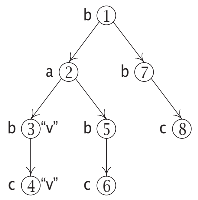


Figure 2.3: An XML data tree

2.2 Querying XML Data

In this work, XML data is considered to be tree-shaped. The tree is an ordered, node-labeled and its nodes are called *elements* of the XML data (for reasons of simplicity, we do not distinguish between different kinds of nodes such as element, text, or attribute nodes, though the presented approach can easily be extended to several node types). Each element can have a number of *local properties* or attributes. Here, we restrict the attributes to the label of an element and the text contained in an element. In Figure 2.3 a sample tree is shown with labels represented left of the element identified by its position in a pre-order traversal of the tree. The text contained in an element is pictured to the right of that element in quotation marks. It is assumed that the siblings of an element are ordered as depicted. There are several structural relations between elements that can be derived from such a document tree: The three base relations *child*, *next-sibling*, and *next* associate an element with its children, its following sibling, and the following element in a pre-order traversal respectively. For each base relation, there is a transitive closure, viz. *descendants*, *next-siblings*, and *nexts*, and an inverse relation (cf. [23]), viz. *parent*, *preceding-sibling*, and *preceding*, that in turn have a closure relation, viz. *ancestors*, *preceding-siblings*, and *precedings*.

Let Relations be the set of relations just defined, Labels the set of possible labels for an element, Texts

[23] Calvanese, D., et al. 2000. Containment of conjunctive regular path queries with inverse. In *Proc. of the International Conference on the Principles of Knowledge Representation and Reasoning (KR)*. 176–185.

the set of possible texts contained in an element. Then an **XML data tree** is formally defined as a tuple $T = (\text{Elements}, \mathcal{A}, \mathcal{L}, \mathcal{T})$ where

—Elements is the set of elements in the tree,
—$\mathcal{A}$: Elements → Relations → $\wp$(Elements) is a mapping from elements and relations to set of elements, such that $\mathcal{A}[\![r]\!](e)$ is the set of elements in T that stand in relation r with the element e,
—$\mathcal{L}$: Labels → $\wp$(Elements) is a mapping from labels to sets of elements, such that $\mathcal{L}[\![\mathsf{l}]\!]$ is the set of elements in T with label l, and
—$\mathcal{T}$: Texts → $\wp$(Elements) is a mapping from texts to sets of elements, such that $\mathcal{T}[\![\text{“}\mathtt{t}\text{”}]\!]$ is the set of elements in T containing the text “t”.

For querying such a data tree T, we use an abstraction of the navigational features of XPath [34; 15] influenced by [96], called **RPQ** (*r*egular *p*ath *q*ueries). RPQ contains nearly all features from positive core XPath [53] adding intersections. RPQ uses *variables* to identify sets of elements from the data tree and operators to restrict the set of elements identified by a variable. The bindings of the variables in an RPQ query with n variables are represented as a set of n-tuples over Elements, where each such tuple $t \in \text{Elements}^n$ represents one combination of variable bindings to elements. For a tuple t, $t.v$ represents the binding of variable v in E. All variables are bound to all elements in the tree, unless restricted by one or more of the following expressions.

The *operators* of RPQ correspond to the relations and properties defined on a data tree. For each of the relations defined above, there is a *relation operator*, as shown in Table 2.1. Relation operators form relation expressions $v\ r\ w$ that restrict the set of variable bindings such that for each binding tuple t the binding for v must stand in r relation to the binding for w. Additionally, for each label $l \in$ Labels there is a

[34] Clark, J. and DeRose, S., Eds. 1999. XML path language (XPath) version 1.0. Recommendation, World Wide Web Consortium.

[15] Berlund, A., et al., Eds. 2002. XML path language (XPath) 2.0. Working draft, World Wide Web Consortium.

[96] Meuss, H. and Schulz, K. 2001. Complete answer aggregates for tree-like databases: A novel approach to combine querying and navigation. *ACM Transactions on Information Systems (TOIS) 19,* 2, 161–215.

[53] Gottlob, G., et al. 2003. The complexity of XPath query evaluation. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS).* 179–190.

	base	inverse	closure	inverse closure
child	$\lhd$	$\rhd$	$\lhd^+$	$\rhd^+$
next-sibling	$\prec$	$\succ$	$\prec^+$	$\succ^+$
next	$\ll$	$\gg$	$\ll^+$	$\gg^+$

Table 2.1: RPQ relations

$$
\begin{aligned}
\mathit{RPQ} &::= \mathit{Identifier}(\mathit{Var})\ \texttt{:-}\ \mathit{Expr}.\\
\mathit{Expr} &::= \mathit{Expr} \wedge \mathit{Expr} \mid \mathit{Expr} \vee \mathit{Expr} \mid (\mathit{Expr}) \mid\\
&\qquad \mid \mathit{Var}\ \mathit{Relation}\ \mathit{Var} \mid \mathit{\mathbf{Label}}\ (\mathit{Var}).\\
\mathit{Relation} &::= \mathit{Base} \mid \mathit{Inverse} \mid \mathit{Base}^+ \mid \mathit{Inverse}^+.\\
\mathit{Base} &::= \lhd \mid \prec \mid \ll . \qquad \mathit{Inverse} ::= \rhd \mid \succ \mid \gg .
\end{aligned}
$$

Table 2.2: Grammar for RPQ

property operator l and for each text “t” $\in$ Texts there is an operator “t”. Property expressions use property operators to restrict the bindings for a variable to elements with a certain property (label or text), e.g., $\mathsf{l}(v)$ restricts the set of variable bindings to binding tuples t such that $t.v \in \mathcal{L}[\![\mathsf{l}]\!]$.

Relation and property expressions can be combined via $\wedge$ or $\vee$ to form conjunctions or disjunctions. An RPQ *query* is an expression of the form $Q(h)\ \texttt{:-}\ E$ where Q is an arbitrary identifier for the query, h is a variable occurring in E that identifies the set of elements that are selected by the query, and E is either a relation or property expression or built up from such expressions using conjunctions or disjunctions. h is referred to as the *head variable*, all other variables occurring in E are body variables. For the precise syntax, please refer to Table 2.2

Given an RPQ expression E, the result of evaluating E against $T = (\text{Elements}, \mathcal{A}, \mathcal{L}, \mathcal{T})$ is $S[\![E]\!](\beta)$, where $\beta = \text{Elements}^n$. The result is a subset of the Elements^n, i.e., a set of tuples, where each tuple contains one binding for each variable in E. From the result set, the bindings for the head variable v are obtained by a simple projection. Hence, for a query $Q(v)\ \texttt{:-}\ E$, $\mathcal{R}[\![Q(v)\ \texttt{:-}\ E]\!](\text{Bindings})$ specifies the set of bindings of the head variable v to elements in the XML data tree. The precise definition of the semantics is given in Table 2.3 by means of the semantic mappings R for queries and S for expressions of RPQ.

Table 2.3 shows that disjunctions, resp. conjunctions of RPQ expressions can be directly mapped to unions, resp. intersections of the result of their operands. Furthermore, the relation and property op-

$$
\begin{aligned}
\mathcal{R} &: \text{Query} \to \text{Bindings} \to \text{Nodes}\\
\mathcal{R}[\![Q(v)\ \texttt{:-}\ E]\!](\beta) &= \pi_v(\mathcal{S}[\![E]\!](\beta))\\[1ex]
\mathcal{S} &: \text{Expression} \to \text{Bindings} \to \text{Bindings}\\
\mathcal{S}[\![E_1 \wedge E_2]\!](\beta) &= \mathcal{S}[\![E_1]\!](\beta) \cap \mathcal{S}[\![E_2]\!](\beta)\\
&= \mathcal{S}[\![E_1]\!](\mathcal{S}[\![E_2]\!](\beta)) = \mathcal{S}[\![E_2]\!](\mathcal{S}[\![E_1]\!](\beta))\\
\mathcal{S}[\![E_1 \vee E_2]\!](\beta) &= \mathcal{S}[\![E_1]\!](\beta) \cup \mathcal{S}[\![E_2]\!](\beta)\\
\mathcal{S}[\![(E)]\!](\beta) &= \mathcal{S}[\![E]\!](\beta)\\
\mathcal{S}[\![v_1\ r\ v_2]\!](\beta) &= \sigma_{t.v_2 \in \mathcal{A}[\![r]\!](t.v_1)}(\beta) =\\
&= \{t \in \beta \mid t.v_2 \in \mathcal{A}[\![r]\!](t.v_1)\}\\
\mathcal{S}[\![\mathsf{l}\,(v)]\!](\beta) &= \sigma_{t.v \in \mathcal{L}[\![\mathsf{l}]\!]}(\beta) = \{t \in \beta \mid t.v \in \mathcal{L}[\![\mathsf{l}]\!]\}\\
\mathcal{S}[\![\text{“s”}\,(v)]\!](\beta) &= \sigma_{t.v \in \mathcal{T}[\![\text{“s”}]\!]}(\beta) = \{t \in \beta \mid t.v \in \mathcal{T}[\![\text{“s”}]\!]\}
\end{aligned}
$$

Table 2.3: Denotational Semantics for RPQ

erators are reduced to selections on the set (or relation) of bindings. Therefore, all properties of the relational algebra carry over to the semantics of RPQ.

In the following, inverse relations are not explicitly considered, for replacing in an RPQ each $v\ \overline{r}\ w$, where $\overline{r}$ is inverse to a base relation r, by $w\ r\ v$ yields an equivalent RPQ without inverse relation. (Note that such a rewriting might result in complex queries with intersections. [107] describes this and more sophisticated rewritings of inverse relations resulting in simpler queries.)

♦

To further illustrate the features of RPQ consider the RPQ query $Q(v4)\ \texttt{:-}\ v0 \lhd v1 \wedge \mathsf{a}(v1) \wedge v1 \lhd v2 \wedge \mathsf{b}(v2) \wedge \text{“v”}(v2) \wedge v1 \prec^{+} v3 \wedge v3 \lhd^{+} v4 \wedge \mathsf{c}(v4)$ that selects all c elements $v4$ that are descendants of elements $v3$ that are following siblings of a elements $v1$ that have at least one b child containing the text “v” and are children of an element. The meaning of the query can be more easily grasped if one pictures the relations among the elements in a graphical manner, as shown in Figure 2.4. This graphical notation of RPQ is referred to in the following as **query graph**, a legend to it is provided in Figure 2.5. On the XML tree from Figure 2.3 this query selects only the c element 8, but not the other c elements, as they are not descendants of an element that is a following sibling of an a. The corresponding bindings of the variables to elements in the data tree is pictured in Figure 2.6

It proves to be helpful, to classify RPQ queries by the allowed relations among variable bindings as these

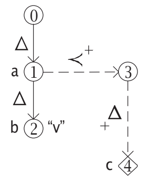


Figure 2.4: Query $Q(v4)\ \texttt{:-}\ v0 \lhd v1 \wedge \mathsf{a}(v1) \wedge v1 \lhd v2 \wedge \mathsf{b}(v2) \wedge \text{“v”}(v2) \wedge v1 \prec^{+} v3 \wedge v3 \lhd^{+} v4 \wedge \mathsf{c}(v4)$

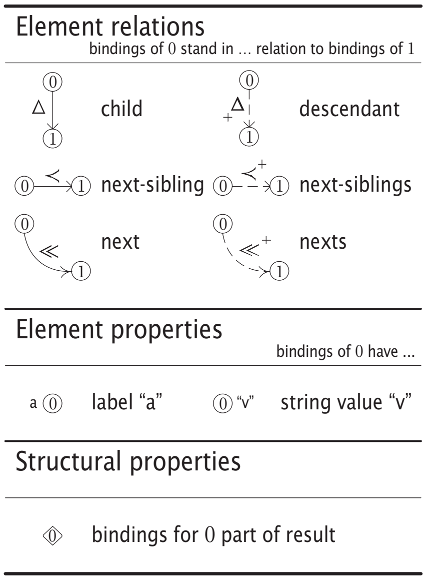


Figure 2.5: Legend to the graphical notation for RPQ

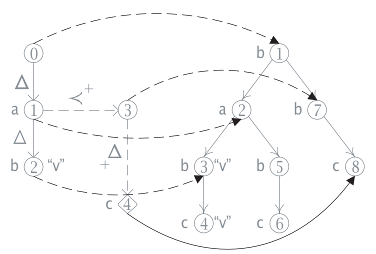
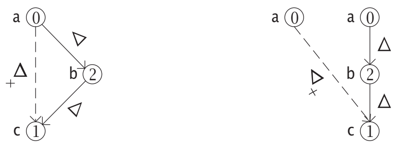


Figure 2.6: Binding for variables from Figure 2.4 to the data tree in Figure 2.3

[107] Olteanu, D., et al. 2002. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*. Lecture Notes on Computer Science (LNCS), vol. 2490. Springer Verlag, 109–125.



(a) General graph query

(b) Distinct-intersection DAG query

Figure 2.7: Graph queries

classification closely corresponds to different classes of complexity for evaluation as shown in [106]. Informally, queries are classified by the shape of their corresponding query graph. If the query graph is a single path (all nodes have at most one in- or outgoing edge respectively) the corresponding query is called *path query*, is the query graph tree-shaped (containing nodes with several outgoing edges) the corresponding query is called a *tree query*, and if the query graph is a full graph (containing also nodes with several incoming edges) the corresponding query is called a *graph query*. Furthermore, *distinct-intersection DAG queries* are queries where the corresponding query graph is acyclic and does not contain two distinct directed paths with the same source *and* sink. The latter restriction restricts the graph in such a way that for all nodes the incoming paths have to be distinct, i.e., may not contain a same node. Figure 2.7 shows a general graph query, that is not a distinct-intersection DAG query, and a very similar distinct-intersection DAG query. Both queries select c elements, if they are descendants of an a and children of an b containing the text "v" that are children of an a. But the first query stipulates additionally, that the b must be children of the *same* a that the c is descendant of. As such, in contrast to the first one the second query still matches if one adds an additional a between elements 2 and 3 in the data tree from Figure 2.3. This lack in expressiveness of distinct-intersection DAG queries is offset by the fact that they can be often more efficiently evaluated (cf. [106]).

Having established a framework for querying XML, we can know turn our attention back to the question how to optimize the just introduced queries using logical query plans.

[106] Olteanu, D., et al. 2003. Advanced techniques for streamed and progressive evaluation of XPath. Research report, University of Munich, Institute for Computer Science.

2.3 Optimizing Queries against XML Streams

2.3.1 Optimization Objective

Where the previous section establishes a framework for querying XML, this section focuses on the optimization of queries against XML data, in particular XML streams. The semantics discussed in the last section allows to query XML with (relation and property) operators that are composed from the basic relation operators. This seems to suggest, that common techniques for query optimization on relational query plans can be easily adapted to XML. Indeed, the Tukwila system [72; 73] demonstrates the feasibility of an approach based on an evaluation model similar to the RPQ semantics *on a small number of elements*: A special operator provides all combinations of elements needed for further evaluation, the remaining operators (e.g., selection or join operators) are evaluated on these combinations following a query plan. For the query from Figure 2.4 this operator returns for each a element in the data each combination of b elements and c elements that are related to the a as given by the query. Obviously, this approach is not judicious for a larger (or even unbounded) number of elements to be queried, as the number of generated tuples can grow exponential in the number of elements.

If the elements and their relations can be easily accessed in arbitrary order, it is possible to retain almost all of the freedom in choosing the order of operator application that the relational algebra provides but still to avoid to compute all combinations of bindings for all variables. In this work, we focus on the influence of *streamed* processing of XML data on (logical) query optimization. As stated above, the pivotal premise of streamed processing is to process the data progressively as it arrives without first creating intermediary data structures to hold the data and without several passes over the data. In a stream, the data tree from Figure 2.3 is serialized in pre-order leading to a stream of nested elements that are represented by

[72] Ives, Z. G., et al. 2001. Integrating network-bound XML data. *IEEE Data Engineering Bulletin 24,* 2, 20–26.

[73] Ives, Z. G., et al. 2002. An XML query engine for network-bound data. *VLDB Journal Special Issue on XML Data Management*.

begin- and end-element markers or tags:

⟨b⟩⟨a⟩⟨b⟩⟨c⟩v⟨/c⟩⟨/b⟩⟨b⟩⟨c⟩⟨/c⟩⟨/b⟩⟨/a⟩
⟨b⟩⟨c⟩⟨/c⟩⟨/b⟩⟨/b⟩.

The sequential nature of the data stream has severe implications on query evaluation and optimization. As it is not acceptable to store the stream for later processing, all operators have to operate on the current elements only and can never "look back". The fundamental observation for querying an XML stream is that *the sequence of the data implies the sequence in which the relation operators can be applied.* This is due to the fact that all the relation operators relate an element with other elements that come later in the stream (assuming we have rewritten inverse relations as described above). To illustrate this, consider again the query from Figure 2.4. There are property operators that operate solely on the bindings of $v1$ or $v2$, e.g., a $(v1)$ or "v" $(v2)$. Conventional optimization techniques might suggest to evaluate first these property operators and then to relate the (hopefully considerable smaller) result bindings with the relation operator $\lhd$ that relates $v1$ and $v2$. Furthermore, assuming it is known that there are very few c elements occurring in the data at all, first to look at the bindings of $v4$ that have the label c and then to relate them via the bindings of $v3$ to $v1$. But in a streamed context, this is not possible, as it would require to store all the elements that occur in the stream but are to be considered later due to the query plan. This would violate the essential objective for streamed processing, to store as few data as possible and to output result as soon as it is available. In a streamed environment, all property operators operating on a elements have to be evaluated before such operators on bs and cs that are related to an a as such bs and cs occur after the related a in the stream.

This fundamental realization restricts the freedom of query optimization severely: *For streamed processing the order of access to the data is dictated by the stream and can not be selected by the optimizer.* The immediate consequence of this property of streamed querying is that the optimization objective changes: Where optimizers for relational databases try to reduce access to secondary storage (i.e., optimize the access paths) and to reduce intermediary results, query optimization for XML data streams is concerned more about

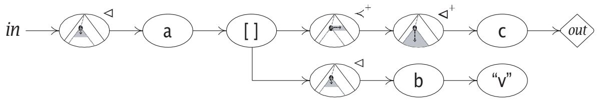


Figure 2.8: Query plan for query from Figure 2.4

(1) generating query plans that respect the order of access to data dictated by the stream, in particular any kind of access to past elements should be avoided at all cost and
(2) minimizing the number of operations performed by the evaluation engine per element. This actually might entail the minimization of the number of elements for which some evaluation beyond mere skipping is required, therefore the minimization of intermediary results.

2.3.2 Query Plans for XML

To allow a deeper discussion of these challenges for optimizing queries on XML streams, an informal introduction to query plans for XML is in place. Based on the operators just defined as part of a (logical) query algebra for navigational query languages such as RPQ or XPath, a query plan is specifying the flow of data through this operators. Figure 2.8 shows a logical query plan for the query from Figure 2.4. It specifies that all data will flow starting from *in* through the operators in the direction of the edges. Note, that there is a single data source, the stream, in contrast to relational query plans that usually contain several relations as sources of data. The operators used in the query plan are the operators from the logical query algebra, i.e., relation and property operators, with some additional *structural operators* added that are used to specify branches and joins in the graph. With these additional operators, the relations among the variables in an RPQ query can be represented in the structure of the query plan without having to retain the variables in the graph:

— Path queries are represented naturally by connecting the corresponding operators into a path, e.g., the query $Q(v2)$:- a$(v1) \wedge v1 \lhd v2 \wedge$ b$(v2)$ that selects all b children of a elements leads to the query plan in Figure 2.10. This query plan can be interpreted as follows: All elements from the stream are passed through the three operators in sequence. The data stream is enriched with bindings to elements, that indicate elements that are selected by the query so

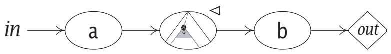


Figure 2.9: Plan for $Q(\nu 2)$:- $\mathsf{a}(\nu 1) \wedge \nu 1 \lhd \nu 2 \wedge \mathsf{b}(\nu 2)$.

far. Initially, there are bindings to all elements in the stream, as stipulated by the RPQ semantics* Each operator changes the bindings it finds in the stream, e.g., the property operator a retains only bindings to elements with label a, the ◁ operator replaces the bindings encountered in the stream by bindings to the children of those elements for which it finds a binding in the stream. There is one special operator, that is not directly derived from the query algebra, the head or *output operator* labeled with *out*. It indicates that the bindings encountered at that point are result.

— Tree queries such as the query from Figure 2.4, for which Figure 2.8 gives a query plan, require the use of one additional structural operator to express several restrictions on the same variable. The *predicate operator* labeled with [ ] indicates that the bindings encountered in the stream are restricted by several relation operators instead of only one.

— Finally, for graph queries (cf. Figure 2.7) two more operators are needed to indicate that the bindings obtained by different sub-plans have to be considered together, either by retaining only those bindings that occur in both (*intersection operator* labeled with ∩) or by accepting all bindings (*union operator* labeled with ∪). The full notation is given in Figure 2.10.

In the described way, the query plan specifies an incremental construction of the query bindings for the result variable only. Not all the tuples of bindings are constructed as the semantics of RPQ suggests, but rather the bindings for different variables are considered in sequence. The order in that they are considered is determined by the relations among them: an expression $\nu 1\, r\, \nu 2$ where r is a relation indicates that the bindings for $\nu 1$ are constructed before the bindings for $\nu 2$ and used by the r relation operator to obtain bindings for $\nu 2$. All property operators for $\nu 1$ have to be placed before any relation operator for a relation expression with $\nu 1$ as source and after relation

*This is due to the implicit binding of all variables in RPQ to all elements in the data tree unless the variables are restricted further. For query languages such as XPath, that allow queries to select, e.g., only the top-most element, appropriate operators can be easily added, but are not considered here for reasons of simplicity.

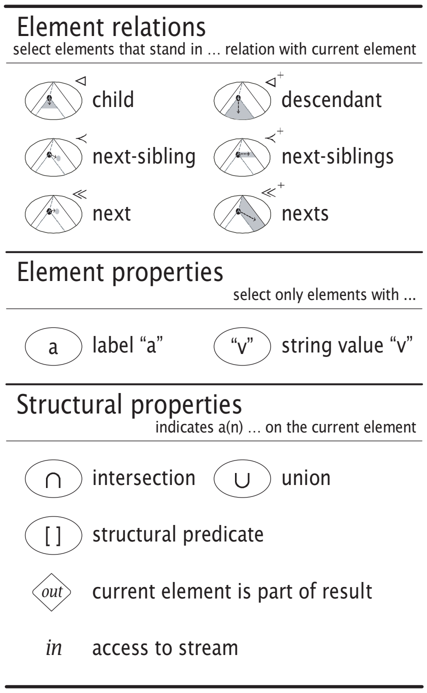


Figure 2.10: Graphical notation for RPQ query plans

operators for a relation expression with $\nu 1$ as sink. Hence, the query plan respects the order of access to elements in the data stream, as stipulated above.

2.3.3 Optimizing XML Query Plans

To give an impression of the optimization possible on these query plans, some examples for optimization techniques are apt to be considered:

— Whereas reordering of operators is a central concept when optimizing query plans for relational databases, in the query plans just introduced the position of most operators is fixed due to the sequential nature of the access to data from the stream. More precisely, the order of the relation operators can not be changed and all non-relation operators (such as a label or text operator) can not be moved before or after a relation operator. Therefore, reordering can only be applied if there are several property or structural properties in between two relation operators. For example, if there is a restriction on the label and the text of a variable, as in the query from Figure 2.4 for variable $\nu 2$, the order of the two property operators is arbitrary and can be determined by the optimizer. In the same way, the optimizer might decide the order of the predicate operator in the corresponding query plan (cf. Figure 2.8) and the label operator a that are both within the same relation operators. Instead of

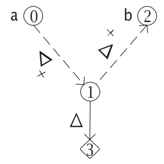


Figure 2.11: Query with inverse relation

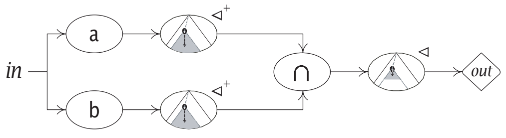


Figure 2.12: Optimized plan for query from Figure 2.11

placing the label operator as shown, it might for example be put on both branches after the prefix.

It should be obvious that this kind of reordering, although in some cases useful, is less central to the optimization of query plans than in the relation case, due to the reasons stated above.

— Another common technique for plan optimization is the replacement of operators or operator groups by other, better suited operators. One example for this optimization has been considered above directly on the query, but can be as well performed in the logical plan optimization: The replacement of operators on inverse relations. In this case, only the optimized query plan will respect the order of access as dictated by the stream. Figure 2.11 depicts the query $Q(\nu 3)$:- $\mathsf{a}(\nu 0) \land \nu 0 \lhd^{+} \nu 1 \land \nu 1 \rhd^{+} \nu 2 \land \mathsf{b}(\nu 2) \land \nu 2 \lhd \nu 3$, that selects the children of an element that is the descendant of an a and has an ancestor b. Informally, this query is equivalent to a query that selects the children of an element, if it is the descendant of both an a and an b, leading to the optimized query plan shown in Figure 2.12. For more such rewritings refer to [107].

— If one recalls, that the second objective for the plan optimization on XML streams presented in Section 2.3.1 is to minimize the number of operations performed per element, it is natural to consider whether it is possible to "reuse" certain operators that process the same (or similar) elements. This strategy can be observed on a query as shown in Figure 2.13. An initial query plan directly derived from the query

[107] Olteanu, D., et al. 2002. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*. Lecture Notes on Computer Science (LNCS), vol. 2490. Springer Verlag, 109–125.

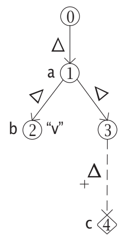
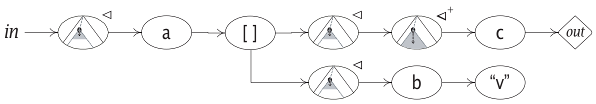
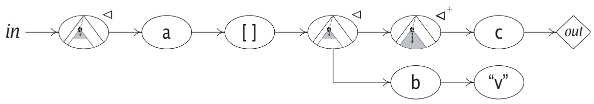


Figure 2.13: Query for prefix sharing



(a) Initial query plan



(b) Optimized query plan

Figure 2.14: Query plan for query from Figure 2.13

graph is shown in Figure 2.14(a). But that query plan is not taking into consideration that both branches following the predicate operator [ ] start with a $\lhd$ operator. In particular, since both operators also process the same input (in particular, the same bindings) and yield the same result, sharing these operator seems very natural as shown in Figure 2.14.

Indeed, operator sharing proves to be one of the most promising strategies for optimizing logical query plans on XML streams, even if the operators are not guaranteed to process the same bindings as in this case, as there is often a considerable overlapping among the bindings that are input for different operators due to the closure relations such as $\lhd^{+}$ or $\ll^{+}$ that cover large fragments of the input stream and due to the fact that some operators might perform operations on all elements in the input (e.g., bookkeeping of the current level of the data conveyed in the stream).

In the remainder of this chapter, the latter avenue shall be investigated more closely, in particular if there are several queries to be evaluated simultaneously against the same stream.

2.3.4 Query Plans for Multiple Queries

As discussed above, due to the requirement to process a stream in a single-pass, multiple queries that are to be evaluated against the same stream have to be processed simultaneously. In the context of optimization, this means that there should be a single query plan generated for all queries to be processed simultaneously. Clearly, the optimization of such a query plan poses several challenges beyond what has been discussed in the last section.

Though no core area of research, multi-query optimization for traditional (non-stream) DBMS has received some attention [48; 123; 119; 124; 120], dominated by two primary approaches: Merging of common subexpressions and merging of local query plans for single queries into a global query plan for several queries. Both techniques are based on finding appropriate sub-queries by reordering [123] that can be shared. Notwithstanding the fact that these techniques have limited appliance for our concern when considering the order of property operators, most of the more advanced techniques (e.g., subsumption of expressions in [119]) to exploit commonality between expressions are tailored to reducing access to secondary storage, whereas for our concern the reduction of the number of operations performed on a single data element is crucial. [119] presents an approach to find an almost optimal set of sub-queries, that if materialized and reused, can improve the speed of the query processing. Clearly, this technique is not applicable to streams where the queries are processed simultaneously instead of sequentially and where the operator order that is heavily employed to find the best set of sub-queries in [119] is much less flexible.

Some of the previous work on evaluating and optimizing queries for XML streams, such as [4; 28; 56], has considered the optimization of multiple queries to be evaluated simultaneously. But, where appropriate, only some form of prefix compaction on tree queries is considered, where operators are shared only if they can be shared continuously from the beginning of the query plan. Furthermore, no systematic consideration to the problem has been presented so far. In the following chapter, related work on multi-query processing and querying XML streams is investigated closely.

This work extends these previous approaches by proposing a novel method to the optimization of multiple-queries, that optimizes the query plan for multiple, simultaneously evaluated queries by considering common sub-queries at any position in the query plan. For example, if the two very similar queries from Figure 2.4 and 2.13 are to be executed simultaneously, the similarities among the queries can be used to reduce the operations per element considerably: In Figure 2.15(a) a common query plan for both queries is given where the edges are labeled with the queries (for space reasons 1 stands for the first, 2 for the second query) they are part of. For convenience, operators belonging to the first query only are colored in blue, those part of the second query only in red, and shared operators remain black. In contrast to a query plan as shown in Figure 2.15(b), where only those operators are shared that are part of a common prefix of the respective query plans, this query plan shares also the label operator c and the output operator. But whereas the sharing of operators part of the common prefix of the respective single query plans creates only negligible overhead on the evaluation (as the bindings generated by the operators are the same of both queries until the queries split up), this inner sharing requires a slight change to the evaluation engine: when elements and bindings to elements can arrive from several operators that are part of different queries, as it is the case for the c operator, the following operators have to process on all bindings encountered, similar to the

---

[48] Finkelstein, S. J. 1982. Common subexpression analysis in database applications. In *Proc. of the ACM SIGMOD International Conference on Management of Data.* 235–245.

[123] Sellis, T. K. 1988. Multiple-query optimization. *ACM Transactions on Database Systems (TODS) 13,* 1, 23–52.

[119] Rosenthal, A. and Chakravarthy, U. S. 1988. Anatomy of a modular multiple query optimizer. In *Proc. of the International Conference on Very Large Databases (VLDB).* 230–239.

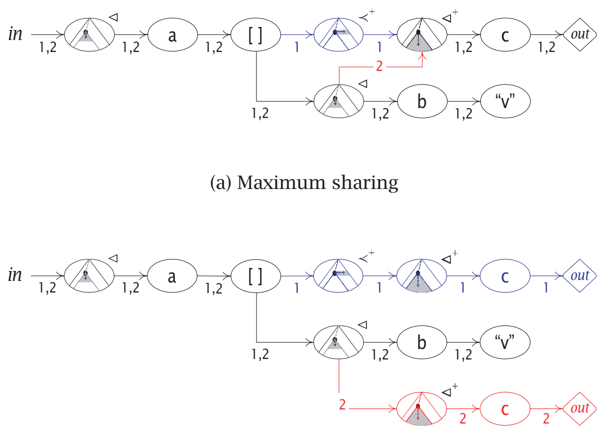
[124] Sellis, T. K. and Ghosh, S. 1990. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 2,* 2, 262–266.

[120] Roy, P., et al. 2000. Efficient and extensible algorithms for multi query optimization. *SIGMOD (ACM Special Interest Group on Management of Data) Record 29,* 2, 249–260.

[4] Altinel, M. and Franklin, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the International Conference on Very Large Databases (VLDB).*

[28] Chan, C.-Y., et al. 2002a. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal (Special Issue on XML Data Management).*

[56] Green, T. J., et al. 2003. Processing XML streams with deterministic automata. In *Proc. of the International Conference on Database Technology (ICDT).* 173–189.



(a) Maximum sharing

(b) Prefix sharing

Figure 2.15: Plans for Figure 2.4 and Figure 2.13

union case. But when the queries are split again or, as in this case, the result for the different queries are determined by the output operator, the bindings encountered at that point have to be split based on the bindings encountered when the sharing started. For example, in this query plan when the result of query 1 (cf. Figure 2.4) is determined by the output operator, only those cs for which bindings where created by the $\lhd$ operator connected to the c label operator have to be considered, but not bindings created by the $\lhd^+$ operator. For a more in-depth description of these additions to the evaluation model see Chapter 7.

This additional processing introduced by inner sharings means that it is not always preferable to share all operators possible, but rather only, if the expected gain is larger than this overhead. For this reason, the optimal query plan for several queries is not necessarily the plan with the lowest numbers of operators, but rather the one that has the best estimated cost (cf. Chapter 8).

Another important observation when considering the best query plans for some queries is that the merged plan may not be cyclic (as any query plan). Therefore, if you consider the query from Figure 2.4 and the query $Q(\nu 4)$:- $\nu 1 \lhd^+ \nu 2 \wedge$ c$(\nu 2) \wedge \nu 2 \lhd \nu 3 \wedge$ a$(\nu 3) \wedge \nu 3 \lhd \nu 4 \wedge$ b$(\nu 4) \wedge$ "v"$(\nu 4)$ for which a query plan is shown in Figure 2.16, the query plan shown in Figure 2.17(a) is not a legal query plan for the two queries. The elements from the stream are processed several times by the same operator in such a cyclic query plan which is contradictory to the initial goal, to reduce the number of operations per element. Furthermore, in such a query plan the simple description of the data flow given above is not any more applicable. Figure 2.17(b) and 2.17(c) show two different legal query plans for the two queries, the first one sharing the two $\lhd$ operators with their corresponding label operators, the second one sharing the $\lhd^+$ operator and its corresponding label operator. Although, the first query plan has a lower number of operators and thus seems to be preferable at the first glance, the second one shares the comparatively expensive $\lhd^+$ operator and might therefore actually provide a better evaluation time. Again, the selection of the query plan depends on the estimation of the cost for evaluating the query plan as discussed in Chapter 8.

After this informal introduction of queries and (logical) query plans for single and multiple queries, the next chapter emphasizes the contribution of this work by presenting a survey of related work on multi-query optimization and query evaluation against XML streams. Building upon the informal presentation in this chapter, a formal definition and description of a query plan is given in Chapter 4 that is used in Chapter 5 to define the optimization problem, how to construct the optimal query plan for several queries or query plans.

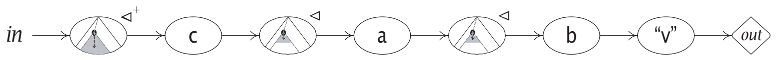
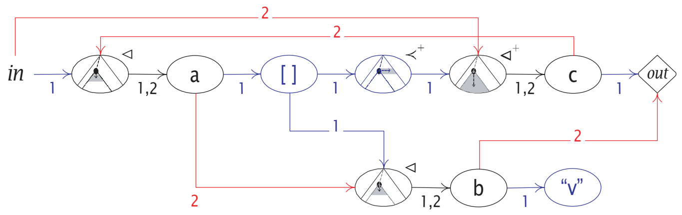
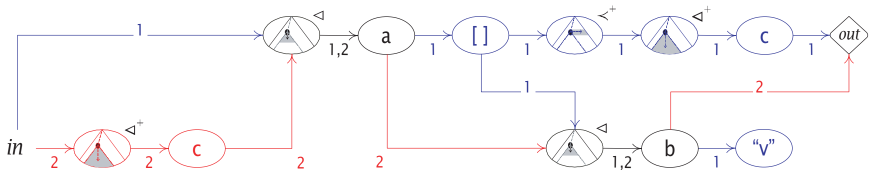


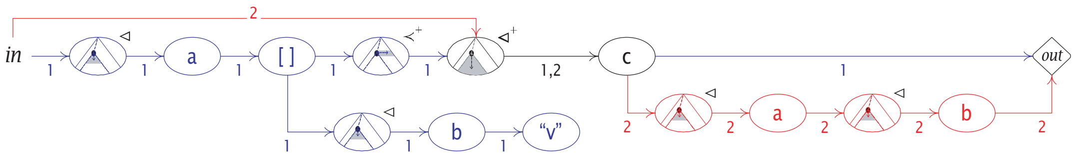
Figure 2.16: Plan for query $Q(\nu 4)$:- $\nu 1 \lhd^+ \nu 2 \wedge \mathsf{c}(\nu 2) \wedge \nu 2 \lhd \nu 3 \wedge \mathsf{a}(\nu 3) \wedge \nu 3 \lhd \nu 4 \wedge \mathsf{b}(\nu 4) \wedge$ "v" $(\nu 4)$



(a) Illegal plan with maximum sharing



(b) Legal plan



(c) Alternative legal plan

Figure 2.17: Query plans for Figure 2.4 and Figure 2.16

Chapter 3

Related work

In this chapter, a comprehensive overview over research relevant for our concern is presented. Though the focus is on comparable XML-based systems, some consideration is given to earlier work on relational databases and tuple streams. Based on the discussion of what sets query optimization for semi-structured data streams apart from traditional techniques in Chapter 2, an overview over related relational systems is presented covering scalable trigger systems, continuous query engines and publish-subscribe architectures based on tuple streams. A more in-depth discussion of XML-based publish-subscribe systems follows, where the number of queries is large compared to the size of the documents. Single query processors on XML streams, as presented in the final section, are tailored to the reverse scenario: a single query is processed over a very large (possibly unbounded) stream.

Contents

In order of their emergence, several areas of research on streaming query systems are discussed. It is worth noting, that the expressiveness of the used query language decreases from the very general ECA-Rules in Section 3.1 to the rather limited query abilities of the publish-subscribe systems in Section 3.3. The decrease in expressiveness is accompanied by an increase in scalability, allowing more and more queries to be evaluated at once.

3.1 Trigger Processing

Trigger processing for active databases has attracted a great deal of interest, cf. [38; 136]. Active databases are database systems supporting event-condition-action rules (also called triggers), i.e., rules that trigger one or more actions if a certain condition is fulfilled on an incoming event, e.g., an update. Hence, triggers, more specifically the conditions of triggers, can be considered as queries on a sequence or stream of incoming events. Most active databases

[38] Dayal, U., et al. 1995. Active database systems. In *Modern Database Systems.* 434–456.

[136] Widom, J. and Ceri, S., Eds. 1996. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann.

only allow a small number of triggers per event type, e.g., per update event on a certain table. [62] describes an approach to scalable trigger processing allowing large numbers of triggers per event type based on the TriggerMan system [45] for asynchronous trigger processing. Similar to our work, their approach is based on the observation, that among a large number of triggers many are likely to differ only in the constants used. Hence, they propose a grouping technique for triggers based on the signature of their respective predicates, where an expression signature defines an equivalence class of all instantiations of that expression with different constant values. The constants occurring in expressions of one class are stored in in-memory data structures for finding all intervals overlapping a point in the information space. To this end, the use of the IBS-tree (interval binary search tree) [63; 64] or the interval skip list [61; 46] as dynamic data structures for interval data is proposed, i.e., to support storage of interval data with efficient insertions and deletions. Furthermore, an indexing technique for the expression signatures is defined, indexing the expression signatures by their data sources. There is a substantial difference to our approach, as in the case of XML streams the predicates are grouped by the inherent order of the stream (cf. Chapter 2) and thus no further indexing of the expression signatures is possible.

[62] Hanson, E. N., et al. 1999. Scalable Trigger Processing. In *Proc. of the International Conference on Data Engineering (ICDE).* IEEE Computer Society Press, 266–275.

[45] Eric, H., et al. 1997. TriggerMan: An asynchronous trigger processor as an extension to an object-relational DBMS. Tech. Rep. 97-024, University of Florida, CISE Department.

[63] Hanson, E. N. and Chaabouni, M. 1990. The IBS-tree: A data structure for finding all intervals that overlap a point. Tech. Rep. WSU-CS-90-11, Dept. of Computer Science and Engineering, Wright State University.

[64] Hanson, E. N., et al. 1990. A predicate matching algorithm for database rule systems. In *Proc. of the ACM SIGMOD International Conference on Management of Data.* ACM Press, 271–280.

[61] Hanson, E. N. 1991. The interval skip list: A data structure for finding all intervals that overlap a point. In *Proc. of Workshop on Algorithms and Data Structures, Ottawa, Canada.* Springer Verlag, 153–164.

[46] Eric N. Hanson, T. J. 1996. Selection predicate indexing for active databases using interval skip lists. *Information Systems 21,* 3, 269–298.

3.2 Continuous Query Systems

Driven by the increasing demand for event-driven information delivery, in the early 1990s continuous query systems on relational data streams have been proposed. In contrast to one-time queries, i.e., queries that are evaluated once over a point-in-time snapshot of the data set, a *continuous query* is continuously processed over a stream of data. Continuous queries are often used for monitoring and flow analysis, as they offer excellent support for alerts and notifications. Typical queries are e.g., “select all stocks whose price increased by 10% over the last 10 minutes” in a financial monitoring system or “report all packets with a certain destination address” for network analysis. Exemplary applications are the financial search and monitoring engine Traderbot (`http://www.traderbot.com`) or the stock news monitoring system “Fly on the Wall” (`http://theflyonthewall.com`).

There is an obvious similarity between event-condition-action rules and continuous query. The most important difference is, that trigger processors usually reside upon traditional DBMS and are tightly integrated with them. Apart of that, continuous query systems can be seen as a specialization of trigger systems, providing a more efficient and scalable query evaluation at the cost of expressiveness.

3.2.1 Continuous Query Systems on Tuple Streams

Early work on continuous query systems has concentrated on data streams consisting exclusively of insertions (append-only) [131]. With the increasing success of the Internet for information delivery, the need for monitoring diverse data streams in a distributed environment has become more pressing. [86] describes the OpenCQ query system, a three-tier architecture for querying streams from diverse sources, and proposes the use of common multi-query optimization enhanced by incremental query evaluation as detailed

[131] Terry, D. B., et al. 1992. Continuous queries over append-only databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data.* ACM Press, 321–330.

[86] Liu, L., et al. 1999. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 11,* 4, 610–628.

in [85]. Catered to network analysis, the Tribecca system [128] employs a rather constricted query language, allowing only operations on the current data item of the stream or on a fixed-size window. Optimization for multiple queries are not considered.

Recently, research on continuous queries over streams of tuples has concentrated on two areas. In the context of the Telegraph project at UC Berkeley, several techniques for *adaptive continuous query systems* have been proposed [89; 30]. In this context, an adaptive continuous query system treating queries and data as duals, i.e., where both data and queries are streaming and multi-query processing is viewed as a join of query and data streams. Therefore, these systems allow new queries as well as new data to arrive at any time. They combine conventional techniques for multi-query optimization (most notably the predicate filter as detailed in [89] that is similar to the approach in the TriggerMan system) with a novel adaptive query planer [9]. Furthermore in [30], the approach of treating data and queries in a symmetric manner is extended to the notion of allowing new queries to be evaluated even on data that has arrived prior to the query.

Another focus in research is the definition and implementation of a *general architecture for relational stream management systems*. There are two major projects currently underway to specify such an architecture: The STREAM project at Stanford and the Aurora project at Brandeis University, Brown University and M.I.T. Both projects share the desire to present a comprehensive framework for processing of relational data on streams.

In [10] a survey over existing approaches and an outlook on potential research issues is presented. Based upon this work, [11] presents a general and flexible architecture for continuous queries clearly identifying the various components of a continuous query system. Features of a traditional DBMS are also provided. Their architecture is flexible in the sense, that it can support any combination of append-only and update-able input and answer stream, whereas our work assumes both streams to be append-only, in particular the answer stream is monotonic, as any element of the input or answer stream that is update-able has to be buffered until it can no longer be updated, e.g., until the end of the stream.

In [101], the STREAM system is further extended by adding resource management and approximation, and a formal semantics for continuous queries (with user-specified sliding windows) over relational data is proposed. Furthermore, the effects of their resource management strategies on established multi-query optimization techniques (considering common subexpressions and subexpression containment) are addressed together with a short discussion of issues on sharing not only common subexpressions but also synopses for approximation.

Due to the focus on monitoring applications the Aurora project [24] differs considerably from the STREAM system, as it is tailored to real-time operation and does not provide traditional DBMS features. By elaborate monitoring of the Quality of Service (QoS) dynamic resource allocation and graceful degradation strategies, such as load shedding during periods of high load, are suggested. Apart of the dynamic resource allocation, the Aurora system can be seen as a generalization of prior work on network monitoring [128] and scalable trigger systems [62].

[85] Liu, L., et al. 1996. Differential evaluation of continual queries. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*. 458–465.

[128] Sullivan, M. and Heybey, A. 1998. Tribeca: A system for managing large databases of network traffic. In *Proc. of the USENIX Annual Technical Conference*.

[89] Madden, S., et al. 2002. Continuously adaptive continuous queries over streams. In *Proc. of the ACM SIGMOD International Conference on Management of Data*.

[30] Chandrasekaran, S. and Franklin, M. J. 2002. Streaming queries over streaming data. In *Proc. of the International Conference on Very Large Databases (VLDB)*.

[9] Avnur, R. and Hellerstein, J. M. 2000. Eddies: Continuously adaptive query processing. In *Proc. of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 261–272.

[10] Babcock, B., et al. 2002. Models and issues in data stream systems. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*.

[11] Babu, S. and Widom, J. 2001. Continuous queries over data streams. *SIGMOD (ACM Special Interest Group on Management of Data) Record*, 109–120.

[101] Motwani, R., et al. 2003. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the Conference on Innovative Data Systems Research (CIDR)*.

[24] Carney, D., et al. 2002. Monitoring streams: A new class of data management applications. In *Proc. of the International Conference on Very Large Databases (VLDB)*.

3.2.2 Continuous Query Systems on Semi-structured Streams

Aside of continuous query systems for relational data streams, continuous queries against semi-structured streams have received some interest from researchers, though publish-subscribe architectures, as outlined in the following section, support an even larger number of queries, hence are better suited for the chief application area of XML, the Internet.

In [32; 31] a scalable continuous query system for XML data streams, called NiagaraCQ, is described. The core approach in NiagaraCQ to achieve scalability is based on the idea to group continuous queries on predicates using their signature similar to the approach taken in [62]. The chief contribution is to apply this technique to XML data streams, where instead of attributes arbitrary path expression can be the data source of a predicate. Furthermore, they extend this approach to include join predicates, i.e., predicates on two data sources. Nevertheless, scalability is severely limited in this approach: First, any data source used in a predicate is buffered entirely during the file scan thus providing random access to the data for the remaining operators. As argued in Chapter 2, this is not appropriate for large documents. Furthermore, though experiments in [32] show the expected increase in performance by employing grouping techniques, the scalability is still limited to thousands of queries by the high expressiveness of the employed query language (XML-QL).

Albeit focused on the integration of network-bound data, the Tukwila system developed at the University of Washington [72; 73] has a similar evaluation model. Well-established techniques for query evaluation in relational databases are extended by the *x-scan* operator, that provides pattern matching of incoming XML data against simple tree expressions and generates in essence tuples of bindings between variables and input trees. Most selection or filter operators are applied to the tuples provided by the *x-scan* operator. Again as discussed in Chapter 2, this is not feasible in our context. In particular, the tuples generated by the *x-scan* operator can be prohibitively large. For this case, [73] suggests to swap the tuples to secondary storage. As demonstrated by the various proposals for publish-subscribe systems and our work the generation of these tuples can be avoided in most cases by pushing the selection operators inside the document scan. The adaption of a virtual memory manager for XML tree fragments similar to the Tukwila *XML Tree Manager* would allow queries over windows larger than the available main memory and might be considered for future work. Furthermore, the matching operator *x-scan* treats XPath expressions as regular path expressions, creating an NFA for each query. The NFA is then translated into a DFA using the standard construction. The disadvantage of this approach, as discussed in the next section, is that the DFA can grow exponentially in the size of the queries. Recent work extends the *x-scan* operator to multiple queries, allowing several queries to be processed in a single pass, but does not provide considerable optimization on the queries.

Based on the techniques employed in the publish-subscribe system XFilter, as presented in more detail in the following section, a continuous query system for mobile clients (CQMC) has been proposed in [108].

3.3 Publish-Subscribe Architectures

With the advent of high-bandwidth communication at low cost, the number of sources generating quickly large amounts of data increases. The increasing number of sources (or publishers) combined with more and more users or subscribers, make an efficient decision which data shall be transfered (on what way) to a subscriber imperative. This decision is at the heart of *publish-subscribe* systems, that provide an efficient way to deliver the desired (parts of) publications to

[32] Chen, J., et al. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In Proc. of the ACM SIGMOD International Conference on Management of Data. *SIGMOD Record 29,* 2, 379–390.

[31] Chen, J., et al. 2002. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proc. of the International Conference on Data Engineering (ICDE).*

[62] Hanson, E. N., et al. 1999. Scalable Trigger Processing. In *Proc. of the International Conference on Data Engineering (ICDE).* IEEE Computer Society Press, 266–275.

[72] Ives, Z. G., et al. 2001. Integrating network-bound XML data. *IEEE Data Engineering Bulletin 24,* 2, 20–26.

[73] Ives, Z. G., et al. 2002. An XML query engine for network-bound data. *VLDB Journal Special Issue on XML Data Management*.

[108] Ozen, B., et al. 2001. Highly personalized information delivery to mobile clients. In *Proc. of ACM International Workshop on Data Engineering for Wireless and Mobile Access.*

subscribers. If this decision is based on the content of the publications (rather than on fixed meta-data, such as an address), a query processor is needed to filter and route the data. Obviously the scalability requirements for such an architecture are even higher than in a continuous query system (e.g., hundreds of thousands to millions of subscribers for a high rate of publications). Hence, there is an inherent trade-off in the design of a publish-subscribe system: The more expressive the subscription language, i.e., the (query) language in which the subscriptions are formulated, the less scalable the query processor. If on the other hand the expressiveness of the subscription language is too low, the selectivity of the subscriptions might suffer, increasing the amount of (potentially useless) data and diminishing the usefulness of the subscription system. Based on this observation, the expressiveness or flexibility of the subscription language used for filtering of the publications can serve as a characteristic trait of a publish-subscribe system:

— In the most basic case, a limited number of predefined channels or groups is provided by the system and a subscriber can only select among these. A famous example for such a *channel-based* publish-subscribe system is the USENET News based on the Network News Transfer Protocol [76] with millions of users and gigabytes of daily traffic. The USENET News system can also serve as an example for the advantages and pitfalls of a channel-based publish-subscribe system: Though it is able to support enormous amounts of subscribers and publications (more than 700 million publications since 1981), most users receive large numbers of news messages they are not interested in, as the selectivity of a group based approach is limited (even for the USENET News with thousands of newsgroups). The communication overhead induced by the transmission of messages unrelated to the users information interest is considerable.

— Therefore, *content-based* publish-subscribe systems allow subscriptions to be specified as predicates over an information space, i.e., a finite number of predefined attributes. In this case, publications are usually represented as attribute-value pairs and subscriptions are conjunctions of predicates. The obvious advantage is increased flexibility (and hence selectivity) for the subscriptions, as the subscriber can combine arbitrary attributes, reducing the number of unwanted publications. The disadvantage is, that a query processor is needed to match publications to subscriptions. Most recent publish-subscribe systems fall into this class.

Clear evidence of the demand for content-based systems is provided by the popularity of the PointCast news distribution network (`http://www.pointcast.com`), documented in [117], or the call for applications and techniques to effectively filter sensor, monitor and tactical data on a battlefield in the context of the DARPA "Battlefield Awareness and Data Dissemination (BADD)" [43] project.

— Though sometimes considered as a special case of content-based publish-subscribe systems, the introduction of structural matching of semi-structured data (usually in XML) poses novel challenges. Most importantly, the number of data sources, i.e., the number of different XML elements referenced in the query, is in general no longer finite due to the hierarchical and possibly recursive structure of XML. A system capable of filtering XML data using an appropriate query language (such as regular path expressions or XPath [34]) is here referred to as *XML-based* publish-subscribe system.

— The most general class is based on methods known from information retrieval considering a flat data model for the message: A subscription specifies some keywords that describe the subscribers intent. Instead of indexing the publications as in traditional information retrieval systems, the subscriptions are indexed. The main advantage of a *retrieval-based* publish-subscribe system is that it allows the user to specify a rather vague intent thus further decoupling the publisher and the subscriber. Those publications matching the intent of the subscriber to some predefined extend are delivered to the subscriber. For large-scale systems it is usually not feasible to use retrieval-based methods for online selection of publi-

[76] Kantor, B. and Lapsley, P., Eds. 1986. Network news transfer protocol – a proposed standard for the stream-based transmission of news. RFC 977, IETF.

[117] Ramakrishnan, S. and Dayal, V. 1998. The pointcast network. In *Proc. of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 520.

[43] Douglass, R., et al. 1997. Battlefield awareness and data dissemination (BADD for the warfighter. In *Proc. of the SPIE*, B. R. Suresh, Ed. Vol. 3080. SPIE – The International Society for Optical Engineering, 18–24.

[34] Clark, J. and DeRose, S., Eds. 1999. XML path language (XPath) version 1.0. Recommendation, World Wide Web Consortium.

cations, but rather for asynchronous delivery, where it is acceptable that the delivery of a publication may be considerably later than the publication time. The SIFT Information Dissemination System [140] utilizes traditional Boolean and Vector Space Model querying, but indexes the subscriptions instead of the documents. As the documents stream in, those publications with a similarity to a subscription above a specified threshold are delivered to the respective subscriber. Load distribution is considered, but not as detailed as in the distributed content-based systems presented in Section 3.3.1.

♦

The decision what kind of subscription language to support, is only one of the issues in a publish-subscribe system. In [12] two key issues with particular impact on the efficiency and scalability of a publish-subscribe system are identified:

(1) The problem of efficiently matching a publication against a large number of subscriptions. Obviously, the selection of an appropriate subscription language, as discussed above, is part of an answer to this problem.

(2) The problem of when and where to perform this matching. Essentially, there are three approaches, as shown in Figure 3.1, where to perform the matching.

— The straightforward solution is to filter at the end-points of the communication: Either as in Figure 3.1(a), all publications (often referred to as messages or events) are sent to each subscriber that can decide which messages are relevant leading to large amounts of unnecessary transmissions, if many subscribers are interested only in few messages. On the other hand, the filtering can take place at the publisher, cf. Figure 3.1(b). If most messages are of no interest for any user, this approach can lead to acceptable behavior.

— The most common approach is to allow a single centralized mediator, as shown in Figure 3.1(c). All messages are transmitted to the mediator which routes them to the subscribers according to the previously specified subscriptions. Obviously, the scalability of a centralized solution is limited by the (communication and processing) capabilities of the central system.

— A distributed mediation promises the highest scalability of all approaches at the cost of an increasingly complex routing. A network of mediators or brokers (cf. Figure 3.1(d)) is responsible for efficient multicasting of messages from publishers to subscribers. Novel techniques, such as query merging [36], enable such efficient multicasting. Distributed mediation can be further distinguished by the topology of the mediator network, in particular some approaches assume a hierarchical division of the network into subnets.

Table 3.1 gives a classification of various publish-subscribe systems according to the point of filtering and the expressiveness of their subscription language. Note, that research on distributed systems has almost exclusively focused on channel- and content-based publish-subscribe systems. Where these results can be applied to XML-based systems, is an open issue.

Based on the characteristics established in this section, the following two sections present a concise overview over the most relevant proposals for content- and XML-based publish-subscribe systems, as these share some characteristics with our work.

3.3.1 Content-based

Where early systems, such as the Elvin notification service [121; 122], have been based on end-point filtering or centralized filtering [118], it is widely accepted that a scalable publish-subscribe system requires apart of an efficient matching algorithm a distributed mediation service, as pictured in Figure 3.1(d). Before some of the proposed matching algorithms shall be discussed in greater detail, a short overview over the various approaches for a distributed mediation service is presented.

Most of the recent research on content-based publish-subscribe systems is focused on efficient architectures and algorithms for a distributed system

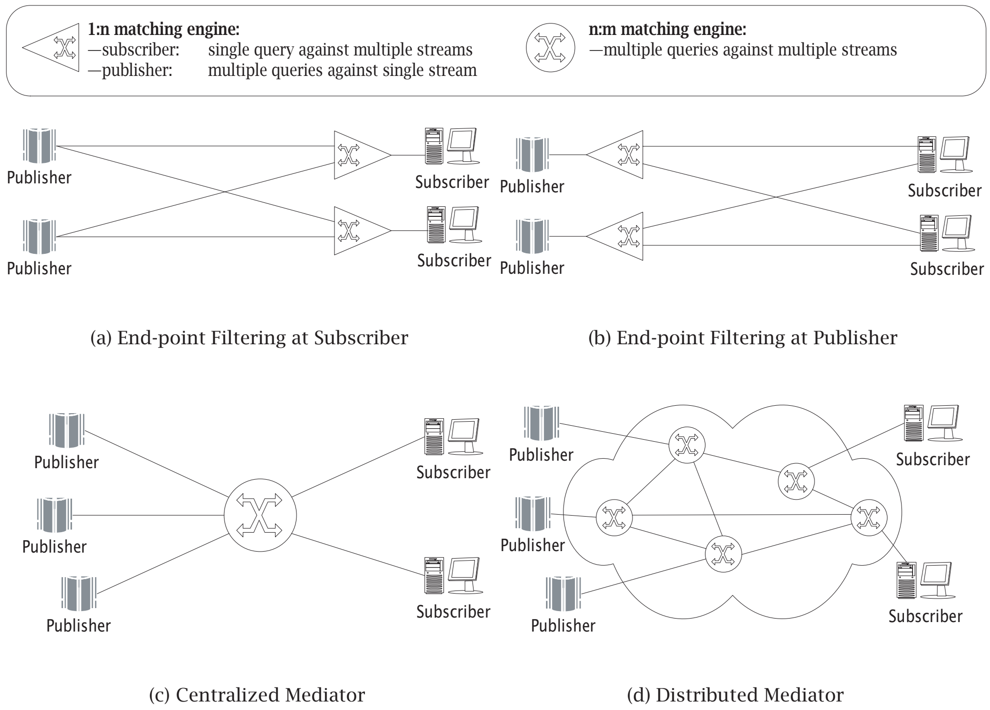
[140] Yan, T. W. and Garcia-Molina, H. 1999. The sift information dissemination system. *ACM Transactions on Database Systems (TODS) 24,* 4, 529–565.

[12] Banavar, G., et al. 1999. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS).* 262–272.

[36] Crespo, A., et al. 2003. Query merging: Improving query subscription processing in a multicast environment. *IEEE Transactions on Knowledge and Data Engineering (TKDE).*

[122] Segall, B., et al. 2000. Content based routing with elvin4. In *Proc. of AUUG2K (Australian Unix and Open Systems User Group).*

[118] Reiss, S. P. 1990. Connecting tools using message passing in the field environment. *IEEE Software 7,* 4, 57–66.



(a) End-point Filtering at Subscriber

(b) End-point Filtering at Publisher

(c) Centralized Mediator

(d) Distributed Mediator

Figure 3.1: Topologies of Publish-subscribe Systems

	Topology			
	End-point	Mediator		
		Centralized	Hierarchical client/server	Peer-to-peer
Channel		Field [118] CORBA Event Service [103] JINI Distributed Event Specification [129] Java Message Service [130]	CORBA Event Service [103] NNTP [76]	IP multicast [39] SoftWired's iBus
Content	Elvin [121]	Yeast [81] CORBA Notification Service [104] Le Subscribe [114; 115; 47]	READY [57; 58] Yu et al. [143] Siena [25; 26]	READY [57; 58] Gryphon [12; 2] Siena [25; 26] Rebeca [97]
XML		Xyleme [102] XFilter [4], YFilter [42; 41] XTrie [28] MatchMaker [83] WebFilter [113]		Snoeren et al. [3]
Retrieval		SIFT [140]		

Table 3.1: Classification of publish-subscribe systems (cf. [26]).

(cf. Table 3.1), where the publications are routed from the publishers to the subscribers over several mediators according to sophisticated routing algorithms. In this field, publish-subscribe systems are sometimes also referred to as *event notification systems*, where publications are messages and subscriptions are notification request. With the advent of the CORBA Event Service and the extended Notification Service [103; 104], providing notification channels with event filtering, durable connections and delivery-guarantee semantics (raising specific issues for designing such a notification service, cf. [52]), the JINI [129] Platform for dynamic distributed systems and the Java Message Service [130] publish-subscribe systems are expected to be deployed in increasing numbers as enterprise messaging applications (e.g., iBus Message Server, `http://www.softwired-inc.com`). Such messaging middle-ware (referred to as Message Oriented Middleware or MOM) provides a reliable architecture for messaging between distributed, decoupled components where publishers and subscribers require no knowledge about one another. Though the CORBA Notification Service allows some content-based filtering, most current enterprise messaging applications are channel-based publish-subscribe systems.

Building on the centralized event notification system Yeast [81], READY [57; 58] provides a general architecture for distributed mediation, where matching starts at subscriber, but parts of the subscription are placed upstream towards the suppliers on the mediators, if a part has a high selectivity. This hybrid approach is necessary in the READY system to reduce the load on the mediators. In the Gryphon [2; 12] and the *S*iena system [25; 26] more elaborate routing algorithms for multicast transmission of subscriptions are employed. Where the Gryphon system focuses on an efficient matching algorithm (discussed below) using global knowledge of all subscriptions, the *S*iena system uses a clever promotion strategy for subscriptions, only promoting the most general subscriptions from subscriber towards the publishers. [26] is an elaborate discussion of the design issues for a large-scale publish-subscribe system. The Rebeca [97] system developed at the University of Darmstadt combines improved multicasting techniques based on the *S*iena system with a technique for incorporating more general constraints than in previous approaches. Finally, Crespo et al. present in [36] a formalization of the query merging problem, allowing several subscriptions at intermediary systems (mediators) to be merged into a single one that can be propagated to the neighboring systems and used for routing, thus limiting the publications send to a subscriber instead of flooding every subscriber with all publications.

♦

Regardless of the topology used, all publish-subscribe systems require an efficient algorithm for matching subscriptions to incoming publications (messages). In [27] two broad categories for matching algorithms in content-based publish-subscribe sys-

[103] Object Management Group, Inc. 2001. *Event Service Specification*, 1.1 ed. Object Management Group, Inc.

[104] Object Management Group, Inc. 2002. *Notification Service Specification*, 1.0.1 ed. Object Management Group, Inc.

[52] Gore, P., et al. 2001. Designing and optimizing a scalable CORBA notification service. *ACM SIGPLAN Notices 36,* 8, 196–204.

[129] Sun Microsystems, Inc. 2001. *Jini^{TM} Technology Core Platform Specification*, 1.2 ed. Sun Microsystems, Inc.

[130] Sun Microsystems, Inc. 2002. *Java Message Service API Specification*, 1.1 ed. Sun Microsystems, Inc.

[81] Krishnamurthy, B. and Rosenblum, D. S. 1995. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering (TSE) 21,* 10, 845–857.

[57] Gruber, R. E., et al. 1999. The architecture of the READY event notification service. In *Proc. of the ICDCS Workshop on Electronic Commerce and Web-Based Applications.*

[58] Gruber, R. E., et al. 2000. READY: A high performance event notification service. In *Proc. of the International Conference on Data Engineering (ICDE).* 668–669.

[2] Aguilera, M. K., et al. 1999. Matching events in a content-based subscription system. In *Proc. of the ACM Symposium on Principles of Distributed Computing.* ACM Press, 53–61.

[12] Banavar, G., et al. 1999. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS).* 262–272.

[25] Carzaniga, A., et al. 2000. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. of the ACM Symposium on Principles of Distributed Computing.* ACM Press, 219–227.

[26] Carzaniga, A., et al. 2001. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS) 19,* 3, 332–383.

[97] Mühl, G., et al. 2002. Filter similarities in content-based publish/subscribe systems. In *Proc. of the International Conference on Architecture of Computing Systems (ARCS).* Lecture Notes in Computer Science, vol. 2299. Springer Verlag, 224–238.

[36] Crespo, A., et al. 2003. Query merging: Improving query subscription processing in a multicast environment. *IEEE Transactions on Knowledge and Data Engineering (TKDE).*

[27] Carzaniga, A. and Wolf, A. L. 2001. Fast forwarding for content-based networking. Tech. Rep. CU-CS-922-01, Department of Computer Science, University of Colorado.

tems are identified:

— The first approach is to start from the attribute constraints derived from the full set of subscriptions and move through them consulting the attributes appearing in the message. Gough and Smith [54] propose the adaption of automata-based string matching methods by imposing some order on the attributes and encoding an event as a string of attribute values in that order. An NFA is constructed from the subscriptions, accepting all string of attribute values matched by a subscription.
A similar approach is detailed in [2], used as the matching algorithm for the Gryphon system. Instead of constructing an NFA and then translating to a DFA as in the previous approach, a so-called matching tree is created directly from the submissions. A careful study of the average or expected matching time shows that this approach has an average complexity sublinear in the number of subscriptions N.
However, any approach starting with the attribute constraints from the subscriptions is in the worst-case linear in the number of subscriptions, as in the worst-case all subscriptions differ only at the last possible attribute in the matching tree or automaton. This disadvantage is obvious, if one considers sparse messages, i.e., messages using only a few attributes. As long as there are possible matches among the submissions, the presented approaches have to test every single attribute occurring in one of the submissions, instead of skipping all tests on attributes not occurring in the message.

— The opposite approach is to start from the attributes of the message and move through them consulting the constraints. This is the approach used in SIFT [140], if a new document is considered to be a "message" whose "attributes" are formed from the set of words appearing in the document. It is also the approach used by Le Subscribe [114; 115; 47]. Le Subscribe goes beyond the SIFT indexing scheme by providing a main-memory matching algorithm that is "processor cache conscious" and by providing heuristic optimizations based on a clustering of subscriptions that share the same constraints over the same attributes, similar to the predicate index of [62], thus creating a highly scalable system capable of filtering for millions of subscriptions. The matching algorithm first determines which predicates are matched by the events and then matches the predicates to subscriptions. To improve the matching of predicates to subscriptions from a naive algorithm linear in the number of subscriptions simply counting for each subscription the satisfied predicates, the subscriptions are clustered by their size and a characteristic predicate, that is the most selective predicate of every subscription in the cluster. Only subscriptions in clusters whose characteristic predicates have been matched by the message are considered further. Moreover, in [47] a dynamic clustering algorithm is proposed, adapting to changes in the subscriptions, similar to adaptivity in continuous query systems, as discussed in Section 3.2.1. Another variant of this approach is presented in [27] where a more powerful subscription language supporting disjunction is employed. Furthermore, a selectivity table is employed to efficiently determine the predicates using a certain attribute, thus allowing to filter out all predicates for attributes not occurring in the message.
These approaches share the common characteristic that their complexity is roughly bound to the number of attributes appearing in the message and not to the number of subscriptions. Hence, for publish-subscribe systems where large number or subscriptions have to be matched at the same time, the second approach is clearly more appropriate. Note, that the second approach is not immediately applicable to XML data, as an XML element (a data source of an XML

[54] Gough, J. and Smith, G. 1995. Efficient recognition of events in a distributed system. In *Proc. of the Australasian Computer Science Conference.*

[140] Yan, T. W. and Garcia-Molina, H. 1999. The sift information dissemination system. *ACM Transactions on Database Systems (TODS) 24,* 4, 529–565.

[114] Pereira, J., et al. 2000. Publish/subscribe on the web at extreme speed. In *Proc. of the International Conference on Very Large Databases (VLDB).* 627–630.

[115] Pereira, J., et al. 2000. Efficient matching for web-based publish/subscribe systems. In *Proc. of the International Conference on Cooperative Information Systems.* Lecture Notes in Computer Science, vol. 1901. Springer Verlag, 162–173.

[47] Fabret, F., et al. 2001. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of the ACM SIGMOD International Conference on Management of Data.* ACM Press, 115–126.

[62] Hanson, E. N., et al. 1999. Scalable Trigger Processing. In *Proc. of the International Conference on Data Engineering (ICDE).* IEEE Computer Society Press, 266–275.

[27] Carzaniga, A. and Wolf, A. L. 2001. Fast forwarding for content-based networking. Tech. Rep. CU-CS-922-01, Department of Computer Science, University of Colorado.

message) can be selected by an infinite number of different paths.

3.3.2 XML-based

Often based on ideas from content-based publish-subscribe systems, XML-based publish-subscribe systems nevertheless pose some novel challenges, most notably the unbounded number of data sources (corresponding to attributes in the content-based case and to nodes of the XML tree for XML-based systems) and the fact that the data sources of a message can not be considered at the same time, thus allowing random access, but are rather streamed itself (as a message can be unbounded in length and depth of the XML tree).

As XML-based publish-subscribe systems differ mainly in the subscription language and the message model (arbitrary XML data instead of simple attribute-value pairs) from content-based approaches, finding an efficient and scalable matching algorithm for XML messages has been the focus in research. All XML-based publish-subscribe systems are based on two assumptions clearly separating them from our work. First, consistent with content-based publish-subscribe systems, it is assumed that a user is only interested whether a document matches or not, but not which parts of a document matches. Thus, these systems use an XPath query similar to a predicate on the document element and are not required to track which elements actually match the query. Furthermore, the size of the publications (documents) is assumed to be rather small, thus enabling certain optimizations on predicate handling. Both assumptions are not valid for a general XPath query processor.

♦

In [102] an XML-based publish-subscribe system, called Xyleme, tailored to monitoring (HTML and) XML documents in the web is described. This approach is separated from traditional publish-subscribe systems in that there is no flow of information from publishers to subscribers that is mediated by the filtering engine, but rather the publish-subscribe system pulls the data from diverse sources. In contrast to our work, this approach being focused on monitoring has a limited expressiveness in regard to structural constraints for XML documents (only queries of the form “does a document contain an element with tag x containing the string y” are supported). In particular, their algorithm is depending on the average number of atomic events that grows exponentially, if full path expressions are allowed in the query language. Furthermore, Xyleme is catered towards smaller documents (which may be warehoused) and can not process data larger than memory.

♦

Most approaches for XML publish-subscribe systems share many similarities to the NFA approach in [54] and the matching tree algorithm in [2]. Based on the XFilter system [4], several filtering engines for the selective dissemination of information (SDI) represented in XML have been proposed recently. These systems are focused on efficient filtering of (relatively small) XML messages or documents according to subscriptions expressed as XPath queries. The XFilter system establishes the use of deterministic finite automata for filtering of XML data, thus extending the approach of [54] to XML data, and proposes a novel query index optimizing state transitions of the DFAs: An incoming element label is used as key in a hash of all element labels occurring in any subscription. In a hash bucket the states (representing a step in the XPath expression) reachable from the current state by the associated hash key are noted. For each such state s in the hash bucket of the incoming element, all states corresponding to steps following the associated step of s in the subscription are added to the appropriate hash bucket. For the average case this leads to a very efficient selection of the state transitions in the DFAs. As with any hash table the proposed query index can degenerate to linear complexity in the number of subscriptions. As shown in [28] the worst-case complexity of XFilter is $O(N \times 2^d)$ where N is the total number of subscriptions and d is the maximum level of the

[102] Nguyen, B., et al. 2001. Monitoring XML data on the Web. *SIGMOD (ACM Special Interest Group on Management of Data) Record 30,* 2, 437–448.

[54] Gough, J. and Smith, G. 1995. Efficient recognition of events in a distributed system. In *Proc. of the Australasian Computer Science Conference.*

[2] Aguilera, M. K., et al. 1999. Matching events in a content-based subscription system. In *Proc. of the ACM Symposium on Principles of Distributed Computing.* ACM Press, 53–61.

[4] Altinel, M. and Franklin, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the International Conference on Very Large Databases (VLDB).*

[28] Chan, C.-Y., et al. 2002a. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal (Special Issue on XML Data Management).*

document. This worst-case occurs for subscriptions of the form $//x[y = v_1]//x[y = v_2]//\ldots//x[y = v_m]$, i.e., expressions of m steps with the same node test and a predicate between another element and a distinct constant value. On a document consisting of a single path of x elements, the number of states in the hash bucket for x grows exponentially in the depth of the document.

XFilter does not perform multi-query optimization apart of the use of the above discussed query index. In [108] support for the construction of results is provided. Due to the small size of the documents, selection (predicate evaluation) and construction can be separated from matching in this approach, thus allowing a direct application of the techniques from XFilter for the matching part. Simple multi-query optimization is performed by evaluating identical queries only once. The optimization of multiple queries by sharing common prefixes is the principal contribution of YFilter [42; 41]. To enable prefix sharing, an NFA is used instead of multiple DFAs in XFilter. The generated NFA is effectively a trie over the strings representing the structural components similar to the NFA used for matching the regular path expression corresponding to the XPath expression. As expected, experimental evaluation shows a considerable lower processing time compared to XFilter on large number of subscriptions. Nevertheless, the worst-case (space and time) complexity of YFilter is exponential in the number of subscriptions, as the DFA constructed from the NFA has an number of states exponential in the number of states in the NFA and the number of states in the NFA is in worst-case linear in the number of subscriptions. Furthermore, two optimizations for the handling of predicates and nested path expressions (such as in `/a[b/c]/d`) are proposed: Selection postponement delays the evaluation of value-based predicates until a structural match is reached (thus avoiding the evaluation of predicates where no structural match is reached for the remaining expression). This approach closely resembles the handling of selections in the Tukwila system [73]. To evaluate nested path expressions the paths are separated (`/a[b/c]/d` into `/a/d` and `/a/b/c`) and evaluated as separate queries, recording for each match of one of the paths which nodes in the document have been matched. These records are consulted afterwards to find the matching subscriptions. Both optimizations are based on the assumption, that is affordable to store possibly all nodes in a document for further processing, an assumption invalid for unbounded streams as in our case. In [41] the authors argue that their experimental evaluation shows that the cost for matching a subscription is no longer the dominant cost if compared to parsing and further processing and thus no further optimizations (e.g., to avoid the exponential complexity) is necessary. In contrast we believe that their result strengthens that the expressive power of the query language can be further improved without considerably harming the efficiency of the evaluation. In contrast to our work, XFilter and YFilter employ a rather weak query language restricted to child and descendant axes and to value-based and simple structural predicates.

♦

For the open-source “XML Toolkit for Scalable XML Stream Processing” [8], another approach for efficiently processing large numbers of XPath expressions against streams has been proposed in [56]. Similar to YFilter a single NFA for all XPath expressions is constructed. But instead of the construction of an eager DFA with possibly exponential number of states, a lazy DFA is proposed. The main contribution is to show, that under certain assumptions the space and time complexity of the lazy DFA is independent of the number of subscriptions. If one considers only simple path expressions with child and descendant, the number of states in the lazy DFA is at most exponential in the size of the schema, i.e., in the number of elements declared in the schema, and the time for processing a single element from the stream inside the DFA is linear in the size of the schema, as any state in the DFA

[108] Ozen, B., et al. 2001. Highly personalized information delivery to mobile clients. In *Proc. of ACM International Workshop on Data Engineering for Wireless and Mobile Access.*

[42] Diao, Y., et al. 2002. YFilter: Efficient and scalable filtering of XML documents. In *Proc. of the International Conference on Data Engineering (ICDE).*

[41] Diao, Y., et al. 2002. Path sharing and predicate evaluation for high-performance XML filtering. Submitted for publication, `www.cs.berkeley.edu/~diaoyl/publications/yfilter-public.ps`.

[73] Ives, Z. G., et al. 2002. An XML query engine for network-bound data. *VLDB Journal Special Issue on XML Data Management.*

[8] Avila-Campillo, I., et al. 2002. XMLTK: An XML toolkit for scalable XML stream processing. In *Proc. of the Workshop on Programming Language Technologies for XML (PLAN-X).*

[56] Green, T. J., et al. 2003. Processing XML streams with deterministic automata. In *Proc. of the International Conference on Database Technology (ICDT).* 173–189.

can have at most for each element in the schema a different outgoing transition and a hash is used for determining which transition is performed. Note however, that naturally if each element is matched by each subscription, the generation of output bindings requires linear time in the number of subscriptions. This is, of course, true for any XML-based publish-subscribe system. Furthermore, for each state in the DFA a set of corresponding NFA states with size $N \times l$ has to be maintained.

It is important to emphasize, that these results only hold for a rather limited query language, as similarly stressed in YFilter [41]. Most notably, if constant values are used in the expressions (a very common occurrence in XPath expressions) time and space complexity are linear in the number of subscriptions in the worst-case. Moreover, predicate handling is not considered thoroughly, only a naive sketch with complexity $N \times l$ is considered. More elaborate treatment of predicates is not discussed in [56]. As mentioned before, we believe that it is not desirable to reduce further the expressiveness of an XPath-based subscription language, but propose efficient methods for handling a larger subset of XPath. Indeed, recent work [60] extends this approach to XPath expressions with predicates. Predicates can be shared among queries, if they are identical, but there is no sharing between expressions inside and outside predicates.

♦

Abutted to the matching tree described in [2], in [29; 28] a novel index structure called XTrie is proposed. The XTrie indexes substring of XPath expressions rather than individual steps as in XFilter or YFilter. A *substring* of an XPath expression e is defined to be a sequence of element labels, such that there is a path in e consisting in children steps with node-tests corresponding to the element labels in order of there occurrence in the sequence. Furthermore, a *substring decomposition* of an XPath expression e is a set of substrings of e, such that each step of e occurs in at least one substring, and is said to be minimal, if each substring s is of maximal length, i.e., there is no longer substring containing s, e.g. the substrings `ab` and `c` form a minimal decomposition of `/a/b//c`. Finally, a “simple” decomposition of an XPath expression e is a substring decomposition S of e, such that for each branching step v (e.g., the `/b` step in `/a/b[c]/d`) in e their is a maximal substring with last node v in S and all other substrings in S are maximal. For each XPath expression (representing a subscription) the “simple” substring decomposition is determined and the resulting substrings are indexed in a traditional trie. Hence, the space cost of an XTrie is dominated by the number of substrings in each XPath expression, while the space cost of an approach similar to XFilter, indexing individual steps, is dominated by the number of element labels.

Furthermore, [28] provides sophisticated optimizations to reduce the number of unnecessary index probes (e.g., by using a lazy XTrie) and to prune redundant partial matchings. Experimental evaluation indicate that this approach outperforms XFilter consistently. A comparison between YFilter and XTrie has not been performed. The query language is a similar subset of XPath as used in XFilter (only child and descendant axes and simple predicates), but allows ordered matching among siblings by support of the `following-sibling` axes. Order between arbitrary nodes in the document (i.e., `following` axes) is not considered and the proposed technique for sibling order can not trivially be extended to arbitrary nodes.

As stated above, by indexing substrings the space and time complexity of XTrie are bounded by the number of substrings in each XPath expression rather than by the number of elements. More precisely, let $N_S = \sum_{i=1}^{N}$ S_i be the number of substrings in all simple decompositions S_i of all N XPath expressions. Naturally, the XTrie itself has a space complexity of $O(N_S)$, but the algorithm also requires several other data structures with $O(N_S \times d)$, where d is the maximum depth of the document. As the algorithm considers all occurrences of a substring in an XPath expression, the worst-case time complexity is $O(\max(l^2 \times d \times N, N_S))$ with l maximum length of a subscription. Consider-

[41] Diao, Y., et al. 2002. Path sharing and predicate evaluation for high-performance XML filtering. Submitted for publication, `www.cs.berkeley.edu/~diaoyl/publications/yfilter-public.ps`.

[60] Gupta, A. K. and Suciu, D. 2003. Stream processing of XPath queries with predicates. In *Proc. of the Proc. of the ACM SIGMOD International Conference on Management of Data.*

[2] Aguilera, M. K., et al. 1999. Matching events in a content-based subscription system. In *Proc. of the ACM Symposium on Principles of Distributed Computing.* ACM Press, 53–61.

[29] Chan, C.-Y., et al. 2002b. Efficient filtering of XML documents with XPath expressions. In *Proc. of the International Conference on Data Engineering (ICDE).* 235–244.

[28] Chan, C.-Y., et al. 2002a. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal (Special Issue on XML Data Management).*

	Approach	Focus	(Worst-case) Complexity Space	Time
XFilter	N DFAs	transition index	$N \times 2^d$	$2^d \times N \times n$
YFilter	NFA	prefix sharing	$\max(n, 2^{N \times l})$	$2^d \times N \times n$
XTrie	Trie	substring indexing	$N \times l \times d$	$l^2 \times d \times N \times n$
XMLTK	single (lazy) DFA	limited-size schema	$N \times l \times 2^S$	$\min(S, N) \times n$
with selection on value:			$\max(2^S, N) \times N \times l$	$l \times N \times n$
MatchMaker		specific index structure	$\max(n, N \times \max(d, l))$	$\max(d, l) \times N \times n$

Table 3.2: Comparison of matching engines for XML-based publish-subscribe systems. Let N be the number of subscriptions, l the maximum length of a subscription, d the depth of the document, n the length of the document, and S the size of the schema for the XML stream.

ing N XPath expressions with descendant axis only and where no element label occurs twice, the number of substrings N_S is bound to the number of steps of all XPath expressions, i.e., to $l \times N$. Thus, the overall worst-case complexity for processing a node is linear in the number of subscriptions and their length, $O(l^2 \times L \times N)$.

♦

Based on a novel index structure, [83] presents a system, called MatchMaker, for efficient matching of large numbers of queries (subscriptions) against an XML stream, where the size of the document is small compared to the number of queries. Special consideration is given to chain queries, i.e., single path queries (without structural predicates). The matching of (chain or tree) queries to a document is specified as a labeling problem, where a node in the document is labeled with all queries selecting that node. The main contribution is a novel index structure, called *dual index*, that can efficiently support the following three types of queries for labels l_1 and l_2, viz., which queries start with l_1, which queries contain l_1/l_2, i.e., l_1 and l_2 in a parent-child relation, and which queries contain $l_1//l_2$, i.e., l_1 and l_2 in an ancestor-descendant relation (at any position). This index is implemented using hash tables. Algorithms based on this index for chain and tree queries are presented distinguished from the above discussed systems in that all matchings of a query are determined, instead of only deciding whether or not a query matches a document. The main disadvantage is, that two passes over the stream are required, i.e., the entire stream has to be buffered in any case. Worst-case space and time complexity are given in Table 3.2. Note, that the used query language is similar in expressiveness to the one used in, e.g., XFilter, only supporting child and descendant axes and structural predicates.

♦

Apart of the aforementioned XML-based publish-subscribe systems, summarized in Table 3.2, in [113] a very fast XML-based publish-subscribe system called WebFilter based on the ideas of Le Subscribe [47] (cf. Section 3.3.1) is proposed, though no details on the matching of event paths to subscriptions is presented. The first XML-based distributed publish-subscribe system is described in [3] with focus on reliable transmission. No consideration of efficient matching of XML documents to subscriptions is provided, but a simple XPath engine (based on the Gnome libxml library) is used. Finally, in [77] an architecture called MDV for distributed meta-data management is proposed. RDF data is filtered using an approach similar to traditional trigger systems.

In Table 3.2 important characteristics of various XML-based publish-subscribe systems are summarized. Note, that all of the systems have a limited ex-

[83] Lakshmanan, L. V. and Parthasarathy, S. 2002. On efficient matching of streaming XML documents and queries. In *Proc. of the International Conference on Extending Database Technology (EDBT)*. 142–160.

[113] Pereira, J., et al. 2001. WebFilter: A high-throughput XML-based publish and subscribe system. In *Proc. of the International Conference on Very Large Databases (VLDB)*. 723–724.

[47] Fabret, F., et al. 2001. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 115–126.

[3] Alex C. Snoeren, Kenneth Conley, D. K. G. 2001. Mesh-based content routing using XML. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*. 160–173.

[77] Keidl, M., et al. 2002. A publish & subscribe architecture for distributed metadata management. In *Proc. of the International Conference on Data Engineering (ICDE)*. 309–320.

```
for $X in $R/a return
    for $Y in $X/b return
        <res>$Y,$X</res>
```

Figure 3.2: Example query in [87].

pressiveness, restricting queries to child and descendant axes and (if at all) to simple predicates. Furthermore, with the notable exception of XMLTK without selections on value, all systems have space and time complexity for matching at least linear in the number of subscriptions. No comparison of the average performance is given, as all of the more recent systems (YFilter, XTrie, and XMLTK) provide experimental evaluation only in respect to XFilter.

3.4 Single Query Processors against XML Streams

Recently, several query processors for single XPath or XQuery expressions against XML streams have been proposed. These proposals are set apart from the publish-subscribe systems discussed in the previous section by the fact, that no consideration to multi-query optimization is given. Consequential, they are usually tailored to support a larger subset of XPath (or XQuery) than the previously discussed systems, thus providing a higher expressiveness at the cost of lower scalability. Furthermore, the material problem is to allow an efficient evaluation over large, possibly unbounded documents rather than a stream of small documents. A subset of XPath that can be easily evaluated in a single run is identified in [40]. In [107] it is shown, that the reverse axes of XPath can be implemented using forward axes only. For an overview of the discussed systems and their complexity refer to Table 3.3.

In [87] an XQuery processor, called XML stream machine (XSM), based on finite state transducers combined with buffers is presented. As they focus on the evaluation of joins enabled by the use of buffers, the supported subset of XQuery regards joins and element creation, but is restricted to a descendant-like axis (with limited expressiveness, as non-recursive data is assumed, thus precluding nested occurrences of elements) and value-based predicates, cf. Query 3.2 for an example. The support of joins and construction immediately mandates the use of buffers for each variable in the body (return or where clause) of an FLWR expression, that is not the loop variable of that FLWR expression, thus leading to a space complexity linear in the size of the stream. Furthermore, as the result of a query can be exponential in the size of the original stream (e.g., Query 3.2), the worst-case time complexity is $O(n^l)$ where n is the size of the stream and l is the length of the query. Experimental evaluation (without nested queries) points to the expected linear time complexity in the size of the data for queries with linear data complexity, i.e., for queries without join and construction. We believe, that these results strengthen our position, that (exact) joins and construction over multiple dimensions are inappropriate for evaluation against possibly unbounded streams.

♦

The $\chi\alpha o\varsigma$ algorithm presented in [13; 14] is a streaming algorithm for handling both forward and reverse axes. Note however, that no horizontal axes, such as following or preceding, are supported. Also only simple structural predicates are considered. The handling of the horizontal reverse axes (and the resulting query tree, called X-Dag) resembles closely the first approach presented in [107]: An expression like /descendant::n/ancestor::m is treated by selecting the n node by two paths, viz., /descendant::n and /descendant::m/descendant::n. The presented algorithm $\chi\alpha o\varsigma$ finds all matchings of a query in $O(l \times n^2)$ space and time for matching where n is the size of the document and l the number of steps in the query. Experimental evaluation in [13] indicates that the $\chi\alpha o\varsigma$ algorithm performs slightly better than a conventional XPath engine such as Xalan.

♦

Based on hierarchical pushdown transducers (HPDT), the XSQ system [112; 111] has many simi-

[40] Desai, A. 2001. Introduction to sequential XPath. In *Proc. of the IDEAlliance XML Conference.*

[107] Olteanu, D., et al. 2002. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM).* Lecture Notes on Computer Science (LNCS), vol. 2490. Springer Verlag, 109–125.

[13] Barton, C., et al. 2002. An algorithm for streaming XPath processing with forward and backward axes. In *Proc. of the Workshop on Programming Language Technologies for XML (PLAN-X).*

[14] Barton, C., et al. 2003. Streaming XPath processing with forward and backward axes. In *Proc. of the International Conference on Data Engineering (ICDE).*

[112] Peng, F. and Chawathe, S. S. 2003b. XSQ: Streaming XPath

Approach		Focus	(Worst-case) Complexity	
			Space	Time
XSM	FSM with buffers	joins, construction	$n \times l$	n^l
$\chi\alpha o\varsigma$		parent, ancestor	$l \times n^2$	$l \times n^2$
XSQ	HDPDT	simple predicates, aggregation	$n + 2^l$	$n \times l$
SPEX	network of DPDTs	RPQ: generic predicates, reverse axes	$l \times n \times d$	$n \times d \times l$

Table 3.3: Comparison of single query processors. Always a single query is assumed. Let l be the length of the query and n the size of the document.

larities with the fundamental approach of SPEX. The evaluation model is based on a hierarchy of transducers similar to a network of transducers as in our system. But there are two important differences in the evaluation models: Most importantly the number of transducers in the XSQ system is exponential in the length of the query (more precisely in the number of the predicates occurring in the query), whereas a SPEX network is always linear in the length of a query. Furthermore, the various pushdown transducers constituting a hierarchy are generated in the XSQ system based on templates for certain predicates, whereas SPEX provides a generic method for predicate handling. Thus, the language supported by XSQ only provides certain predicates, viz., value-based and simple structural predicates, in particular no nested predicates or multi-step predicates, and no support for horizontal or reverse axes, such as following or parent. Some consideration for aggregations is given, that might also be considered in future versions of SPEX. Experimental evaluation in [112; 111] points to significant improvements compared to traditional XPath engines such as Xalan, but also shows that a system with a more restricted language, in particular without predicate handling, such as XMLTK [56] can outperform this approach. However, the time for evaluation of a single query is consistently lower than the time for parsing.

♦

The SPEX evaluation model this work is based upon is discussed in Chapter 7. Here, only the complexity is considered for comparison. In [106], it is shown that the SPEX evaluation model has time *and* space complexity $O(n \times d \times l)$ where n is the size, d the depth of the stream, and l the size of the query.

This overview of related work establishes that none of the previous approaches for the optimization of multiple queries on XML streams has considered the sharing of operators over the entire query graph instead of prefixes only. Furthermore, no systematic description of the problem and its properties has been given so far. In the next chapter, a formal description of logical query plans is introduced as bases for the problem description in Chapter 5.

queries. In *Proc. of the International Conference on Data Engineering (ICDE)*.

[111] Peng, F. and Chawathe, S. S. 2003a. XPath queries on streaming data. In *Proc. of the Proc. of the ACM SIGMOD International Conference on Management of Data*.

[56] Green, T. J., et al. 2003. Processing XML streams with deterministic automata. In *Proc. of the International Conference on Database Technology (ICDT)*. 173–189.

[106] Olteanu, D., et al. 2003. Advanced techniques for streamed and progressive evaluation of XPath. Research report, University of Munich, Institute for Computer Science.

Chapter 4

Concise Representation of XML Query Plans

As foundation for the discussion of the optimization methods proposed in the next chapters, a concise, yet powerful formal representation for a (logical) query plan is established in this chapter together with its properties.

Contents

4.1 Formalization of a Query Plan

To facilitate a formal description of our problem, how to generate an optimal query plan for several queries or query plans, a precise definition of a query plan and its properties is required.

4.1.1 Evaluation Model

The concrete structure and generation of a query plan depends naturally on the query language and logical algebra employed in the targeted evaluation engine. In the following, the properties of an evaluation engine that have influence on the definition of a query plan, are described by means of an *evaluation model* $\mathcal{E}$. An evaluation model specifies the (possibly infinite) sets of queries $Q_\mathcal{E}$ and query plans $P_\mathcal{E}$ that are considered legal together with a translation $t_\mathcal{E}$ from queries to query plans that gives for each query a *set* of query plans that can be used to evaluate that query in this evaluation model. In general, there are (possibly infinite) many query plans associated with a single query, so that the optimizer can select among these query plans the one with the lowest expected cost for evaluation. The cost of a query plan is also specified as part of the evaluation model by means of a function $c_\mathcal{E}$ that assigns to each query plan the expected cost for evaluation.

In Section 2.3.2, query plans are introduced as graphs with operators of the logical algebra as vertices. An evaluation model $\mathcal{E}$ specifies the set of operators $O_\mathcal{E}$ that are valid in a query plan for that evaluation model. Furthermore, vertices in a query plan can have several properties (such as the actual label of a label operator in the query plans shown in Section 2.3.2). Therefore, the evaluation model $\mathcal{E}$ determines not only the set $R_\mathcal{E}$ of valid properties under this evaluation model, but also which properties can be associated with an operator, and provides means to determine, whether and how two properties can be merged into a single one: "merging" here indicates that the resulting property entails both original properties in such a way, that both properties can still be evaluated (cf. Chapter 7 for a discussion of multi-properties in SPEX).

Formally, an **evaluation model** $\mathcal{E}$ is a octuple $(Q_\mathcal{E}, P_\mathcal{E}, t_\mathcal{E}, c_\mathcal{E}, O_\mathcal{E}, R_\mathcal{E}, r_\mathcal{E}, \mu_\mathcal{E})$ where

—$Q_\mathcal{E}$ is the set of valid queries for $\mathcal{E}$,
—$P_\mathcal{E}$ is the set of valid query plans for $\mathcal{E}$,

—$t_{\mathcal{E}} : Q_{\mathcal{E}} \to \wp(P_{\mathcal{E}})$ associates each query with the set of query plans that evaluate the query,
—$c_{\mathcal{E}} : P_{\mathcal{E}} \to \mathbb{R}$ is the cost function assigning to each query plan a cost under that evaluation model, it is required that the cost of a query plan can be computed in polynomial time,
—$O_{\mathcal{E}}$ is the set of operators in query plans from $P_{\mathcal{E}}$,
—$R_{\mathcal{E}}$ is the set of properties in query plans from $P_{\mathcal{E}}$,
—$r_{\mathcal{E}} : O_{\mathcal{E}} \to \wp(R_{\mathcal{E}})$ maps each operator to the set of properties that are allowed for that operator, and
—$\mu_{\mathcal{E}} : R_{\mathcal{E}} \times R_{\mathcal{E}} \to R_{\mathcal{E}}$ is a partial function that assigns to pairs of properties the property resulting from merging the two if they can be merged. It is assumed, that for all $p \in R_{\mathcal{E}}$ $\mu_{\mathcal{E}}(p,p) = p$.

In this section, we concentrate on the properties of an evaluation model that define the operators and properties of vertices in a query plan. The characterization of valid queries and query plans, as well as their relation is not detailed in this work, except exemplarily in Chapter 7 on the SPEX evaluation model, but it is assumed, that there is some way to determine the valid queries and query plans, in particular whether a query plan can be used to evaluate a given query.

4.1.2 Query Plan

Based on such an evaluation model, it is now easy to formally define a query plan: A **query plan** P for an evaluation model $\mathcal{E}$ is a quadruple (G, τ, π, q) where

—$G = (V, E)$ is a *directed graph* with vertices V and edges E,
—$\tau : V \to O_{\mathcal{E}}$ assigns to each vertex in G an operator,
—$\pi : V \to R_{\mathcal{E}}$ is a partial function, that associates to some vertices in G a property, such that, for all vertices v, if there is a property $r' \in R_{\mathcal{E}}$ with $r' = \pi(v)$ then $r' \in r_{\mathcal{E}}(\tau(v))$, i.e., each vertex can have at most one property assigned to it and that property must be allowed for the operator the vertex represents (for ease of notation, we understand in the following, for all vertices $v, w \in V$, $\pi(v) = \pi(w)$ as $(\exists r \in R_{\mathcal{E}} r = \pi(v) \implies \exists r' \in R_{\mathcal{E}} r' = \pi(w) \wedge r = r') \vee (\nexists r \in R_{\mathcal{E}} r = \pi(v) \implies \nexists r' \in R_{\mathcal{E}} r' = \pi(w))$, otherwise $\pi(v) \neq \pi(w)$),
—$q : E \to \wp(Q_{\mathcal{E}}) \setminus \varnothing$ maps each edge in G to a non-empty set of queries this edge is part of,

Each vertex has an operator type, but not all vertices have a property assigned to it. Only edges are assigned to queries as the incident vertices of an edge are naturally relevant for all queries that edge is part of. Therefore, it is sufficient to assign queries to edges (with the slight exception of isolated vertices, which, for reasons of conciseness, are not considered here). For convenience, we extend q to vertices in the following way: Let $edges(v \in V) = \{e \in E : \exists y \in V : e = (x, y) \vee e = (y, x)\}$ be the incident edges of a vertex v), then

$$q' : V \cup E \to \wp(Q_{\mathcal{E}}) :$$
$$x \in V \cup E \mapsto \begin{cases} q(x) & x \in E \\ \bigcup_{e \in edges(x)} q(e) & x \in V \end{cases}$$

Unless mentioned otherwise, this extension of q is used in the following.

To ease the discussion of query plans for multiple queries, two further definitions are helpful: First, we naturally extend the notion of isomorphism from graphs to query plans, i.e., two query plans $P_1 = (G_1, \tau_1, \pi_1, q_1)$ and $P_2 = (G_2, \tau_2, \pi_2, q_2)$ for the same evaluation model $\mathcal{E}$ are **isomorphic**, denoted by $P_1 \simeq P_2$, if there is a bijection $\phi : V_{G_1} \to V_{G_2}$ such that for all x and $y \in V_{G_1}$

—$(x, y) \in E_{G_1}$ is equivalent to $(\phi(x), \phi(y)) \in E_{G_2}$, (the images of vertices adjacent in G_1 are adjacent in G_2),
—$\tau(x) = \tau(\phi(x))$ (the operator assigned to a vertex and its image are identical),
—either $\pi(x)$ and $\pi(\phi(x))$ are both undefined or $\pi(x) = \pi(\phi(x))$ (the property assigned to a vertex and its image are coherent),
—if $(x, y) \in E_{G_1}$, $q((x, y)) = q((\phi(x), \phi(y)))$ (the queries assigned to corresponding edges in the two graphs are identical).

Second, the **restriction** of a query plan $P = (G, \tau, \pi, q)$ to a query Q is defined as a query plan $P|_Q = (G_{P|_Q}, \tau_{P|_Q}, \pi_{P|_Q}, q_{P|_Q})$ where

—$G_{P|_Q} = (V', E')$ with $E' = \{\, e \in E_G \mid Q \in q(e) \,\}$ and $V' = \{\, v \in G_V \mid Q \in q(v) \,\}$ (the graph G restricted to those edges that are assigned to Q and their adjacent vertices),

—$\tau_{P|_Q} = \tau|_{V'}$ (ordinary restriction on functions),

—$\pi_{P|_Q} = \pi|_{V'}$ (ordinary restriction on functions),

—$q : E' \to \wp(Q_{\mathcal{E}}) \setminus \varnothing$ is now a constant function mapping all edges $e \in E'$ to the singleton set $\{Q\}$.

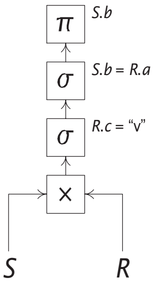


Figure 4.1: Query plan for `SELECT S.b FROM S, R WHERE S.b = R.a AND R.c = "v"`

♦

It is crucial to observe that the definitions of evaluation model and query plan given above do not restrict the shape and properties of a query plan in any way except that they require that a query plan can be represented as a digraph. Since any undirected graph can be represented as a directed graph (with double the number of edges), this requirement is actually no restriction. The information about the kind of graphs that are valid as query plans for a specific evaluation model is contained in the translation function $t_{\mathcal{E}}$ that associates queries with valid query plans for evaluating them. When discussing the problem, how to find an optimal query plans for multiple queries, and heuristics for solving it, more specific knowledge about the characteristics of a valid query plan under a certain evaluation model will prove very beneficial, cf. Chapter 6.

For this reason and for illustrating the just introduced formal notions of evaluation model and query plan, a closer look at two concrete examples for evaluation models and the kind of query plans they allow is indicated.

4.2 Use Case: Traditional Relational Query Plans

Figure 4.2 shows the initial query plan for the relational query $\pi_{S.b}(\sigma_{S.b=R.a}(\sigma_{R.c=\text{“}v\text{”}}(R \times S)))$ as discussed in Section 2.1.1.

Formally, this query plan P is defined as the quadruple (G, τ, π, q), where $G = (V, E)$ is the graph depicted, with six vertices $V = \{v_1, \dots v_6\}$ and five edges $E = \{(v_1, v_3), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_6)\}$. For the definition of τ, π, and q the evaluation model this query plan is based on is required: Informally, a possible evaluation model for relational queries $\mathcal{R}$ uses the set of operators of the relational algebra plus an access operator for accessing relations. Each operator has different properties, e.g., projection operators have the property on which set of attributes to project, selection operators carry the selection expression, and access operators the relation accessed by them. Using the textual representation of an operator or property as identifier, $\tau : \{v_1 \mapsto \text{acccess}, v_2 \mapsto \text{access}, v_3 \mapsto \times, v_4 \mapsto \sigma, v_5 \mapsto \sigma, v_6 \mapsto \pi\}$ and $\pi : \{v_1 \mapsto \text{R}, v_2 \mapsto \text{S}, v_4 \mapsto \text{R.c = “v”}, v_5 \mapsto \text{S.b = R.a}, v_6 \mapsto S.b\}$. q maps each edge to the singleton set containing only the query shown above. The graphical representation used for relational query plans so far can therefore easily be mapped to a more formal specification as indicated here.

One might observe, that the query plans under this evaluation model $\mathcal{R}$ can be characterized by the properties of the underlying graph: Actually, any graph is a valid query plan for some relational query, if it adheres to the following four restrictions:

—the graph is *acyclic*,

—the graph is *connected*, if the query plan as in the examples evaluates a single query only,

—its vertices can be assigned to operators and properties in a way consistent with $\mathcal{R}$, and

—all sources of the graph (i.e., vertices without incoming edge) are access operators and access operators are assigned to sources only,

These conditions are sufficient and required for any graph to be a query plan under $\mathcal{R}$ for some relational query (to which the edges of the query plan can be assigned).

4.3 Use Case: Query Plans for XML Streams

In Section 2.3.2 the notion of query plans for querying XML streams has been introduced. Here, a formal interpretation of the graphical representation in that section is presented.

First, an evaluation model has to be specified. As in Section 2.3.2, the RPQ semantics is used as basis for that evaluation model $\mathcal{X}$: The set of operators for query plans in $\mathcal{X}$ is the set of operators in RPQ, i.e.,

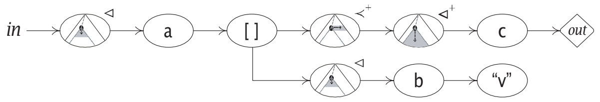


Figure 4.2: Query plan for query from Figure 2.4

all relation and property operators (label and text operator), enhanced with the structural operators: the input, output, predicate, intersection, and union operator. The set of properties is the (infinite) set of restrictions on label and text of an element allowed by RPQ, partitioned by $r_{\mathcal{X}}$ into properties assignable to label and text operators.

Based on this evaluation model $\mathcal{X}$, the query plan for the query $Q(\nu 4)$:- $\nu 0 \lhd \nu 1 \wedge \mathsf{a}(\nu 1) \wedge \nu 1 \lhd \nu 2 \wedge \mathsf{b}(\nu 2) \wedge \text{“v”}(\nu 2) \wedge \nu 1 \prec^{+} \nu 3 \wedge \nu 3 \lhd^{+} \nu 4 \wedge \mathsf{c}(\nu 4)$ shown in Figure 4.2, can be formally interpreted as a quadruple $P = (G, \tau, \pi, q)$, where

—$G = (V, E)$ is the graph as depicted, i.e., the graph with 11 vertices $V = \{\nu_1, \ldots, \nu_{11}\}$ and 10 edges $E = \{(\nu_1, \nu_2), (\nu_2, \nu_3), (\nu_3, \nu_4), (\nu_4, \nu_5), (\nu_5, \nu_6), (\nu_6, \nu_7), (\nu_7, \nu_8), (\nu_4, \nu_9), (\nu_9, \nu_{10}), (\nu_{10}, \nu_{11})\}$,

—τ maps each vertex to the operator shown, i.e., $\tau = \{\nu_1 \mapsto \mathit{in}, \nu_2 \mapsto \lhd, \nu_3 \mapsto \mathit{label}, \nu_4 \mapsto [\,], \nu_5 \mapsto \prec^{+}, \nu_6 \mapsto \lhd^{+}, \nu_7 \mapsto \mathit{label}, \nu_8 \mapsto \mathit{out}, \nu_9 \mapsto \lhd, \nu_{10} \mapsto \mathit{label}, \nu_{11} \mapsto \mathit{text}\}$,

—π maps vertices to the properties they carry, i.e., $\pi = \{\nu_3 \mapsto \mathsf{a}, \nu_7 \mapsto \mathsf{c}, \nu_{10} \mapsto b, \nu_{11} \mapsto \text{“v”}\}$, and

—q maps each edge to the original query.

Again, it is most revealing to note some of the properties of such a query plan:

—These query plans are by definition once more *acyclic* as the underlying query language RPQ does not entail recursive expressions.

—As in the previous case, their input operator must be a source of the graph and any input operator occurring in the graph must be a source. Furthermore, there has to be exactly one output operator for each query evaluated by the query plan, although the same output operator can be part of several queries.

—Of course, there must be a way, to assign all vertices of the graph to operators and properties in a way consistent with $\mathcal{X}$.

These properties hold for query plans evaluating single queries such as the one discussed above, as well as for query plans covering multiple queries, cf. Figure 2.15(a).

Any graph that respects these three properties, is a query plan for some RPQ query in $\mathcal{X}$ (to which the edges can be assigned to). Note, in particular, that a query plan does not have to be connected, although one might observe that all connected components in the query plan have as source the same operator, viz. the input operator, that has no properties (we consider a single data source, the stream of XML data, only), thus can be merged for all connected components, leading to a connected graph with a single source.

Based on the formal representation of an evaluation model established in this chapter, the problem of multi-query optimization by operator sharing as introduced in Section 2.3.4 is formalized in the following chapter.

Chapter 5

The Minimum Common Super-Plan Problem

This chapter finally formalizes the problem of finding an optimal query plan for the simultaneous evaluation of multiple queries. The problem is formalized as an optimization problem and its properties with respect to complexity and approximability are investigated by comparison and reduction from similar problems mostly form graph theory.

Contents

In Section 2.3.4, the core objective of this work is established: optimize multiple queries by computing a query plan that evaluates all queries and can be evaluated efficiently. Extending previous work using some form of prefix compaction for tree-shaped query plans [4; 28], we concentrate on finding an optimal way to share operators among the queries. Figure 2.15(a) illustrates this approach with a possible common query plan for the queries from Figure 2.4 and 2.13 based on the query plans from Figure 2.8 and 2.14(b). Recall, that the edges are labeled with the queries (abbreviated by 1 and 2) and that operators belonging to the first query only are colored in blue, those part of the second query only in red, and shared operators remain black.

In this chapter, the proposed approach to optimization of multiple queries is elaborated and formalized as an optimization problem. The following section provides a short reexamination of relevant definitions and notations concerning optimization problems and their properties.

[4] Altinel, M. and Franklin, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the International Conference on Very Large Databases (VLDB).*

[28] Chan, C.-Y., et al. 2002a. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal (Special Issue on XML Data Management).*

5.1 Complexity and Approximability of Optimization Problems

5.1.1 Optimization Problems

Following [35; 75; 7] finding a feasible solution for any valid input instance of a certain problem is considered an optimization problem, if the sought-after solution is optimal with respect to some measure of quality associated with a solution and the optimization objective. Formally, an **optimization problem** Π over an alphabet Σ is described by a quadruple $(I_\Pi, S_\Pi, m_\Pi, \text{goal}_\Pi)$, where

(1) $I_\Pi \subseteq \Sigma^*$ is the space of *input instances.*
(2) $S_\Pi : I_\Pi \to \Sigma^*$ associates with each input instance $x \in I_\Pi$ the space of *feasible solutions* for x.
(3) $m_\Pi : I_\Pi \times \Sigma^* \to \mathbb{R}_0^+$ is the measure or *objective function* specifying for each pair (x, y) such that $x \in I_\Pi$ and $y \in S_\Pi(x)$ a positive number indicating the quality of the solution y under input instance x.
(4) $\text{goal}_\Pi \in \{\min, \max\}$ indicates whether Π is a maximization or a minimization problem.

For an input instance x, the set of optimal solutions of x is denoted by $S^*_\Pi(x) = \{y \in S_\Pi(x) : m_\Pi(x, y) = \text{goal}_\Pi\{\, n \in \mathbb{R}_0^+ \mid \exists z \in S_\Pi(x) : n = m_\Pi(x, z)\,\}$. Obviously, all optimal solutions have the same quality that will be denoted as $\text{opt}_\Pi(x)$, i.e., $\text{opt}_\Pi(x) = m_\Pi(x, y)$ for any $y \in S^*_\Pi(x)$.

One should observe, that for each optimization problem Π there is a corresponding decision problem Π_D: The decision problem asks whether there exists a feasible solution y of an instance x with a quality bounded by some $K > 0$ for a maximization or $K < 0$ for a minimization problem. Furthermore, if the optimization problem can be solved in polynomial time by a deterministic algorithm, so can the decision problem. In other words, if the decision problem is already NP-complete, the optimization problem can as well not be solved in polynomial time by a deterministic algorithm unless P = NP.

[35] Crescenzi, P. and Panconesi, A. 1991. Completeness in approximation classes. *Information and Computation 93,* 2, 241–262.

[75] Kann, V. 1992. On the approximability of the maximum common subgraph problem. In *Proc. 9th Symp. Theoretical Aspects of Computer Science.* Number 577 in Lecture Notes in Computer Science. Springer Verlag, 377–388.

[7] Ausiello, G., et al. 1999. *Complexity and Approximation: Combinatorial Optimization Problems and their Approximability Properties.* Springer Verlag, Berlin.

5.1.2 NPO Problems

Following this observation, optimization problems can be divided into two classes by their inherent complexity: the class PO for which a deterministic polynomial-time algorithm exists and the class NPO for which a non-deterministic polynomial-time algorithm is known. These classes strictly correspond to the classes for decision problems P and NP, in particular the notion of hardness can be extended from a decision problem to its corresponding optimization problem. Hence, P ≠ NP implies PO ≠ NPO and vice versa.

More formally, an optimization problem $\Pi = (I, S, m, \text{goal})$ belongs to the class NPO and is called an **NPO problem** if it is short and easy-to-recognize, i.e., if

(1) the space of instances I can be recognized in polynomial time,
(2) the solutions are short, i.e., the size of a solution is reasonable close to the size of the input instance. Formally, it is required, that there exists a polynomial p such that, for any $x \in I$ and $y \in S(x)$, $|y| \le p(|x|)$,
(3) the solutions are easy to recognize, i.e., for any y such that $|y| \le p(|x|)$ it is decidable in polynomial time whether y is a solution for x, and
(4) the objective function is computable in polynomial time.

It is easy to see, that the corresponding decision problem of an NPO problem is in NP.

An optimization problem Π is called **NP-hard**, if every decision problem $\Pi' \in$ NP can be solved in polynomial time by an algorithm which uses an oracle that, for any instance $x \in I_\Pi$, returns an optimal solution $y \in S^*_\Pi(x)$ together with its value $\text{opt}_\Pi(x)$. Therefore, if the corresponding decision problem Π_D of an NPO problem Π is NP-complete, Π is NP-hard, since Π_D can be solved in the above described manner and all other NP problems can be solved by an algorithm that solves Π_D.

An NPO problem Π is said to be *polynomially bounded* if a polynomial p exists such that, for any instance x and for any solution $y \in I_\Pi(x)$, $m_\Pi(x, y) \le p(|y|)$. The class NPO PB is the set of polynomially bounded NPO problems.

5.1.3 Approximability of NP-hard Problems

Since the optimization problems discussed later in this chapter prove to be NP-hard, a closer look at this class of problems is indicated. As there is no deterministic polynomial-time algorithm known for solving problems from this class, we sacrifice optimality and start looking for approximate solutions computable in polynomial time. Of particular interest is the question, how well such a problem can be approximated, i.e., whether an approximate solution can be guaranteed to be still reasonably close to an optimal solution. The following notations are based on [67; 6; 7].

To measure the quality of an approximation algorithm, several notions can be used (e.g., absolute or relative error). Following [6; 7], the performance ratio of an approximation algorithm is employed here for determining the quality of the approximation provided.

For an optimization problem Π, the **performance ratio** $R(x, y)$ of a solution y under an input instance x for Π is the ratio of the quality of the solution to the quality of the optimal solution for x, i.e.,

$$R(x, y) = \max\left(\frac{m_\Pi(x, y)}{\text{opt}_\Pi(x)}, \frac{\text{opt}_\Pi(x)}{m_\Pi(x, y)}\right).$$

By definition, the performance ratio of a solution is always ≥ 1. Furthermore, this definition allows a unified treatment of minimization and maximization problems with regard to the performance ratio of a solution.

Based on the performance ratio of the solution an approximation algorithm computes, the quality of that algorithm can now be judged: An approximation algorithm $\mathcal{A}$ for an optimization problem Π is called $r(n)$-**approximate algorithm** for P where $r : \mathbb{N}_0 \to \mathbb{R}_0^+$ is a function, if, for any instance x such that $S_\Pi(x) \neq \emptyset$, the performance ratio of the feasible solution $\mathcal{A}(x)$ with respect to x is bounded by $r(|x|)$, i.e.,

$$R(x, \mathcal{A}(x)) \leq r(|x|).$$

If an optimization problem admits an $r(n)$-approximate deterministic polynomial-time algorithm we say that it is *approximable within* $r(n)$.

Finally, an algorithm $\mathcal{A}$ for an optimization problem Π is said to be an **approximation scheme** for Π, if it returns, for any instance $x \in I_\Pi$ and for any rational $\epsilon > 1$, feasible solution of x whose performance ratio is at most ϵ.

♦

These definitions can be used to classify optimization problems based on their approximability, i.e., on whether there is a (deterministic) approximation scheme or an $r(n)$-approximate algorithm with polynomial complexity for that problem (all the subset relations are strict if P ≠ NP):

(1) APX ⊆ NPO is the class of NPO problems, such that there exists a deterministic $r(n)$-approximate algorithm for *some* constant function r (naturally, $r(n) \geq 1$ for all $n \in \mathbb{N}$. Similarly, one can define classes log-APX ⊆ poly-APX ⊆ exp-APX ⊆ NPO where r is a logarithmic, polynomial, or exponential function.

(2) PTAS ⊆ APX is the class of NPO problems that adhere to an deterministic approximation scheme with polynomial complexity in the size of the input instance, i.e., an deterministic approximation scheme with time complexity bounded by $p(|x|)$ for some polynomial p and all input instances x. Note, that the approximation scheme can still be exponential in the approximation bound ϵ, i.e., it can be $2^{1/(\epsilon-1)}p(|x|)$ or $|x|^{1/(\epsilon-1)}$.

(3) FPTAS ⊆ PTAS is the class of NPO problems with a fully polynomial deterministic approximation scheme, i.e., an deterministic approximation scheme with time complexity bounded by $p(|x|, 1/(\epsilon - 1))$ for some polynomial p.

Even more interesting, than the mere inclusion of a problem in some of these classes, is naturally the negative result, that a problem can not be approximated better than any problem in a certain class. This leads to the notion of completeness in the different approximability classes. To that end, a reduction of one optimization problem to another is needed that preserves the approximability features. Several such reductions are proposed in the literature (gap-preserving reduction [67], AP- or PTAS-reduction [7], E-reduction [78], F- and P-reduction [35]), here the L-reduction [110] is

[67] Hochbaum, D., Ed. 1996. *Approximation Algorithms for NP-hard Problems*, 1st ed. Brooks Cole.

[6] Arora, S. 1998. The approximability of NP-hard problems. In *Proc. of the ACM Symposium on Theory of Computing*. 337–348.

[78] Khanna, S., et al. 1999. On syntactic versus computational views of approximability. *SIAM Journal on Computing 28,* 1, 164–191.

[35] Crescenzi, P. and Panconesi, A. 1991. Completeness in ap-

employed, that represents essentially the strictest notion of reduction, since it requires that the relative error of an approximated solution in comparison to an optimal solution for one problem is linearly related to the relative error for the other problem:

Let A and B be two optimization problems in NPO. A is said to be **L-reducible** to B, in symbols $A \leq_L B$, if two functions f and g and two positive constants α and β exist such that,

(1) $f : I_A \to I_B$ maps instances of A to instances of B, such that for any $x \in I_A$, if $S_A(x) \neq \emptyset$ then $S_B(f(x)) \neq \emptyset$.
(2) $g : I_A \times S_B \to S_A$ maps instances of A and solutions for B to solutions from A, i.e., for any $x \in I_A$ and any $y \in S_B(f(x))$, $g(x,y) \in S_A(x)$.
(3) f and g are computable in polynomial time in the size of their input parameters.
(4) For any $x \in I_A$, $\mathrm{opt}_B(f(x)) \leq \alpha \, \mathrm{opt}_A(x)$, i.e., the quality of the optimal solutions is linearly related.
(5) For any $x \in I_A$ and for any $y \in S_B(f(x))$, the relative error of the solutions is linearly related:

$$|\,\mathrm{opt}_A(x) - m_A(x, g(x,y))| \leq \beta\,|\,\mathrm{opt}_B(f(x)) - m_B(f(x),y)|.$$

It follows from the definition, that if $A \leq_L B$ and $B \in$ APX (respectively, $B \in$ PTAS), then $A \in$ APX (respectively, $A \in$ PTAS).

Based on this notion of reducibility, we can now define the classes of problems that are in one of the approximation classes and can not be approximated better: Given a class C of NPO problems (where C can be the entire class of NPO problems, the class of polynomial-bounded NPO problems NPO PB, or one of the approximation classes defined above), a problem Π is C-**hard** (with respect to the L-reducibility) if, for any $\Pi' \in C$, $\Pi' \leq_L \Pi$. A C-hard problem is C-**complete** (with respect to the L-reducibility) if it belongs to C.

In [7] it is shown that an NPO-complete problem (an NPO PB-complete problem) can *not* be approximated within 2^{n^ϵ} (n^ϵ) for any $\epsilon > 0$ unless P = NP.

proximation classes. *Information and Computation 93,* 2, 241–262.

[110] Papadimitriou, C. H. and Yannakakis, M. 1991. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences 43*, 425–440.

[7] Ausiello, G., et al. 1999. *Complexity and Approximation: Combinatorial Optimization Problems and their Approximability Properties.* Springer Verlag, Berlin.

5.2 Minimum Common Super-Plan

The previous section establishes a framework in which the problem how to find an optimal query plan for the simultaneous evaluation of multiple queries can be formalized and investigated:

The problem to find the optimal query plan for the simultaneous evaluation of multiple queries under a given evaluation model $\mathcal{E}$, referred to as **minimum common super-plan** for a set of queries, is formally defined as an optimization problem MCSP $= (I = \wp(Q_\mathcal{E}), S, c_\mathcal{E}, \min)$ comprised by

(1) the input instances $I = \wp(Q_\mathcal{E})$ for the problem, i.e., all sets of valid queries $x \in \mathcal{E}$,
(2) the function S associating with each input set of queries $x \in \wp(Q_\mathcal{E})$ the set of feasible solutions of the MCSP problem under that input, i.e., the set of query plans that evaluate exactly all queries from x, formally, for all $x \in I$, $S(x) \subseteq P_\mathcal{E}$ and for each $p \in S(x)$ and for all $q \in x$, $p \in t_\mathcal{E}(q)$ and there exists no $q' \in Q_\mathcal{E} \backslash x$ such that $p \in t_\mathcal{E}(q')$,
(3) the objective function $c_\mathcal{E}$ assigns to each query plan (and thus to each solution) a cost used to judge the quality of several query plans evaluating the same queries, and,
(4) the optimization objective min indicating that the optimal solution has minimal cost with respect to the objective function $c_\mathcal{E}$ among all solutions for a certain input.

A solution for the MCSP is a query plan that allows the simultaneous evaluation of all queries in the input set with a cost optimal under the cost function $c_\mathcal{E}$ of the evaluation model $\mathcal{E}$. Although this definition seems very natural, the use of queries as input instances has severe consequences: Each query in the input instance can be evaluated in multiple ways, reflected by the query plans that are associated to it via the translation function $t_\mathcal{E}$ of the corresponding evaluation model $\mathcal{E}$. The number of query plans per query (representing different strategies for the evaluation of a query) can be in general very large (possibly even infinite), e.g., for relational queries the number of (logical) query plans is roughly exponential in the size of the query as any reasonable order of operators has to be considered. [120] expands a conventional

[120] Roy, P., et al. 2000. Efficient and extensible algorithms for multi query optimization. *SIGMOD (ACM Special Interest Group on Management of Data) Record 29,* 2, 249–260.

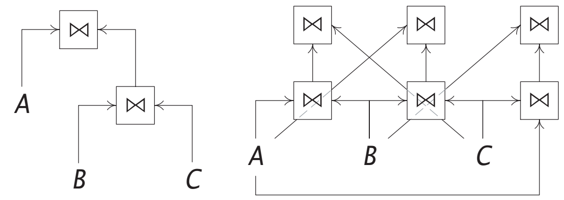


Figure 5.1: Expanded query plan (cf. [120])

query plan for a relational query to contain all possible orders of operators as shown in Figure 5.1. This expanded query plan is exponential in the size of the original query plan (and therefore in the size of the query). Based on this expanded query plan a simple but effective greedy heuristic is proposed to compute those prefixes of the query plans that are common in the query set and therefore should be materialized.

Since we are not only interested in common prefixes, but rather in any commonalities among the queries, the complexity of the problem further increases. As discussed in the following section, for the evaluation models of interest as proposed in Chapter 4, the complexity of detecting these commonalities is exponential in the size of the input, hence increasing the complexity of the problem exponentially in the number of queries, if we assume that there are roughly exponentially many query plans per query.

Therefore, we believe that it is preferable to consider not all evaluation strategies for a query, but rather to restrict oneself to a single such strategy represented by a query plan per query. Although this allows local optimization on the query plans, it is obvious that it precludes certain global optimizations: Consider e.g., the case where $A \bowtie B$ is very common among the queries, but without global knowledge an optimizer can not know whether to join A and B or B and C first in a query such as $A \bowtie B \bowtie C$ depicted in Figure 5.1. Even in this simple case, global knowledge would clearly be helpful. Nevertheless, considering the inherent complexity of the problem (and in face of the experimental results shown in Chapter 9) the restriction to a single evaluation strategy per query is considered essential in all practical cases.

In the following, we will therefore only consider this simplified problem, the **stable minimum common super-plan** SMCSP problem: The problem to find the optimal query plan for the simultaneous evaluation of multiple queries according to some evaluation strategies specified as query plans under a given evaluation model $\mathcal{E}$, is formally defined as an optimization problem SMCSP $= (I = \wp(Q_{\mathcal{E}} \times P_{\mathcal{E}}), S, c_{\mathcal{E}}, \min)$ comprised by

(1) the input instances I for the problem, i.e., $I = \wp(Q_{\mathcal{E}} \times P_{\mathcal{E}})$, i.e., an input instance is a set of queries together with their query plan (for reasons of clarity, it is assumed that the query plans are evaluating the corresponding query only, i.e., that for all instances $x \in I$ and for all queries q, q' and query plans p with $(q, p) \in x$, $p \in t_{\mathcal{E}}(q')$ implies $q' = q$).
(2) the function S associating with each input instance $x \in I$ the set of feasible solutions of the MCSP problem under that input, i.e., the set of query plans that evaluate exactly all queries from x according to the specified query plan for that query. Formally, for all instances $x = (p, q) \in I$ and all solutions $\sigma \in S(x) \subseteq P_{\mathcal{E}}$, $\sigma \in t_{\mathcal{E}}(q)$, $\sigma|_q \simeq p$, and there exists no $q' \in Q_{\mathcal{E}} \backslash x$ such that $p \in t_{\mathcal{E}}(q')$.
(3) the objective function $c_{\mathcal{E}}$ assigns to each query plan (and thus to each solution) a cost used to judge the quality of several query plans evaluating the same queries,
(4) the optimization objective min indicating that the optimal solution has minimal cost with respect to the objective function $c_{\mathcal{E}}$ among all solutions for a certain input.

It is worth noting, that the difficulty of finding a solution for the SMCSP depends noticeably on two questions:

—How hard is it, to find feasible solutions? The answer to this question depends on the properties of the underlying graphs of valid query plans. One can roughly say, the more restricted the structure of these graphs is, the easier to find feasible solutions.

—How hard is it, to find among these feasible solutions an optimal solution, i.e., what properties does the cost function adhere to? Under trivial cost function (e.g., a cost function assigning to all query plans the same cost) the optimization problem is trivial once a feasible solution has been found. Interesting cost functions on the other hand, such as a cost function that assigns cost based on the number of vertices in the underlying graph of a query plan, lead to considerable complexity for finding the optimal solution.

Based on these observations, the following section provides a classification of problem instances based on the properties of the underlying graphs allowed in query plans by the evaluation model and shows well-studied problems mostly from graph theory that provide insight in the different complexities of the problem instances.

5.3 Related Problems

Finding a maximum common substructure of a set of structures (be it a graph, tree, string, etc.) and the dual problem of finding a minimum common superstructure, i.e., a structure entailing all the input structures as substructures, has been investigated for some time now: Starting from the famous problems of graph and subgraph isomorphism (e.g., [134; 93]), the problem of finding the maximum common subgraph (mcs) of two graphs has received considerable attention and is proven not only to be NP-hard but also to be hard to approximate. [75] shows several variants of this problem together with their approximability properties. In particular, the general maximum common subgraph problem is shown to be as hard to approximate as the maximum clique problem, i.e., APX-hard. The best known approximation algorithm for mcs is $O(n/\log n^2)$-approximate. Restricting the feasible solutions to connected subgraphs, the problem becomes even harder to approximate [142]: In [75] it is shown that the maximum *connected* common subgraph (mccs) is NPO PB-complete, i.e., cannot be approximated within n^ϵ for any $\epsilon > 0$ (unless P = NP). [22] introduces the notion of the minimum common super-graph (MCS) of two graphs, i.e., a graph that contains both graphs as subgraphs and is minimal among such graphs. It is shown that the general MCS problem can be reduced to the mcs problem and is therefore at least as difficult to approximate as the mcs problem. Being already NP-hard for two graphs, it is obvious that finding a minimum common super-graph is also NP-hard for a set of graphs.

Since finding a minimum common subgraph has proven to be intractable and even hard to approximate, more restricted structures, such as trees or graphs, have been investigated: The problem of finding a shortest common super-string of a set of strings is known to be NP-complete [90] and APX-complete [16], but several approximation algorithms have been proposed [132; 133; 5], the best of which achieves a 2.5 performance guarantee. While the smallest supertree problem for two trees can be computed in polynomial time [139; 59], the problem for more than two trees is NP- and APX-complete [7].

Table 5.1 summarizes the related problems together with their complexity if the size of an input instance is 2 or unbounded respectively. Note, that even for rather simple structures, the problem of finding a common substructure becomes NP-hard when the input size is not bounded.

In the following, it will be shown that the mccs problem can be reduced to a certain instance of the SMCSP with a specific cost function. Therefore, the SMCSP is NP-hard and NPO PB-complete as well, if the evaluation model does not restrict the structure of the query plans. The same results hold also if only acyclic query

[134] Ullmann, J. R. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM 23,* 1, 31–42.

[93] McGregor, J. J. 1982. Backtrack search algorithms and the maximal common subgraph problem. *Software-Practice and Experience 12*, 23–34.

[75] Kann, V. 1992. On the approximability of the maximum common subgraph problem. In *Proc. 9th Symp. Theoretical Aspects of Computer Science.* Number 577 in Lecture Notes in Computer Science. Springer Verlag, 377–388.

[142] Yannakakis, M. 1979. The effect of a connectivity requirement on the complexity of maximum subgraph problems. *Journal of the ACM 26,* 4, 618–630.

[22] Bunke, H., et al. 2000. On the minimum common supergraph of two graphs. *Springer Computing 65,* 1, 13–25.

[90] Maier, D. and Storer, J. A. 1977. A note on the complexity of the superstring problem. Tech. Rep. 233, Princeton University. Oct.

[16] Blum, A., et al. 1994. Linear approximation of shortest superstrings. *Journal of the ACM 41*, 630–647.

[132] Turner, J. S. 1989. Approximation algorithms for the shortest common superstring problem. *Information and Computation 83,* 1 (Oct.), 1–20.

[133] Ukkonen, E. 1990. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica 5*, 313–323.

[5] Armen, C. and Stein, C. 1994. A $2\frac{3}{4}$-approximation algorithm for the shortest superstring problem. Tech. Rep. PCS-TR94-214, Department of Computer Science, Dartmouth College, Hannover (NH).

[139] Yamaguchi, A., et al. 1997. An approximation algorithm for the minimum common supertree problem. *Nordic Journal of Computing 4,* 3, 303–316.

[59] Gupta, A. and Nishimura, N. 1998. Finding largest subtrees and smallest supertrees. *Algorithmica 21,* 2, 183–210.

[7] Ausiello, G., et al. 1999. *Complexity and Approximation: Combinatorial Optimization Problems and their Approximability Properties.* Springer Verlag, Berlin.

problem	$\lvert x\rvert = 2$ complexity	$\lvert x\rvert$ unbounded complexity	approximability
shortest common **super-string** [90; 132; 133; 16; 5]	linear	NP-hard	APX-hard
minimum common **super-tree** [139; 59]	polynomial	NP-hard	APX-hard
minimum common **super-graph** [22]	reducible to maximum common subgraph		
maximum common **subgraph** [75]	NP-hard	NP-hard	APX-hard
maximum common **connected subgraph** [142; 75]	NP-hard	NP-hard	NPO PB-hard

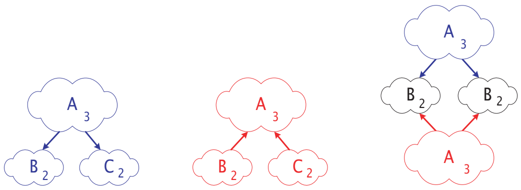
Table 5.1: Related problems (let $\lvert x\rvert$ be the size of an input instance)

plans are allowed, as in the relational and the SPEX evaluation model. It is important to stress, however, that these reductions are based on choosing the cost function in a particular way and therefore do not apply, if the cost function used is of another kind. It remains an open issue, whether interesting non-trivial classes of cost functions can be identified, that allow to approximate the SMCSP problem within a bounded performance guarantee.

As stated above, to reduce the mccs to the SMCSP problem, the evaluation model must be chosen carefully. First, it is required that arbitrary graphs are allowed as query plans. The graphs may be restricted to be acyclic (since the acyclicity does neither harm the reduction, nor the complexity and approximability of the mccs, as shown in [141]) or connected, but may not be restricted to, e.g., planar graphs, trees, or path graphs. Both evaluation models described in Chapter 4 fall into this class, since the only substantial restriction on the kind of graphs allowed is in both cases the acyclicity.

Second, the cost function of the evaluation model has to ensure that the maximum common connected subgraph of the set of input graphs is always included in an optimal solution (we understand "G_1 includes G_2" in the following as: there is a subgraph in G_1 that is isomorphic to G_2), i.e., any solution not containing it must be of penalized. In the case, where arbitrary graphs are allowed as query plans, it can be guaranteed, e.g., by a cost function $\kappa_{vertices}$ that assigns to each query plan the number of vertices it contains as cost, that an optimal solution is a solution containing the mcs. For acyclic graphs, such a cost function is not guaranteed to include the mcs in an optimal solution: Figure 5.2 shows such a case: Both query plans P_1 and P_2 include the connected graphs A, B, and C with 3, 2, and 2 vertices respectively, but in P_1 there is an edge from a vertex in A to one in B and another

[141] Yannakakis, M. 1978. The node-deletion problem for hereditary properties. Tech. Rep. 240, Computer Science Laboratory, Princeton University.



(a) P_1 (b) P_2 (c) SMCSP

Figure 5.2: Example for a SMCSP not entailing the mccs of two graphs

edge from a vertex in A to one in C, whereas in P_2 the edges run in opposite direction. Therefore, either A or B and C can be part of a feasible solution of SMCSP of P_1 and P_2, since including both A and B or A and C leads to a cyclic graph. The optimal solution under $\kappa_{vertices}$ is depicted in Figure 5.2(c): since including A precludes including both B and C it is preferable under such a cost function to include B and C but not A, although A is clearly the mccs of the two graphs (since it is connected).

To insure for acylic as well as for cyclic graphs, that the mccs is always included in the solution of the SMCSP requires a slightly different cost function: κ_{mccs} assigns to each query plan a cost based on the inverse of the number of vertices in the largest subgraph that is connected and shared by all queries that evaluate the query plan. Formally, under an evaluation model $\mathcal{E}$ let $C(G)$ be the largest connected component of the graph G, $edges(v \in V) = \{e \in E : \exists y \in V : e = (x,y) \vee e = (y,x)\}$ the incident edges of a vertex v, and $\mathcal{Q} = \bigcup_{e \in E_x} q_x(e)$ then

$$\begin{aligned}
\kappa_{\mathsf{mccs}} :& (G_x = (V_x, E_x), \tau_x, \pi_x, q_x) \in P_{\mathcal{E}} \mapsto \\
& 1/(1 + \lvert C((\{v \in V_x \mid \exists e \in edges(v) \forall q \in \mathcal{Q} : \\
& \qquad p \in t_{\mathcal{E}}(q) \implies q \in q_x(e))\}, \\
& \{e \in E_x \mid \forall q \in \mathcal{Q} : \\
& \qquad p \in t_{\mathcal{E}}(q) \implies q \in q_x(e)\}))\rvert).
\end{aligned}$$

This cost function guarantees that the mccs of a set of graphs is always included in an optimal solution of the SMCSP under κ_{mccs}: the graph constructed by sharing the mccs and adding all remaining edges and vertices from the original query plans without sharing is acyclic and therefore a query plan (if the query mappings of the edges are adapted accordingly) and has the lowest possible cost, since there can be no larger connected component shared among all query plans.

Under these cost functions guaranteeing that an optimal solution for the SMCSP always includes the mccs shared among all queries, i.e., in such a way that all edges of the mccs are assigned to all queries from the input, it is possible to extract the mccs from an optimal solution of the SMCSP in polynomial time: it is the largest connected component in the SMCSP of the subgraph of an optimal solution of the SMCSP that is obtained if only edges in the solution are retained, that are assigned to all queries from the input. Note, that the largest connected component of a graph can be obtained in time quadratic in the graph size and constructing the subgraph containing only edges assigned to all input queries is quadratic in the size of the graph and the input.

This reduction provides that the SMCSP in general (i.e., for non-restricted evaluation models) is as hard to solve as the mccs, i.e., NP-hard. Moreover, under the cost function κ_{mccs}, it should be fairly clear that this reduction is the basis for an L-reduction from the mccs to the SMCSP. Since a maximization problem is reduced to a minimization problem, either the cost function has to be inverted as part of the reduction or the corresponding maximization problem of the SMCSP under the inverted cost function has to be considered. It is easy to convince oneself, that any algorithm providing a solution of the SMCSP with a performance guarantee ϵ can be used to solve the mccs within $\epsilon\alpha$ for some rational α. Therefore, under the κ_{mccs} the SMCSP also inherits the approximability results of the mccs problem, i.e., is NPO PB-complete or not approximable by a (deterministic) polynomial algorithm within n^{ϵ} for any $\epsilon > 0$ unless P = NP.

Summing up, the comparison with related problems in the field of graph theory shows that the general SMCSP problem is NP-hard and NPO PB-complete. But theses results depend on choosing an evaluation model with very specific properties, in particular with a special purpose cost function. Under more reasonable cost function, such as $\kappa_{vertices}$ the problem can only be shown to be NP- and APX-hard, as the mcs problem can be reduced to it. Whether there are classes of cost functions, that allow a better approximation or restrictions is an open issue. Further improvements might be achieved if one restricts the kind of query graphs that are valid under a certain evaluation model, although this might severely limit the expressiveness of the corresponding queries. Therefore, neither of the two evaluation models discussed in Chapter 4 restricts the query graphs beyond the acyclicity requirement, that does not affect the complexity or approximability results presented in this chapter.

Based on these results and the definition of the problem above, the next chapter proposes several heuristics for solving the SMCSP. The quality of these approximation algorithms is evaluated experimentally in Chapter 9, since no reasonable theoretical bounds for the approximation of the SMCSP can be established.

Chapter 6

Heuristics for the Stable Minimum Common Super-Plan Problem

In this chapter, heuristics for the stable minimum common super-plan problem as defined in the previous chapter are described and compared with respect to their complexity. In particular, two sets of heuristics are investigated each based on different assumptions about the evaluation model and therefore the query plans to be optimized.

Contents

Founded on the definition of the stable minimum common super-plan SMCSP problem presented in the previous chapter, the following illustrates several heuristics for finding a solution for the SMCSP that is hopefully near the optimal one. Recall, that the SMCSP can not be approximated within n^{ϵ} for any $\epsilon > 0$, therefore no performance guarantees for the quality of the heuristics discussed here are given. In Chapter 9 the heuristics are however extensively evaluated under a realistic evaluation model, in particular for realistic cost functions as established in Chapter 8.

Before the actual heuristics can be discussed, the steps of the optimization process are sketched in the following section.

6.1 Strategies for the SMCSP

Given a set of queries, we apply the following steps to optimize them into a single query plan that allows the efficient evaluation of all the input queries simultaneously:

(1) First the queries are *translated* into a (local) query plan and optimized. As discussed in the previous chapter, a solution to the SMCSP will be stable in respect to the original query plan for a query, i.e., the order and type of operators in a query will not be changed. Although, this precludes certain optimizations on the queries based on global knowledge, it makes the problem far more feasible for practical cases. Consequentially, one might often be better of, not to perform optimizations on the queries that are based on local heuristics. For the SMCSP problem, it is often more important that the initial query plans provide retain as much as possible any similarities in the queries. Therefore, a good choice proves to be a rather canonical form of a query plan, that might not be the optimal query plan for a query based on isolated knowledge about the query, but increases the

chance that similarities in the queries are reflected in the query plan. Such a canonical form can, e.g., eliminate syntactical variants of the same semantic construct. Constructing a canonical form does not, however, preclude all local optimizations, e.g., the compaction of common prefixes of branches shown in Figure 2.14 can be applied safely, since it applies in all cases.

(2) To lower the number of query plans that have to be merged with each other, one can reduce the number of query plans or cluster the query plans in such a way that only query plans that are sufficiently similar will be merged with each other:
In particular, if it is known that the query plans are likely to be very similar, identical query plans or query plans that are subgraphs of another one should be detected a priori. Note, that detection of identical query plans is based on graph isomorphism, for which no polynomial time algorithm is known, but can be approximated efficiently. The detection of query plans that are subgraphs of another one is even NP-hard, since it is based on subgraph isomorphism. Nevertheless, this *duplicate merging* becomes even more promising, if one realizes, that many query plans might differ only in certain properties of some vertices, but not in the operators assigned to the vertices. E.g., query plans often differ only in the constant value within a selection or label operator. In such a case, if the evaluation model allows the merging of these properties, even such query plans can be merged a priori.
Clustering the query plans into sets of sufficiently similar query plans and solving the SMCSP for the entire set of query plans, by constructing a solution from solutions on the clusters without sharing among these solutions can reduce the processing time by the number of clusters obtained. This is especially promising, if the input queries are known to adhere to a clustered distribution. Clustering of graphs in general, has been received some attention in recent years, in particular in the context of biological data and non-standard databases, cf. [21].

(3) This reduced set of canonical query plans can now be merged. There are essentially three different strategies to merge a set of query plans:

Sequential strategy. All query plans are considered sequentially and merged one after another into an ever growing (multi-) query plan. For each query plan, all operators originating from one of the already merged queries are considered.

Clustered strategy. Instead of considering for all query plans all other (already merged) query plans, query plans are merged only with query plans of sufficient similarity. The extreme case is that a query plan is only merged with the query plan where the expected gain (measured by means of the objective function of the SMCSP) is the highest.

Global strategy. Instead of considering the query plans isolated from each other as in the previous cases, it might be more efficient to detect subgraphs that are frequently occurring among the query plans. Based on these frequent subgraphs, an (approximate) solution of the SMCSP can be constructed. Frequent subgraph or substructure discovery has been investigated quite extensively [82; 70; 69; 68], in particular in the context of biological data and semi-structured data [135].

In this work, we concentrate on the sequential strategy, although an algorithm based on the clustered strategy is discussed shortly in the following section. To define the strategy for merging a set of query plans sequentially in an increasing multi-query plan, two questions have to be answered: In which order will the query plans be considered and how is a query plan merged into the query plan resulting from the mergings of the previous query plans. An exact solution based on the sequential strategy would therefore require to merge the n query plans of average size l in all possible $n!$ permutations, where for each permutation the merging requires roughly $O(n\times(n\times l)^l)$ steps.

[21] Bunke, H. 2000. Recent developments in graph matching. In *Proc. of the International Conference on Pattern Recognition (ICPR).* Vol. 2.

[82] Kuramochi, M. and Karypis, G. 2002. An efficient algorithm for discovering frequent subgraphs. Tech. Rep. 02-026, Computer Science Departement, University of Minnesota.

[70] Inokuchi, A., et al. 2002. General framework for mining frequent patterns from structures. In *Proc. of the International Workshop on Active Mining (AM 2002).* 23–30.

[69] Inokuchi, A., et al. 2003. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning 50,* 3, 321–354.

[68] Inokuchi, A., et al. 2000. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the European Conference on Principles and Practice of Knowledge Discovery and Data Mining (PKDD2000).* 13–23.

[135] Wang, K. and Liu, H. 1999. Discovering structural association of semistructured data. *IEEE Transactions on Knowledge and Data Engineering (TKDE).*

Clearly, such an exact solution is feasible only for very small numbers of query plans and very small query plans. Therefore, in the following several approximation algorithms for both problems are proposed and discussed based on their complexity and properties. In Chapter 9 an experimental evaluation of these approximation algorithms based on the SPEX evaluation engine discussed in Chapter 7 shows that in practical cases at least some of the heuristics can give good solutions in reasonable time, in particular if compared to a tree prefix merger as used in previous work.

The following discussion is grouped into algorithms that are used to determine the order in which the query plans from the input are considered and algorithms that merge a query plan into another query plan (that might be already a query plan for evaluating multiple queries). The first group of algorithms is referred to as *set mergers*, the second as *pair mergers* corresponding to the number of query plans they operate on. Since each set merger uses an (arbitrary) pair merger to perform the actual merging of the query plans, once the order is determined, the pair mergers are presented first.

6.2 Pair Mergers: Algorithms for Merging Pairs of Query Plans

As stated above, a **pair merger** is an algorithm $\mathcal{A}$ that takes as input two query plans N and M evaluating disjunct sets of queries Q_N and Q_M and constructs a new query plan P that is a feasible solution for the SMCSP with input $\{ (q, p) \mid q \in Q_N \cup Q_M \wedge (p \simeq N|_q \vee p \simeq M|_q) \}$. If both Q_N and Q_M are singleton sets, the algorithm constructs a solution of the SMCSP with input N and M. In the following, we assume without loss of generality that $|N| \leq |M|$.

In the following, two classes of pair mergers are presented differing in what operations must be supported by the evaluation model the input plans are part of:

—Pair merger that construct a solution by merging vertices from the input plans step-by-step are called **incremental** pair mergers. There are two assumptions underlying these heuristics: First, during the sequential processing of the vertices, it must at any time be possible, to decide, whether a partial solution can be completed to a full solution by considering the remaining vertices only. In other words, it must be decidable in polynomial time in the size of the input, given a partial solution, whether there is some valid query plan that is a full solution and that this partial solution can be completed to. Furthermore, the cost function used as objective function of the SMCSP must provide means to compute the cost of a partial solution.

—The other class of pair mergers proposed here is based on the idea to start from a random solution searching in the "neighborhood" of that solution for a better one. The "neighborhood" of a solution are all solutions that can be constructed from the original one by applying some transformation function. Following common notation [99], these algorithms are called *local search* pair mergers. The main advantage of this strategy is that at any time only full solutions are considered overcoming the restrictions discussed for incremental pair merger. However, there must be some meaningful way to transform a solution into another one. If either the evaluation model used does not adhere to the requirements of the incremental pair mergers or constructing a random solution is clearly easier than constructing a good solution by means of an incremental pair merger, a local search pair merger is preferable to an incremental one.

In Chapter 9 one other pair merger is used for reasons of comparisons: the trivial or *plain pair merger* that constructs the solution P by simply copying all vertices and edges together with their respective mappings for τ, π, and q from P_1 and P_2 into P. The resulting query plan obviously evaluates all queries from Q_1 and Q_2, but there is no sharing of operators between queries from the two sets.

Table 6.1 gives an overview over all pair mergers with a short characterization and their complexity.

6.2.1 Incremental Pair Mergers

As discussed above, an incremental pair merger merges vertices from the input plans N and M incrementally into a solution P. The general idea of these algorithms is to consider for each vertex v from N "interesting" vertices w in M that can be merged with v.

The variants proposed in the following mostly differ with respect to the following two questions: in

[99] Michalewicz, Z. and Fogel, D. B. 2000. *How to Solve It: Modern Heuristics*, 1st ed. Springer Verlag.

pair merger	principle	complexity
exhaustive	computes all solutions	$O(\chi(N,M)^{\lvert N\rvert} \times (T(\texttt{cost}) + T(\texttt{merge}) + T(\texttt{validCandidates})))$
tree prefix	tree-shaped plans, prefixes only	$O(N \times (\max(\deg(M), \lvert S\rvert) + T(\texttt{cost}) + T(\texttt{merge})))$
graph prefix	acyclic plans, prefixes only	$O(\max(\deg(M), \lvert S\rvert)^{\lvert N\rvert} \times (T(\texttt{cost}) + T(\texttt{merge})))$
initial greedy	best gain heuristic, considers vertices from N and candidates separately	$O(\mathsf{N} \times (\chi(N,M) \times (T(\texttt{cost}) + T(\texttt{merge})) + T(\texttt{validCandidates})))$
progressive greedy	best gain heuristic, considers pairs of vertices from N and candidates	$O(\mathsf{N}^2 \times (\chi(N,M) \times (T(\texttt{cost}) + T(\texttt{merge})) + T(\texttt{validCandidates})))$
random	selects random vertex from N and random candidate	$O(\mathsf{N} \times (T(\texttt{cost}) + T(\texttt{merge}) + T(\texttt{validCandidates})))$
deterministic hill-climber	local search, deterministic neighbor selection	$O(\tau \times \iota \times (T_{random} + \nu(N,M) \times (T_{trans} + T(\texttt{cost}))))$
stochastic hill-climber	local search, probabilistic neighbor selection	$O(\tau \times \iota \times (T_{random} + \nu(N,M) \times (T_{trans} + T(\texttt{cost}))))$
simulated annealer	local search, probabilistic neighbor selection with decreasing probability	$O(\tau \times r_{max} \times (T_{random} + \nu(N,M) \times (T_{trans} + T(\texttt{cost}))))$

Table 6.1: Comparison of pair mergers (for the notations, refer to the specific sections)

what order are the vertices of N considered and what are "interesting" vertices for merging with a vertex from N and how are these vertices ordered? Aside of these differences, all incremental pair merger have a very similar structure shown in Figure 6.1: The `pair-smcsp` function takes as input two query plans N and M and returns a query plan P that is a feasible solution for the SMCSP with input $\{ (q,p) \mid q \in Q_N \cup Q_M \wedge (p \simeq N|_q \vee p \simeq M|_q) \}$ as stated above. The actual search is performed by `findCommonPlan` which takes as input two query plans N and M and a list of vertices $R \subset V_N$. It returns a query plan M' obtained by merging all vertices from R with "interesting" vertices from M according to the specific heuristic.

In the following, we denote a sequence similar to a set but with angle brackets and enhanced by the order in which the elements are ordered, e.g., $\langle\, v \mid v \in V_N \,\rangle_{\leq}$ is the sequence of all vertices in N topologically sorted by some partial order $\leq$. If no order is given for the sequence, the elements are arbitrarily ordered. $\langle\rangle$ denotes the empty sequence. To access the first element (respectively the remaining elements) of a sequence S, $\texttt{head}(S)$ (respectively $\texttt{tail}(S)$) is used.

`cost` denotes some implementation of the cost function $c_{\mathcal{E}}$ part of the evaluation model $\mathcal{E}$ used. It is assumed that `cost` can compute not only the cost of full solutions but also the cost of partial solutions, as discussed above. Chapter 8 details several cost functions with different complexities.

Finally, the `merge` function describes how a vertex $v \in N$ is merged with a vertex $w \in M \cup \{v\}$. For ease of presentation, the case where v is not merged with any vertex in M but rather added as a new vertex is represented by merging v with v. Figure 6.2 shows an outline of the actual merging algorithm: The query plan M' containing the merging from v to w is constructed by adding all edges between v and a vertex $z \in N$ that is already merged as edges between w and $\vec{z}$ updating also the queries that edge is part of. Only if the $v = w$ a new vertex with the same operator and properties as v is inserted into the resulting query plan at the beginning of the construction. In that case, also the edges are added to the new vertex. Given constant access to $\vec{z}$ for any vertex z and an efficient representation of the graph structure of a query plan with linear-time iteration over the incident edges of a vertex and constant or amortized constant test

```
funct pair-smcsp(N, M) ≡
  ⌈findCommonPlan(⟨ v | v ∈ V_N ⟩, N, M)⌋.

funct findCommonPlan(R, N, M) ≡
  ⌈if (R = ⟨⟩)
     then M
     else M_best ← N ∪ M
          v ← head(R)
          while (not all "interesting" vertices in M for merging with v have been considered) do
            w ← next "interesting" vertex in M for merging with v
            M′ ← merge(v, w, N, M)
            M′ ← findCommonPlan(tail(R), N, M′)
            if (cost(M′) < cost(M_best))
              then M_best ← M′ fi
          od
          M_best
   fi⌋.
```

Figure 6.1: Skeleton of an incremental pair merger

whether two given vertices are adjacent as described in Chapter 10, this function can be implemented in $O(\deg(N) \leq |N|)$ where, for a query plan P, $\deg(P)$ is the degree of underlying graph.

For convenience, we abbreviate the test whether two vertices v and w can be merged by a boolean function `mergeable` that returns **true** if the same operators are assigned to v and w and their properties, if they exist, can be merged into a new one:

```
funct mergeable(v, w) ≡
  ⌈τ(v) = τ(w) ∧ π(v) = π(w) ∧
       (∃r ∈ R_𝓔 : r = π(v)  ⟹
       ∃r′ ∈ R_𝓔 : r′ = μ_𝓔(π(v), π(w)))⌋.
```

Assuming merging two properties is a constant operation, this test can be performed in constant time.

Based on the skeleton from Figure 6.1, the following sections detail the four incremental pair mergers that are each using different answers to the questions discussed above: (1) the exhaustive pair merger, that considers all possible mergings from N into M, the prefix pair merger for which only common prefixes are interesting for merging, (2) the greedy pair merger who tries to approximate a good solution by merging always those vertices next that deliver the best gain if merged, and, for comparison, (4) the random pair merger choosing mergings arbitrarily.

The differences between the algorithms are described by means of two functions, `sortVertices`(N, M) that returns a sequence of vertices from N ordered according to the specific heuristic of the specific algorithm and `interestingCandidates`(v, N, M) that returns a sequence of vertices from M that are "interesting" for merging with $v \in V_N$. Note, that any vertex that is "interesting" for merging with v must be allowed for merging with v.

As detailed above, it is required that there is some means specific to the evaluation model, to determine whether the incomplete query plan resulting from a merging of v with some vertex in M can still be completed to a full solution. This is facilitated here by means of a function `validCandidates`(v, N, M) that returns all vertices in M that can be merged with v such that the result can be completed to a full solution. $\chi(N, M) = \max\{\, n \in \mathbb{N} \mid \exists v \in N : n = |\texttt{validCandidates}(v, N, M)| \,\} \leq |M|$ denotes the maximum number of vertices in M valid for merging with a vertex in N and is bounded by $|M|$.

To illustrate this function, recall the query plans for the XML streams proposed in Section 2.3.2 and 4.3 together with their evaluation model $\mathcal{X}$. Only acyclic graphs can constitute a valid query plan for $\mathcal{X}$. For a query plan P whose underlying graph is acyclic, let $<_P$ be a partial order on the vertices of P, such that, for all vertices $v, w \in V_P$, $v <_P w \equiv \exists n \in \mathbb{N}, v_1, \ldots, v_n : v = v_1 \wedge w = v_n \wedge (v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n) \in$

```
funct merge(v, w, N = ((V_N, E_N), τ_N, π_N, q_N), M) ≡
 ⌈M′ = ((V_M′, E_M′), τ_M′, π_M′, q_M′) ← M
  if v = w
    then let v′ be a new vertex
         V_M′ ← V_M′ ∪ {v′}
         τ_M′ ← τ_M′ ∪ {v′ ↦ τ_N(v)}
         if (∃r ∈ R_𝔛 : r = π_N(v))
           then π_M′ ← π_M′ ∪ {v′ ↦ r} fi
         v ← v′
  fi

  for each z ∈ { y ∈ V_N | (v, y) ∈ E_N } do
      if (∃y ∈ V_M′ : z⃗ = y)
        then if (w, z⃗) ∈ E_M′
               then q_M′((w, z⃗)) ← q_M′((w, z⃗))∪
                                   q_N(v, z)
               else E_M′ ← E_M′ ∪ {(w, z⃗)}
                    q_M′((w, z⃗)) ← q_N(v, z)
             fi
      fi
  od

  for each z ∈ { y ∈ V_N | (y, v) ∈ E_N } do
      if (∃y ∈ V_M′ : z⃗ = y)
        then if (z⃗, w) ∈ E_M′
               then q_M′((z⃗, w)) ← q_M′((z⃗, w))∪
                                   q_N(z, v)
               else E_M′ ← E_M′ ∪ {(z⃗, w)}
                    q_M′((z⃗, w)) ← q_N(z, v)
             fi
      fi
  od
  M′⌋.
```

Figure 6.2: Algorithm for merging two vertices

```
funct validCandidates_𝒳(v, N, M) ≡
 ⌈C ← { w ∈ M | mergeable(v, w) }
  for each w ∈ { x ∈ N | ∃y ∈ M : y = x⃗ } do
      if w <_P v
        then C ← C \ { y ∈ M | y ≤_M w⃗ } fi
      if v <_P w
        then C ← C \ { y ∈ M | w⃗ ≤_M y } fi
  od
  C ← C ∪ {v}⌋.
```

Figure 6.3: Finding vertices in M valid for merging with v under the evaluation model $\mathcal{X}$

E_P, i.e., $v <_P w$, if there is a path from v to w.

For this evaluation model `validCandidates` is shown in Figure 6.3: The valid vertices w for merging with v are initialized to all vertices that have the same type as v and the property assigned to w can be merged with the one assigned to v, if there are properties assigned to them at all. Among these vertices only those vertices w are retained such that a merging of v with w does not create a cycle in the graph. This can be ensured, if all ancestors of vertices $\vec{w} \in M$ that are merged to an ancestor $w \in N$ of v and all descendants of vertices $\vec{z} \in M$ that are merged to an descendant $z \in N$ of v are removed from the initial set as illustrated in Figure 6.4, where black indicates the initial setup, blue possible mergings for v, and red the consequential cyclic additions. The lines ending in dots indicate mergings and the gray triangles the descendants resp. ancestors of z and w. Neither $\vec{v}_1$ nor $\vec{v}_2$ in Figure 6.4 are valid for merging with v, since either of the two mergings results eventually in a cycle. It is important to notice, that conventional cycle detection algorithms, such as [125], can not be applied since the cycle might be created only later in the processing. In the example, only once the vertex $x \in N$ is also merged to a vertex in M a cycle occurs regardless of how x is added to M (even if it is added without sharing). This cycle avoidance algorithm can be implemented linear in the number of vertices in M (by tagging all vertices in M that have been considered once as descendant or ancestor of some merged vertex) and is therefore as efficient as the best dynamic cycle detection algorithms [125], i.e., cycle detection algorithms in face of changing graphs as in this case. Nevertheless, it is rather costly, since it is linear in M and M increases with the number of query plans merged as discussed in Section 6.3.

It should be emphasized again, that this method of determining a valid pair of vertices for merging is specific to the case, where the only restriction on the structure of a query plan is, that it is acyclic. For evaluation models, that use other means to determine what a valid query plan is, different methods for determining these pairs are required, if there are any.

[125] Shmueli, O. 1983. Dynamic cycle detection. *Information Processing Letters 17*, 4, 185–188.
[125] Shmueli, O. 1983. Dynamic cycle detection. *Information Processing Letters 17*, 4, 185–188.

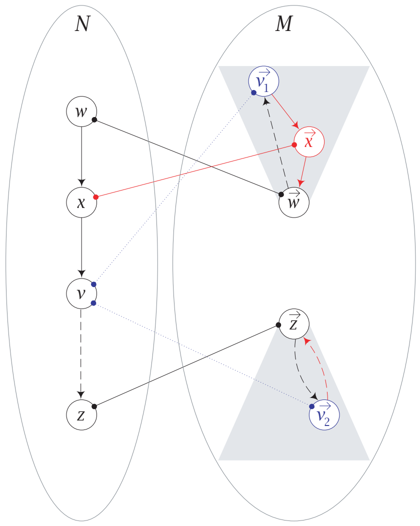


Figure 6.4: Avoiding cycles

Exhaustive Pair Merger

The only pair merger that computes the optimal solution, is the exhaustive pair merger. Following chapter 5, the SMCSP is NP-hard, therefore any deterministic algorithm returning an optimal solution for the SMCSP is exponential. As the name indicates, this pair merger exhaustively computes all possible mergings from N into M and selects the best.

Since all mergings are computed, the order in which the vertices are considered is not relevant. Therefore, the exhaustive variant of $\texttt{sortVertices}_{\text{exh}}(N, M) \equiv \lceil\langle\, w \mid w \in V_N \,\rangle\rfloor$ returns the vertices in no particular order.

For the same reason, the implementation of $\texttt{interestingCandidates}_{\text{exh}}(v, N, M) \equiv \lceil\langle\, w \mid w \in \texttt{validCandidates}(v, N, M)\,\rangle\rfloor$ is similarly trivial returning an arbitrary-ordered sequence of those vertices that can be merged with v.

With these implementations for $\texttt{sortVertices}_{\text{exh}}$ and $\texttt{interestingCandidates}_{\text{exh}}$, the skeleton algorithm from Figure 6.1 computes an optimal solution in time

$$O(\chi(N, M)^{|N|} \times T_{base})$$

where $T_{base} = T(\texttt{validCandidates}) + T(\texttt{cost}) + T(\texttt{merge})$ and, for an algorithm $\mathcal{A}$, $T(\mathcal{A})$ is the time complexity of $\mathcal{A}$. In practical cases, one can observe that the number of valid merge candidates for most vertices from N is clearly smaller than $|M|$, since only vertices assigned to the same operator and a property that can be merged with the property of the vertex from N are valid. Assuming an equal distribution of the types and no properties assigned to the vertices, the complexity shrinks to roughly

$$O\left(\frac{|M|}{|O_{\mathcal{E}}|}^{|N|} \times T_{base}\right),$$

where $O_{\mathcal{E}}$ is the set of operators in the evaluation model $\mathcal{E}$. If only acyclic query plans are considered valid (as in the evaluation models shown in Chapter 4), this number shrinks even more, as some vertices are not any more valid candidates for merging, since such mergings would result in cycles.

One of the more important optimizations on exhaustive algorithms is the branch-and-bound technique, where branches of the search tree for which a partial solution has already higher cost than the best known full solution are skipped. But branch-and-bound optimization can only be applied if the cost function used is *monotonic* in the sense, that adding additional mergings to a partial solution can not result in lowering the cost, but either leaves the cost unchanged or increases it. If the cost function admits to this criteria, a simple branch-and-bound test can be inserted before line 13 of `findCommonPlan`: **<u>if</u>** $\texttt{cost}(M') > \texttt{cost}(M_{best})$ **<u>then</u>** **<u>continue</u>** **<u>fi</u>**. Such a branch-and-bound optimization can be further improved, if one considers both the vertices from N and the candidates for merging in such an order that solutions that are likely to have a low cost are generated early. The same heuristics for ordering the vertices in a promising way can be used as for the greedy pair merger discussed below.

Prefix Pair Merger

In contrast to the exhaustive pair merger discussed previously, the prefix pair merger is very efficient but produces in most cases equally poor solutions. As the name suggests, the prefix merger merges only common prefixes of the input plans. There are two variants of this merger differing in the assumptions they make about the structure of the query plans:

The first variant, similar to previous optimization techniques for multiple XML queries [4; 28], is referred

[4] Altinel, M. and Franklin, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the International Conference on Very Large Databases (VLDB)*.

[28] Chan, C.-Y., et al. 2002a. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal (Special Issue on XML Data Management)*.

to as *tree prefix pair merger* and assumes that a query plan is tree-shaped and at any branch it is assumed that for each child vertex in the first query plan N there is either no or a unique child vertex in M that can be merged with it. Whether there is such a child vertex and which it is, can be determined in time at most linear in the number of children, in particular without descending into the branches beyond the level of the children. Furthermore, it is assumed that a query plan constructed by using these kind of mergings only is always valid.

Figure 6.5 shows how these assumptions translate into an algorithm for selecting the vertex from M to be merged with a vertex $v \in N$. $\texttt{interestingCandidates}_{\text{tree prefix}}$ returns always a singleton sequence that is the only vertex that v will be merged to. This reflects the determinism in the selection of the vertex w, that v is to be merged into. w is determined by testing whether the first parent p (it is assumed there is only one parent, since the query plans are assumed to be tree shaped, so $\mathcal{P}$ should always either be empty or a singleton set) is already merged into a $\vec{p}$. If that is the case, the prefixes of N and M ending in p respectively $\vec{p}$ are merged and v can be merged with a child of $\vec{p}$ that is `mergeable` to v if there exists such a child. If either the parent is not merged or there is no such child, the singleton sequence $\langle v \rangle$ is returned, indicating that v is to be merged into a new vertex in M. There is one special case, that needs to be treated: a source of the graph (i.e., the root of the tree). A source vertex of N is merged with the first source vertex of M found that is `mergeable` with it. Again, if the query plans are tree shaped, there is only one source vertex in M that is `mergeable` with a given source vertex in N.

This algorithm is only correct, if the vertices of N are considered in a topological order, guaranteeing that a parent is always merged before its children. So $\texttt{sortVertices}_{\text{tree prefix}}(N, M) \equiv \lceil \langle\, v \mid v \in V_N \,\rangle_{<_N} \rfloor$ orders the vertices in the topological order $<_N$ as described above.

Observe, that the tree prefix pair merger as described above can not be applied to cyclic graphs, since it relies on the existence of a topological order on the vertices. But on acyclic query plans the algorithm actually succeeds, but generates solutions that are in general not optimal, even if one restricts sharing among query plans to sharing of prefixes. This is due to the fact that this algorithm assumes that for each vertex $v \in N$, child of a vertex $p \in N$ that is merged to $\vec{p} \in M$, there is only a single children of $\vec{p}$ that is mergeable with v (the same applies for the case of the source vertex).

Better solutions can be produced, if all such children are considered. This is the strategy implemented by the graph prefix pair merger shown in Figure 6.6. Once again the vertices from N are assumed to be processed in topological order (precluding cyclic graphs), but now all vertices c that are children of a vertex $\vec{p}$ that is merged to a parent p of v are considered candidates for merging with v, if $\texttt{mergeable}(c, v)$. The special case of the source vertices is handled analogously. Clearly, this algorithm leads to exponential complexity. On the same lines as the exhaustive pair merger, the graph prefix pair merger can be improved by adding a branch-and-bound test and considering the candidates for a vertex from N in an order such that promising solutions are generated first, if the underlying cost functions is monotonic with respect to adding new mergings.

More precisely, let $S = \{\, v \in M \mid \neg\exists p \in M : (p, v) \in E_M \,\}$ be the set of sources of M, then the complexity of the graph prefix pair merger is

$$O(\max(\deg(M), |S|)^{|N|} \times (T(\texttt{cost}) + T(\texttt{merge})))$$

in contrast to the quadratic complexity of the tree prefix pair merger

$$O(N \times (\max(\deg(M), |S|) + T(\texttt{cost}) + T(\texttt{merge})))$$

Note, that in contrast to all other pair mergers, neither of the prefix mergers uses `validCandidates`, since they assume that the prefix mergings performed always generate valid query plans, as stated above. If this restriction does not hold, the selected candidate has furthermore to be tested for validity using `validCandidates` increasing the complexity accordingly.

As discussed, the main restriction of the tree prefix pair merger is the poor quality of the solutions it generates once the query plans do not adhere closely to the requirements posed by the algorithm. The graph prefix pair merger on the other hand provides only a slight improvement over the exhaustive pair merger with respect to complexity but constructs far inferior solutions. But, if the evaluation model used admits to the requirements posed by the tree prefix pair merger, in particular if all query plans that use only

```
funct interestingCandidates_tree prefix(ν, N, M) ≡
 ⌈C ← ⟨ν⟩
  𝒫 = ⟨ w ∈ V_N | (w, ν) ∈ E_N ⟩
  if 𝒫 ≠ ∅
    then p ← head(𝒫)
         if (∃w ∈ V_M : w = p⃗)
           then for each c ∈ { w ∈ V_M | (p⃗, w) } do
                    if mergeable(c, ν) then C ← ⟨c⟩
                                             break fi
                od
         fi
     else for each c ∈ { w ∈ V_M | ¬∃p ∈ V_M : (p, w) ∈ E_M } do
              if mergeable(c, ν) then C ← ⟨c⟩
                                       break fi
         od
  fi
  C⌋.
```

Figure 6.5: Finding `interestingCandidates` in the tree prefix pair merger

```
funct interestingCandidates_graph prefix(ν, N, M) ≡
 ⌈C ← {ν}
  𝒫 = ⟨ w ∈ V_N | (w, ν) ∈ E_N ⟩
  if 𝒫 ≠ ∅
    then for each p ∈ 𝒫 do
              if (∃w ∈ V_M : w = p⃗)
                then for each c ∈ { w ∈ V_M | (p⃗, w) } do
                         if mergeable(c, ν) then C ← C ∪ {c} fi
                     od
              fi
         od
     else for each c ∈ { w ∈ V_M | ¬∃p ∈ V_M : (p, w) ∈ E_M } do
              if mergeable(c, ν) then C ← C ∪ {c} fi
         od
  fi
  ⟨ ν | ν ∈ C ⟩⌋.
```

Figure 6.6: Finding `interestingCandidates` in the graph prefix pair merger

prefix merging are valid, the tree prefix merger can often produce reasonable solution in an extremely short time compared to other heuristics.

Greedy Pair Merger

Implementing a simple greedy heuristic, the greedy pair merger proves in practical cases to provide a reasonable compromise between the quality of the constructed solution and the time for the construction, cf. Chapter 9.

The greedy pair merger chooses for a vertex $v \in N$ the vertex w from M that is to be merged with v based on greedy heuristic: Only that $w \in M$, where the cost of M increases the least, if v and w are merged, is actually merged with v.

In Figure 6.7, an implementation of the `interestingCandidates` function for the greedy pair merger is shown: It returns the singleton sequence containing a valid candidate for merging with v that has minimal cost among all such candidates.

But in which order should the vertices of N be considered in such a case? Differing in their answer to this question, we propose two variants of the greedy pair merger: The first variant, called *initial* greedy pair merger, computes an order among the vertices of N initially. Therefore, it is assumed that there is some way to measure the cost of a vertex $v \in N$ and that cost is used as an indication of the expected gain when v is merged with some vertex in M. In other words, this variant requires that there is some function $\texttt{cost}(v, N, M)$, such that $\texttt{cost}(v, N, M) \approx \max\{\, n \in \mathbb{N}_0 \mid \exists w \in \texttt{validCandidates}(v, N, M) : |\texttt{cost}(M) - \texttt{cost}(\texttt{merge}(v, w, N, M))| = n \,\}$. $\texttt{cost}(v, N, M)$ is not required to compute the exact maximum, just to give an estimation of the best expected gain for merging v with any vertex from M. The initial greedy pair merger orders by the value returned by $\texttt{cost}(v, N, M)$ in descending order, i.e., the vertex with the highest value first.

This variant can however not ensure that always the next best pair of vertices is considered, since the order of the vertices is based on the valid candidates in the initial graph M only. So if due to some prior mergings, certain vertices are no longer candidates this is not reflected in the order: E.g, if there are two vertices $v', v'' \in M$ that can be merged in the initial graph M with a vertex $v \in N$, such that the gain when merging with v' is γ and the gain when merging with v'' is $\delta < \gamma$, the position of v is determined by γ. Assume that there are vertices x, y, such that $\texttt{cost}(x, N, M) > \texttt{cost}(v, N, M) = \gamma > \texttt{cost}(y, N, M) > \delta$. Then first x, then v and finally y is merged by the initial greedy pair merger. If the merging of x now invalidates v' as a candidate for merging with v (e.g., as the resulting query plan could not anymore be complete to a full solution), v is still considered before y and merged with v'' since that is the best merging candidate that is still valid. Merging v and v'' in turn might invalidate the best merging of y which would have provided a gain higher than δ.

Therefore, a more elaborate version of the greedy pair merger, referred to as *progressive* greedy pair merger, is provided in Figure 6.8: It does not compute the order how the vertices in N are to be merged initially, but rather after each merging the next best merging is determined by considering all remaining vertices N with their respective candidates from M. Note, that this algorithm as well as all the other pair mergers with the exception of the exhaustive pair merger can be as easily formulated using iteration instead of recursion but are given in a recursive variant for ease of presentation.

The disadvantage of the progressive greedy pair merger is naturally the increased complexity. Whereas the initial greedy pair merger has the quadratic complexity

$$O(\mathsf{N} \times (\chi(N, M) \times (T(\texttt{cost}) + T(\texttt{merge})) + T(\texttt{validCandidates}))),$$

where χ and T are used as above, for the progressive variant the complexity increases by a factor $|N|$ to

$$O(\mathsf{N}^2 \times (\chi(N, M) \times (T(\texttt{cost}) + T(\texttt{merge})) + T(\texttt{validCandidates}))),$$

since in all $|N|$ steps all remaining vertices from N are considered.

The initial variant has a similar complexity as the tree prefix pair merger except that in each step the valid candidates have to be computed. Experimental evaluation (cf. Chapter 9) points to the fact that with reasonable cost functions under the evaluation model from Section 4.3, the initial variant provides the better trade-off between construction time and result of the constructed solution.

```
funct interestingCandidates_greedy(v, N, M) ≡
⌈var best candidate best, cost of best candidate cost_best ← ∞
 for each w ∈ validCandidates(v, N, M) do
     M′ ← merge(v, w, N, M)
     if cost(M′) < cost_best then
                                     cost_best ← cost(M′)
                                     best ← w
     fi
 od
 ⟨best⟩⌋.
```

Figure 6.7: Finding `interestingCandidates` in the greedy pair merger

```
funct findCommonPlan(R, N, M) ≡
⌈if (R = ⟨⟩)
   then M
   else M_best ← N ∪ M
        n_best ← head(R)
        for each v ∈ R do
            for each w ∈ validCandidates(v, N, M) do
                M′ ← merge(v, w, P, M)
                if (cost(M′) < cost(M_best))
                  then M_best ← M′
                       n_best ← v
                fi
            od
        od
        M_best ← findCommonPlan(R\{n_best}, N, M)
        M_best
 fi⌋.
```

Figure 6.8: Progressive greedy pair merger

Random Pair Merger

The final pair merger proposed in this section is for comparison only: the order of the vertices as well as the selection of a candidate among the valid candidates for a vertex is randomized. More precisely, `orderVertices`(N) returns random sequence of the vertices in N and `interestingCandidate`(v, N, M) returns a singleton sequence containing a random vertex from `validCandidates`(v, N, M).

The complexity of the random merger differs from the complexity of the initial greedy pair merger in the fact that the random choice is assumed to be constant, therefore the maximum number of valid candidates $\chi(N, M)$ is not influencing the complexity:

$$O(\mathsf{N} \times (T(\mathtt{cost}) + T(\mathtt{merge}) + T(\mathtt{validCandidates})))$$

Interestingly, for an evaluation model such as $\mathcal{X}$, presented in Section 4.3, the random pair merger has the same worst-case complexity as the tree prefix merger, as both $T(\mathtt{validCandidates})$ and $\deg(M)$ are bounded by M, but in most cases the tree prefix merger actually performs *better*, as $\deg(M)$ is usually clearly smaller than M whereas $T(\mathtt{validCandidates})$ is often very near to M for the evaluation model $\mathcal{X}$.

```
funct pair-smcsp(N, M) ≡
 ⌈findCommonPlan(N, M)⌋.

funct findCommonPlan(N, M) ≡
 ⌈S_best ← N ∪ M
  r ← 0
  while r < MAX-TRIES do
    S ← random solution
    t ← 0
    while t < MAX-ITERATIONS-PER-TRY do
      S′ ← interestingNeighbour(S, M, N)
      if S′ = S then break
                 else S ← S′ fi
      t ← t + 1
    od
    if cost(S) < cost(S_best)
      then S_best ← S fi
    r ← r + 1
  od
  S_best⌋.
```

Figure 6.9: Skeleton of a local search pair merger

6.2.2 Local Search Pair Mergers

As mentioned above, the incremental pair mergers (with the exception of the exhaustive merger) require a way to determine the cost and the validity of a *partial* solution, i.e., if a partial solution can be completed to a full solution. In contrast, the pair mergers discussed in this section operate on full solutions only, but require a means to find solutions that are similar to a given solution.

Basically, these pair mergers are local search algorithms [99], i.e., algorithms that start from several randomly generated solutions and improve these random solutions by considering other solutions that are sufficiently similar. Two similar solutions are also referred to as "neighbors" and the set of similar solutions for a solution as "neighborhood" of that solution.

Figure 6.9 sketches the skeleton of the local search pair merger: A common query plan for N and M is computed by generating `MAX-TRIES` random solutions. Each such solution S is improved by selecting an "interesting" neighbor S' of S and continuing the search from S'. This process is repeated until no further "interesting" neighbor can be found (i.e., $S' = S$), but at most `MAX-ITERATIONS-PER-TRY` times for a single randomly generated solution.

The difference in the local search algorithms presented in the following, is the definition of an "interesting" neighbor: the deterministic hill-climber selects among all neighbors of a solution the one with the lowest cost, the stochastic hill-climber selects the first neighbor that is acceptable with respect to some acceptance probability based on the relative merit of the solution, i.e., the difference between the cost of the neighbor and the base solution. The simulated annealing algorithm further improves the stochastic hill-climber by decreasing the acceptance probability the longer the search takes.

The definition an "interesting" neighbor for a solution S under input M and N, specified by means of `interestingNeighbour`(S, M, N), is clearly depending on the question how to compute a neighbor. As discussed above, there must be some way in the evaluation model to compute, for a given solution, all solutions that are similar with respect to some transformation between solutions. In the following, this transformation is used to provide a function

[99] Michalewicz, Z. and Fogel, D. B. 2000. *How to Solve It: Modern Heuristics*, 1st ed. Springer Verlag.

```
funct neighbours(S, M, N) ≡
  ⌈𝒩 ← ∅
   for each v ∈ V_N do
       x ← v⃗
       S′ ← unmerge(v, x, S, N)
       for each w ∈
           validCandidates(v, N, S′)\{x} do
           S″ ← merge(v, w, N, S′)
           𝒩 ← 𝒩 ∪ {S″}
       od
   od
   𝒩⌋.
```

Figure 6.10: Computing the neighbors of a solution using `validCandidates`

`neighbours`(S, M, N) that computes the set of neighbors of the solution S for the input N and M.

To illustrate this function, we give a definition based on `validCandidates` from the previous section. This definition has the virtue of being applicable, whenever it is possible to test the validity of partial solutions, i.e., if a partial solution can still be completed to a full one and therefore can be applied to each evaluation model that adheres to the requirements posed by the incremental pair mergers. In Figure 6.10 this implementation is detailed: Each vertex v from N is un-merged in S, which is the reverse operation of the `merge` function specified in Figure 6.2, so that S' is identical to S except for $\vec{v}$. In S' all candidates for merging with v except the vertex originally merged with v are considered and merged with v resulting in new solutions that are added to the set of neighbors.

In the following, we denote the maximum size of the set of neighbors of a solution with input N and M as $\nu(N, M)$ and T_{trans} as the time for computing a single neighbor of a solution. Using this notation, $T(\texttt{neighbours}) = O(\nu(N, M) \times T_{trans})$ and, for the implementation of `neighbours` from in Figure 6.10, $T_{trans} = O(T(\texttt{merge}) + T(\texttt{validCandidates}) + T(\texttt{unmerge}))$.

Deterministic Hill-Climber

The basic local search algorithm, referred to as deterministic hill-climber, uses a non-probabilistic acceptance criteria for selecting the "interesting" neighbor: As Figure 6.11 shows, a solution S' among all neighbors of S (generated by `neighbours`) is selected, if it

```
funct interestingNeighbour_det.(S, M, N) ≡
  ⌈S_best ← S
   for each S′ ∈ neighbours(S, M, N) do
       if cost(S′) < cost(S_best)
          then S_best ← S′ fi
   od
   S_best⌋.
```

Figure 6.11: Finding the "interesting" neighbor for the deterministic hill-climber

has the lowest cost among all neighbors including S itself.

The worst-case complexity for a single improvement iteration of the inner loop of the deterministic hill-climber is the maximum number of solutions in the neighborhood for a solution under input N and M times the time T_{trans} for constructing a neighbor for a given solution times $T(\texttt{cost})$. The inner loop is executed for each of the τ = `MAX-TRIALS` independent trials at most ι = `MAX-ITERATIONS-PER-TRIAL` times. For each independent trial also a random solution is generated in time T_{random}. If one uses the random pair merger to generate these random solutions, $T_{random} = O(\mathsf{N} \times (T(\texttt{cost}) + T(\texttt{merge}) + T(\texttt{validCandidates})))$. Therefore, the overall complexity of the deterministic hill-climber is

$$O(\tau \times \iota \times (T_{random} + \nu(N, M) \times (T_{trans} + T(\texttt{cost})))).$$

The advantage of the algorithm is that it improves a given solution very quickly, if possible. The downside of this quick improvement is that this algorithm is prune to become stuck inside local optima. Since never a neighbor with higher or same cost as S is selected, even if the difference in cost is very low, there is no chance to escape from a local optima in the cost function.

Stochastic Hill-Climber

This disadvantage is addressed by the remaining two local search mergers. The stochastic hill-climber improves the deterministic one, by accepting a solution as "interesting" even if the cost of that solution is somewhat higher than the cost of the initial solution.

In Figure 6.12, the stochastic variant of `interestingNeighbour` is shown. It has an additional parameter, here assumed to be a constant, that is used to determine how much influence the

```
funct interestingNeighbour_stoch.(S, M, N, T) ≡
  ⌈S_best ← S
   for each S′ ∈ neighbours(S, M, N) do
       if random[0, 1) < 1 / (1 + e^((cost(S′) − cost(S)) / T))
          then S_best ← S′
                break
       fi
   od
   S_best⌋.
```

Figure 6.12: Finding the “interesting” neighbor for the stochastic hill-climber

relative merit of a solution S' with respect to the original solution S has on the acceptance of S' as “interesting”: The higher T is the less influence the relative merit $\mathtt{cost}(S') - \mathtt{cost}(S)$ has, for very high values of T the stochastic hill-climber degenerates to a random search. On the other hand, a very low T, e.g., $T = 1$, a neighbor S' is only accepted if it has a lower cost than S as in the deterministic hill-climber. [99] provides a more detailed discussion of the influence of T. It should be noted, that the choice of T is clearly not independent of the cost function used.

The stochastic hill-climber has the same worst-case complexity as the deterministic hill-climber, i.e.,

$$O(\tau \times \iota \times (T_{random} + \nu(N, M) \times (T_{trans} + T(\mathtt{cost})))).$$

But in contrast to the deterministic hill-climber it will in most practical cases not consider all neighbors of a solution but only a smaller number, whereas the deterministic hill-climber always traverses all neighbors.

A serious problem of the stochastic hill-climber is often that the acceptance probability remains constant over the entire run of the algorithm. Even at last step it can happen, that a solution is accepted that is clearly worse than the current one.

Simulated Annealing

In [80] a technique borrowed from observations on statistical mechanics, called simulated annealing has been proposed. The derived local search algorithm improves the stochastic hill-climber by reducing T and

```
funct findCommonPlan(N, M) ≡
  ⌈S_best ← N ∪ M
   r ← 0
   T ← T_max
   while r < MAX-TRIES do
     S ← random solution
     t ← 0
     repeat
       T ← T_max · e^(−t·c)
       S′ ← interestingNeighbour(S, M, N, T)
       if S′ = S then break
                else S ← S′ fi
       t ← t + 1
     until T < T_min
     if cost(S) < cost(S_best)
       then S_best ← S fi
     r ← r + 1
   od
   S_best⌋.
```

Figure 6.13: Skeleton of a simulated annealing

thereby the acceptance ratio over the run of the algorithm. In analogy to statistical mechanics, T is referred to as temperature of the algorithm and there are three more parameters, the initial T_{min} and the end temperature T_{max} together specifying the temperature range considered in the algorithm and the cooling ratio c that influences, as the name indicates, the way T is lowered during runtime.

The general skeleton for a local search merger has to be slightly adapted as shown in Figure 6.13: The termination condition of the inner loop is not any more based on the constant `MAX-ITERATIONS-PER-TRY`, but rather on the current temperature in comparison to the minimal temperature. Furthermore, the temperature is cooled down after each iteration using a cool-down formula depending on the number of the iteration t and the cooling ratio c, proposed in [127]

$$T = T_{max} \cdot e^{-tc}.$$

For `interestingNeighbour` the same implementation as for the stochastic hill-climber can be employed.

[80] Kirkpatrick, S., et al. 1983. Optimization by simulated annealing. *Science 220,* 4598, 671–680.

[127] Spears, W. M. 1996. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26. American Mathematical Society, Chapter Simulated Annealing for Hard Satisfiability Problems, 533–558.

Again, selecting the parameters T_{max}, T_{min} and c is far from trivial. Quoting [127] "these choices in parameters will entail certain tradeoffs. For a given setting of MAX-TRIES, reducing T_{min} and/or increasing T_{max} will allow more ... [iterations] to be made per independent attempt, thus decreasing the number ... [of independent tries]. A similar situation occurs if we decrease or increase the decay rate. Thus, by increasing the temperature range (or decreasing the decay rate) we reduce the number of independent attempts, but search more thoroughly during each attempt. The situation is reverse if one decreases the temperature range (or increases the decay rate). Unfortunately it is not at all clear whether it is generally better to make more independent attempts, or to search more thoroughly during each attempt."

The complexity of the simulated annealer depends therefore on these three parameters. More precisely, the inner loop of the algorithm shown in Figure 6.13 is executed

$$r_{max} = \max\{\ n \in \mathbb{N} \mid T_{min} \leq T_{max} \cdot e^{-nc}\ \} = \left\lceil \frac{\ln \frac{T_{min}}{T_{max}}}{c} \right\rceil$$

times for each independent trial.

This leads to a complexity for the simulated annealing algorithm presented here of

$$O(\tau \times r_{max} \times (T_{random} + \nu(N,M) \times (T_{trans} + T(\texttt{cost})))).$$

Again, the simulated annealer seldomly actually traverses all neighbors of a solution, in contrast to the deterministic hill climber.

♦

The above described algorithms could be further improved, e.g., adapting research on problems from graph theory, where improved versions of the simulated annealing approach are considered [138] to SMCSP problem. This remains for future work.

♦

Recalling the traditional set of techniques for solving hard problems, one might notice that one of the more prominent techniques has not been considered here: solving a problem by solving smaller instances of the problem first and combining the results to solve a larger instance, core strategy of divide-and-conquer or dynamic programming algorithms. Actually, there happens to be a reason for this omission: There is no easy way to combine the solutions produced for smaller problem instances into a solution of the larger problem. More precisely, note that even an optimal solution of the SMCSP for a set of n query plans of size m has in worst case a size of $n \times m$. Furthermore, solving the SMCSP problem for a set of n query plans of size m is almost as hard as solving the SMCSP problem for a set of l query plans of size $(n/l) \times m$. Therefore, combining the solutions obtained for subsets of the input is in this case almost as hard as solving the problem directly. Due to this reason, no heuristics based on the divide-and-conquer technique are presented here.

Another obvious area of optimization techniques not covered in this work, are genetic algorithms and evolutionary programs [98]. In particular, genetic algorithms for structural matching [37] might provide a good starting point for deriving evolutionary programs that solve the SMCSP problem. However, any genetic algorithm is based on mutation, crossover and selection. While we have given a transformation function between solutions, that could be extended for mutation, and selection can be based on the cost of a solution, a crossover operation between solutions that are general graphs seems to be hard to find. Finding such a crossover operation, has to be left for future work.

Table 6.1 sums up the various pair mergers together with their complexities. Note, that for the evaluation models that this work is primarily concerned about, such as the ones presented in Chapter 4, the assumptions the incremental pair mergers are based on are fulfilled and the local search pair merger are generally more expensive than the heuristics incremental pair merger, such as the variants of the greedy pair merger, as the maximum size of a neighborhood of a solution $\nu(N,M)$ is in this case $|N| \times |M|$. The experimental evaluation presented in Chapter 9 confirms these theoretical observations.

[138] Xu, L. and Oja, E. 1990. Improved simulated annealing, boltzmann machine, and attributed graph matching. L. Almeida, Ed. LNCS 412. Springer Verlag, 151–161.

[98] Michalewicz, Z. 1996. *Genetic Algorithms + Data Structures = Evolution Programs*, 2nd ed. Springer Verlag.

[37] Cross, A. D. J., et al. 1996. Genetic search for structural matching. In *Computer Vision – ECCV '96*, R. C. B. Buxton, Ed. LNCS 1064. Springer Verlag, 514–525.

```
funct set-smcsp(𝒫, 𝒜) ≡
⌈M ← empty query plan
 𝒪 ← orderQueries(𝒫, 𝒜)
 for each P ∈ 𝒪 do
     M ← 𝒜(P, M)
 od
 M⌋.
```

Figure 6.14: General skeleton for a set merger

6.3 Set Mergers: Algorithms for Merging Sets of Query Plans

Based on the pair mergers proposed in the previous section, we can now define how a set of query plans can be merged into a large plan. The essential idea is that the query plans are merged incrementally, i.e., a set of query plans $\{P_1, \ldots, P_n\}$ is merged by using a pair merger $\mathcal{A}$ to merge P_1 and P_2 into $\mathcal{A}(P_1, P_2)$ that in turn is merged with P_3 and so on. At the end, we obtain $\mathcal{A}(\mathcal{A}(\mathcal{A}(P_1, P_2), P_3), \ldots, P_n)$ which is a solution for the SMCSP with input $\{P_1, \ldots, P_n\}$ and their corresponding queries.

The obvious questions raised are (1) whether the order in which the query plans are considered affects the outcome and, if so, (2) how to determine the best order. The answer to the first question is positive. In general, the order of mergings can affect the result, in particular if the pair merger used computes only an approximate solution.

Since finding the best order in which to consider the query plans is exponential in the number of query plans, we propose four heuristics for determining a sufficiently good order among the query plans. These heuristics follow mostly the skeleton for a set merger shown in Figure 6.14 differing only in the way the query plans are ordered. Let in the following $\mathcal{P} = \{P_1, \ldots, P_n\}$ be the set of query plans considered, $\mathcal{M}$ the pair merger employed with complexity $T(\mathcal{M}, n, m)$ for merging two query plans with size n and m, and $l = \max\{\, |P| \mid P \in \mathcal{P} \,\}$ the maximum size of a query plan:

(1) The *arbitrary order* set merger processes the query plans in no particular order. The advantage is the low overhead over the complexity of $\mathcal{M}$, resulting in an overall complexity of $O(n \times T(\mathcal{M}, n \times l, l))$. This is paid for by a poor quality of the solution in the many cases. Note, that the complexity of $\mathcal{A}$ might depend on the size of its input which here is $n \times l$.

(2) The *initial separate order* set merger orders the query plans initially by their cost in descending order based on the assumption that the query plans with the highest cost have the highest potential for a large gain from merging. The complexity of this merger is $O(n \times (T(\mathcal{M}, l, n \times l) + T(\mathtt{cost})))$.

(3) The second set merger that determines the order of the query plans initially, is the *initial pairwise order* set merger. Instead of using the cost of a query plan, for each query plan P the highest relative gain $g(P)$ if merged with any other query plan is used to determine its priority for merging. More precisely, $g(P) = \max\{\, r \in \mathbb{R} \mid \exists Q \in \{P_1, \ldots, P_n\} : r = (\mathtt{cost}(P \cup Q) - \mathtt{cost}(\mathcal{M}(P, Q))) \,\}$ is used to order the query plans in descending order. The initial pairwise order merge has therefore a complexity at least quadratic in the number of query plans, viz. $O(n^2 \times (T(\mathcal{M}, l, l) + T(\mathtt{cost})) + n \times (T(\mathcal{M}, l, n \times l) + T(\mathtt{cost})))$.

(4) Finally, the *progressive pairwise order* merger implements another greedy heuristic, differing from the previous two ones: Before each merging, the next query plan to merge into the result of the previous mergings is determined by actually merging all query plans into that result and retaining the best such merging as shown in Figure 6.15. This results in a complexity of $O(n^2 \times T(\mathcal{M}, l, n \times l) + T(\mathtt{cost}))$. Whether this or the previous algorithm is faster, depends on the influence the size of the input has on the performance of $\mathcal{A}$.

The experimental evaluation in Chapter 9 actually shows that for the setup considered there, the order in which the query plans are considered has almost no influence on the quality of the result, allowing the fast arbitrary order optimizer to be employed.

This short discussion of extending the pair mergers proposed in the previous section to merging sets of query plans is concluded by an outlook on an algorithm based on the clustered strategy discussed at the begin of this chapter.

```
funct set-smcsp(𝒫, 𝒜) ≡
  ⌈M ← empty query plan
   while 𝒫 ≠ ∅ do
     P_best ← empty query plan
     cost_best ← ∞
     for each P ∈ 𝒜 do
        M′ ← 𝒜(P, M)
        if cost(M′) < cost_best
          then cost_best ← cost(M′)
               M ← M′
               P_best ← P
        fi
     od
     𝒫 ← 𝒫\{P_best}
   od
   M⌋.
```

Figure 6.15: Progressive pairwise order set merger

6.3.1 Pairwise Set Merger: Example for the Clustered Strategy

The pairwise set merger is based on the assumption, that the query plans are clustered with respect to their similarity. In other words, it is assumed that there are sets of very similar query plans such that the query plans inside of one of these sets are rather similar but the similarities among query plans in different such sets is very small.

This intuition can be translated into an algorithm as shown in Figure 6.16: For each query plan P, the query plan $Q \neq P$ is computed where the relative gain, i.e., the gain compared to the case where Q and P are not merged, viz. $\mathtt{cost}(P \cup Q) - \mathtt{cost}(\mathcal{A}(P,Q))$. Q is considered the query plan that is most rewarding to merge P with. Together with Q also the mapping from vertices in P to vertices in Q is determined from $\mathcal{A}(P,Q)$.

Based on this information, the common super-plan M is constructed in the following manner: For each query plan P the just computed best partner for merging Q is considered. If Q has not yet been processed, it is added without merging into M, and P is merged into M using the mapping computed above, i.e., each vertex $v \in V_P$ is mapped to the vertex $z \in M$ mapped to the vertex $w \in Q$ that v has been assigned by the mapping computed in the first step. Q will not be processed any more, if encountered later.

Consider, as an example, four query plans $\{P_1, \ldots, P_4\}$. Assume that the first step (line 6–16 in Figure 6.16) results in *partners* : $P_1 \mapsto P_2, P_2 \mapsto P_3, P_3 \mapsto P_1, P_4 \mapsto P_3$. Such a result can occur, e.g., if $\mathtt{cost}(\mathcal{A}(P,Q)) = \mathtt{cost}(\mathcal{A}(Q,P))$ does not hold in general. The second step (line 19–29 in Figure 6.16) constructs M based on these assignments for *partners* by adding P_2 without merging into the empty query plan and mapping P_1 into the result as specified by *maps*. Since P_2 has already been processed, it is skipped and P_3 is merged into M according to the mapping to vertices from P_1 (which are already mapped into M) computed in the first step. P_4 is merged analogously.

To further illustrate this example, assume $P_1, \ldots, P_4$ as in Figure 6.17(a). Then the following can be observed if we assume a simple cost function based on the operators the vertices use (cf. Chapter 8):

—Both P_2 and P_3 have three vertices with P_1 in common. If we assume that the cost for $\lhd^+$ is higher than the cost for b, P_2 becomes the most promising query plan for merging with P_1.

—With P_2 again both P_1 and P_3 have the same number of vertices in common, but if we assume $\prec^+$ has higher cost than $\lhd$, P_3 is the most promising query plan for merging with P_2.

—P_3 has as discussed three vertices in common with P_2 and P_1. Is the cost of b higher than the cost of b, P_1 is preferable to P_2 for merging with P_3.

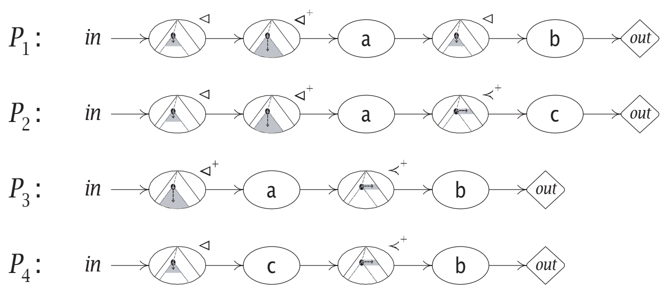
—Finally, P_4 has only two vertices in common with P_1 and with P_3. Again we assume a higher cost for $\prec^+$ than for $\lhd$ and select P_3 as the best query plan for merging with P_4.

Based on these mappings, the query plan shown in Figure 6.17(b) is constructed, that is a solution for the SMCSP with input $\{P_1, \ldots, P_4\}$.

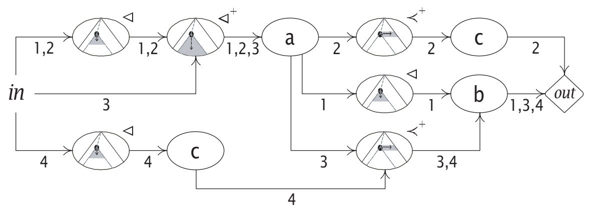
The disadvantage of this algorithm can also be derived from Figure 6.17: The constructed shares considerably less vertices than the optimal query plan shown in Figure 6.17(c) under a cost function that assigns to a query plan a cost based on the number of its vertices. This is due to the fact, that vertices from one query plan are only shared with vertices from a single other query plan, but not from multiple query plans. Therefore, e.g., the $\prec^+$ from P_3 is not shared with the $\prec^+$ from P_2, as vertices from P_3 are only shared with vertices from P_1.

```
funct set-smcsp(𝒫, 𝒜) ≡
 ⌈partners : 𝒫 → 𝒫
  maps : 𝒫 → mapping from vertices of one query plan to vertices of another one

  for each P ∈ 𝒫 do
      gain_best ← ∞
      for each Q ∈ 𝒫\{P} do
          M′ ← 𝒜(P, Q)
          if cost(P ∪ Q) − cost(M′) > gain_best
            then gain_best ← cost(P ∪ Q) − cost(M′)
                 partners(P) ← Q
                 maps(P) ← get map v ∈ V_P ↦ v⃗ ∈ V_Q ∪ V_P from M′
          fi
      od
  od

  M ← empty query plan
  processed : 𝒫 → {true, false}
  for each P ∈ 𝒫 do
      processed(P) ← false
  od
  for each P ∈ 𝒫 do
      if processed(P) then continue fi
      Q ← partners(P)
      if ≠ processed(Q)
        then processed(Q) ← true
             M ← M ∪ Q
      fi
      M ← merge all vertices v from P with vertices in M that are merged with the
               vertex from Q that v is mapped to v by maps(P)
      processed(P) ← true
  od
  M⌋.
```

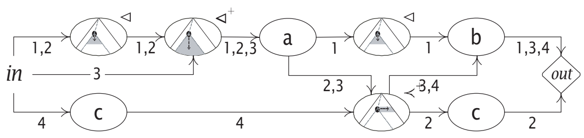
Figure 6.16: Pairwise set merger



(a) Input plans $P_1, \ldots, P_4$



(b) Result of pairwise merger



(c) More compact query plan

Figure 6.17: Example for pairwise merger

The clear advantage of this algorithm is that it uses $\mathcal{A}$ only with input of size l, where l is the maximum size of a query plan in the input, whereas all other set mergers proposed above use $\mathcal{A}$ with input up to $n \times l$ where n is the number of query plans from the input. The construction phase of the pairwise set merger can be implemented linear in n and l, so that the overall complexity is roughly

$$O(n^2 \times T(\mathcal{A}, l, l) + n \times l).$$

♦

The discussion of the pairwise set merger concludes our set of algorithms proposed for merging sets or pairs of query plans into a solution of the SMCSP. The last remaining corner stone in solving the SMCSP is a consideration of the cost functions used to evaluate the cost of a query plan. Such a consideration requires some knowledge about the underlying evaluation engine. Therefore, the next chapter presents a concise overview of the SPEX evaluation engine and how it can be adapted to process multiple queries. Based on the SPEX engine, Chapter 8 specifies several classes of cost functions together with concrete examples tailored in most parts to the SPEX engine. Using these cost functions, the algorithms proposed in this chapter can finally be evaluated in Chapter 9.

Chapter 7

Use Case: SPEX

In this chapter, it is shown how to extend the SPEX evaluation engine to support the evaluation of query plans for evaluating multiple queries as constructed by the algorithms in the previous chapter. After a short introduction into SPEX, the evaluation of a query plan for multiple queries is detailed, in particular for query plans with shared operators that are not part of a prefix of the query plan.

Contents

To facilitate the definition and analysis of reasonable cost functions in Chapter 8 and as basis for the experimental evaluation presented in Chapter 9, we give here a short introduction to the SPEX engine referring to [105; 106] for a more detailed description.

7.1 SPEX in a Nutshell

In [79; 105], a novel evaluation engine for *s*treamed and *p*rogressive *e*valuation of *X*ML queries against streams, called SPEX, has been introduced: A query is translated into a network of transducers closely resembling a query plan as presented in Section 2.3.2. The presented network of transducers combines a worst-case complexity close to the optimal complexity for evaluating queries against XML data in main-memory shown in [53] with several strong advantages over conventional approaches for querying XML streams

— The main feature aside from the good complexity is the high extensibility and flexibility of the approach. As emphasized by the ease of extending SPEX to multiple queries presented in Section 7.2, the network-based approach combined with highly independent transducers provides a framework that can be easily extended with new transducers implementing additional operations or with new methods for combining existing components. This enables the (logical) optimizer as discussed in Chapter 2.3 to freely choose from a large number of different evaluation strategies the preferred way of evaluating a query based, e.g., on estimations about the expected evaluation time.

— Furthermore, [106] proposes a layered approach

---

[105] Olteanu, D., et al. 2003. An evaluation of regular path expressions with qualifiers against XML streams. In *Proc. of the International Conference on Data Engineering (ICDE).*

[106] Olteanu, D., et al. 2003. Advanced techniques for streamed and progressive evaluation of XPath. Research report, University of Munich, Institute for Computer Science.

[79] Kiesling, T. 2002. Towards a streamed XPath evaluation. M.S. thesis, University of Munich, Institute of Computer Science.

[105] Olteanu, D., et al. 2003. An evaluation of regular path expressions with qualifiers against XML streams. In *Proc. of the International Conference on Data Engineering (ICDE).*

[53] Gottlob, G., et al. 2003. The complexity of XPath query evaluation. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS).* 179–190.

[106] Olteanu, D., et al. 2003. Advanced techniques for streamed and progressive evaluation of XPath. Research report, University of Munich, Institute for Computer Science.

to querying XML streams based on the expressiveness of the queries supported: Several classes of queries are identified, differentiated by the manner in which they relate different constraints expressed in the query into path, tree, and graph queries (cf. Section 2.2), or by the operators allowed in such a query. For graph queries, the worst-case complexity of evaluating a query containing only $\lhd$ and $\lhd^+$ together with the corresponding inverse relations is better than if also horizontal relations among elements, such as $\prec^+$, are used.

— Finally, the SPEX components of the SPEX network are simple deterministic push-down transducers with very low computational requirements that can be implemented on simple devices with low CPU utilization. The communication among these transducers is similarly well suited for low-end devices as a transducer only annotates certain elements in the XML stream, resulting in at most a constant overhead over the original input stream.

For further details on the features and properties of SPEX, please refer to [79; 105; 106] and to the graphical front-end [126] that allows to observe a prototype implementation in detail.

To facilitate the description, how query plans for multiple queries can be evaluated by the SPEX engine, a closer look at certain concepts and components of the SPEX network is required. In the following, we present a slightly simplified view of a SPEX network closely resembling the query plans used in this work leaving out certain technical details of no concern to the issue at hand that are discussed in [106].

The essential idea of the SPEX engine is that each operator of a query plan is implemented by a specific transducer. These transducers are connected as specified by the query plan, so that an element from the stream flows through the transducers in the same order as the query plan dictates. Consider, e.g., the query plan from Figure 2.9 for the path query $Q(v2)$:- $\mathsf{a}(v1) \wedge v1 \lhd v2 \wedge \mathsf{b}(v2)$. Starting from

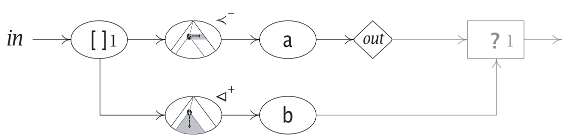


Figure 7.1: Query plan with simple predicate extended by determination network

the stream access operator *in*, an element is first processed by the a label operator, then by the $\lhd$ relation operator, the b label operator, and finally the output operator *out*. When the element passes an operator it is determined, whether the element is selected by that operator. If that is the case, the element is annotated as selected. In the sample query plan from Figure 2.9, the *in* operator selects all elements encountered in the stream, the (transducer corresponding to the) a label operator retains from these only the elements with label a as selected, all other elements are not any more selected. The $\lhd$ operator selects for each such a its children. So after the $\lhd$ operator only the children of an a are selected and then restricted by the b label operator. The *out* operator does not change the selected elements but rather considers all selected elements encountered as part of the result of the query.

So far the evaluation can only handle path queries. To support tree queries resulting in tree-shaped query plans, a means for connecting results of separate parts of the network is needed. Recall, that a tree query results in a predicate operator labeled with [ ]. Figure 7.1 shows a query plan that is extended by a so-called *determination network* depicted in gray. This determination network closely resembles the structure of the query plan, but is inverted. For each predicate operator in the query plan there is a corresponding *determination operator*, here depicted by a box labeled with **?**.

To see, why this determination network is introduced, recall the semantic of the query plan shown: It selects all following-siblings with label a of an element if that has also a descendant b. When evaluating such a query plan the problem arises that when an a element is encountered that is a following-sibling of some element, it must be determined, whether there is also a b descendant of the same element. This test is performed by the determination operator that corresponds to the predicate operator: Whenever a selected element passes the predicate operator, it is annotated with a (unique) condition. This condition is retained

[79] Kiesling, T. 2002. Towards a streamed XPath evaluation. M.S. thesis, University of Munich, Institute of Computer Science.

[126] Spannagel, M. 2003. SPEX Viewer: A graphical user interface for SPEX. Project thesis, University of Munich, Institute for Computer Science.

[106] Olteanu, D., et al. 2003. Advanced techniques for streamed and progressive evaluation of XPath. Research report, University of Munich, Institute for Computer Science.

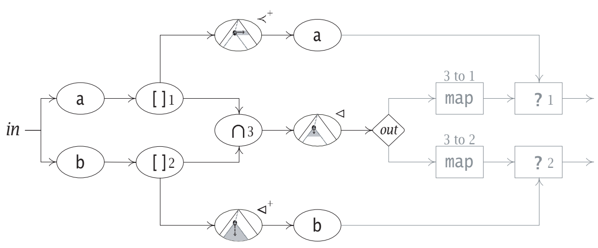


Figure 7.3: Query plan with intersection extended by determination network

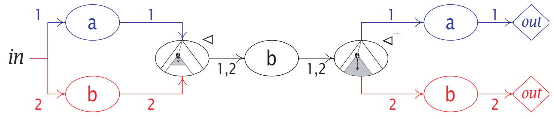


Figure 7.4: Query plan for two queries 1 and 2

when other operators change the selected elements until the element passes the corresponding determination operator. When the determination operator encounters selected elements from both branches that are annotated with the same condition, that condition is determined to be fulfilled and the corresponding result can be generated.

If there are nested predicates in a query plan, the picture gets slightly more complex as shown in Figure 7.2: At the second predicate the conditions for the first predicate are replaced, but a mapping between the conditions for the two predicates is retained, so that after the determination operator for the second predicate, the conditions for the outer first predicate can be obtained by a new operator labeled with `map`. [106] shows how these mappings can be created and that they have no impact on the complexity for evaluating a query with a SPEX network.

Finally, graph queries can as well be handled by this approach. Consider, e.g., the query plan and determination network shown in Figure 7.3. At the intersection operator, elements from both incoming edges carry conditions. To keep the annotation of elements in the stream constant, the intersection operator creates a new condition if from both incoming edges a selected element is encountered. This condition has to be mapped back to the original conditions in the determination network as shown in Figure 7.3.

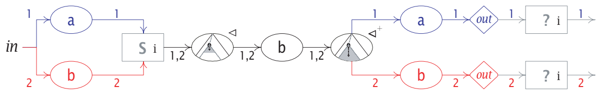


Figure 7.5: Query plan for two queries 1 and 2

7.2 Evaluating Query Plans for Multiple Queries

The extension to multi-query plans proves to be very natural and easy: Consider the multi-query plan shown in Figure 7.4. In this query plan the two queries 1 and 2 share a common path in the middle of the query plan.

In such a query plan, a new operator, referred to as *shared path begin* operator and depicted as a box labeled **S**, is introduced at the point where the shared path starts. When a selected element flows through this operator from one of the incoming edges it is annotated with a condition. It is furthermore recorded for which of the incoming edges (or queries), a selected element is encountered, since it is very possible (in the case of Figure 7.5 actually guaranteed) that the element is selected only on one of the incoming edges. Here, an element is selected either by query 1, if it is a a element, or by query 2, if it is an **b** element, or by none of them.

In the determination network at the end of the query plan, shown in Figure 7.5 the queries are separated again if they are not already as in this case and for each query a determination operator is used that determines a condition only, if a selected element has been encountered for that query by the corresponding shared path begin operator when the condition was created.

Interestingly, this extension to the SPEX network proves to be very efficient: The overhead introduced by sharing a path is not larger than the cost of a predicate operator and occurs only at the beginning of the path. The more costly the shared path is (e.g., if it is long or entails expensive operators), the lower is the overhead of sharing per operator. To reflect this overhead, a cost function $\kappa_{merging}$ is defined in the following section that penalizes vertices with incoming edges that are assigned to different sets of queries, i.e., the cost of vertex at the begin of a shared path is increased roughly by the cost of a predicate operator.

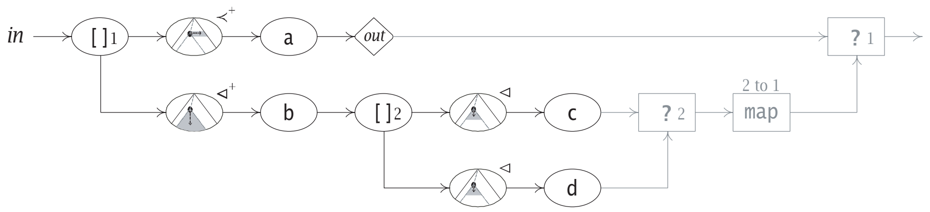


Figure 7.2: Query plan with nested predicate extended by determination network

Chapter 8

Cost Estimation in a Streamed Environment

As the final part of the optimization framework, this chapter discusses the cost estimation of query plans. Cost functions are classified by their complexity with respect to the size of a query plan and for each of the classes a generic cost function as well as a cost function specific to SPEX is proposed.

Contents

Recall, that part of an evaluation model as defined in Chapter 4 is a cost function that assigns a cost to each valid query plan. In this chapter, a natural class of cost functions, called *vertex-based* cost functions, is investigated characterized by the assumption, that a cost function can be extended to the vertices of a query plan such that the cost of that query plan can be computed from the costs of the vertices in linear time. In other words, we assume that for an evaluation model $\mathcal{E}$ with a cost function $c_{\mathcal{E}}$, there is a function $c_{vertex} : V_P \to \mathbb{R}_0^+$ where P is a query plan from $\mathcal{E}$, such that $c_{\mathcal{E}}(P) = \alpha + \beta \sum_{v \in V_P} c_{vertex}(v)$ for some constants $\alpha, \beta \in \mathbb{R}_0^+$.

In the following, we extend $c_{\mathcal{E}}$ to vertices such that for all vertices v in a valid query plan of $\mathcal{E}$, i.e., for all $v \in \bigcup_{p \in P_{\mathcal{E}}} V_P$, $c_{\mathcal{E}}(v) = c_{vertex}(v)$.

The advantage of such cost functions is that they are easy to define, allow to compare vertices based on their cost (as required, e.g., by the initial greedy merger discussed in Section 6.2.1), and can naturally be broadened to partial solutions (as demanded, e.g., by the incremental pair mergers proposed in Section 6.2.1.

In the following sections, this class of cost functions is further investigated.

8.1 Classes of Cost Functions

In Section 6.2 several pair mergers for merging two query plans have been proposed. The complexity of all mergers depends among other factors also on the time required for computing the cost of a query plan, denoted there as $T(\texttt{cost})$. Based on this observation, we distinguished here the vertex-based cost functions further by the manner in which they compute the cost of a vertex:

Independent vertex-based cost function. A vertex-based cost function is called *independent*, if the cost of a vertex is depending on properties of that vertex only and is independent of the rest of the graph. Furthermore, the cost of the vertex is unaffected if other vertices or edges in the graph are changed. The time for computing the cost of a vertex with a such a cost function is constant with respect to the number of vertices in a query plan.

Local vertex-based cost function. If at most the local neighborhood of a vertex in the query plan, e.g., incident edges or adjacent vertices, can affect the cost of a vertex, the cost function is referred to as *local* cost function. The time for computing the cost of a vertex with a local cost function is, with respect to the number of vertices in the query plan, bounded by the degree of the query plan.

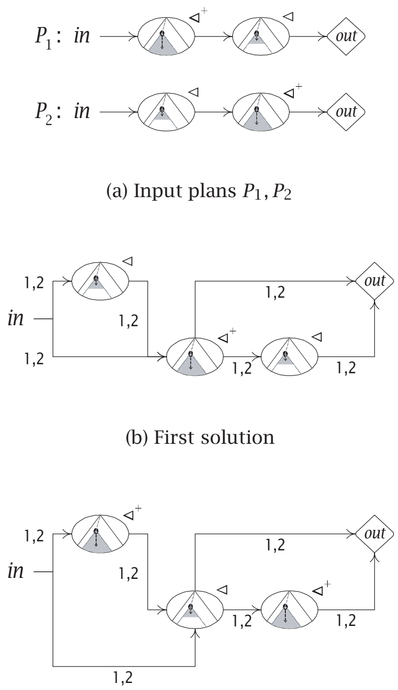
Global vertex-based cost function. Finally, a *global* cost functions does not restrict the number of vertices that the cost of a single vertex might depend on. The time for computing the cost of a vetex with a global cost function is linear in the number of vertices in the query plan. Under such a cost function, any change in the graph can affect the cost of any vertex.

In the following, each of the classes is detailed with concrete examples for cost functions used in the experimental evaluation presented in Chapter 9. The different cost functions are motivated with examples based on the query plans introduced in Section 2.3.2. Recall, that the corresponding evaluation model $\mathcal{X}$, cf. Section 4.3, allows only acyclic query plans.

8.1.1 Independent Cost Functions

The most basic independent cost function, denoted as $\kappa_{vertices}$, assigns to each vertex the same constant cost α, i.e., for all vertices v, $\kappa_{vertices}(v) = \alpha$. Note, that a solution of the SMCSP under this cost function is the valid query plan with the smallest number of vertices. Under an evaluation model $\mathcal{E}$, such that for any graph G there are π, τ, q such (G, π, τ, q) is a valid query plan for $\mathcal{E}$, the SMCSP with $\kappa_{vertices}$ as objective function resumes to the minimum common super-graph problem.

But in most practical cases, such a cost function does not provide a good estimate for the expected cost of evaluating a query plan. Consider, e.g., for evaluation model $\mathcal{X}$ the two query plans P_1 and P_2 from Figure 8.1(a). Recall, that only acyclic query plans are valid in $\mathcal{X}$. Therefore, either the two $\lhd$ or the two $\lhd^+$ vertex from P_1 and P_2 can be shared but not both, since the resulting query plan is acyclic. $\kappa_{vertices}$ gives no indication which of the two query plans from Figure 8.1(b) and 8.1(c) is preferable, they are assigned the same cost. For the SPEX engine, the evaluation of a $\lhd^+$ operator however proves to be more expensive than a $\lhd$ operator, therefore a cost function should be able to express that the alternative solution is preferable.



(a) Input plans P_1, P_2

(b) First solution

(c) Alternative solution

Figure 8.1: Relevance of edge count

For a query plan $P = ((V, E), \tau, \pi, q)$ under an evaluation model $\mathcal{E}$, let

$$\tau' : V \to R_{\mathcal{E}} \cup \{\bot\}$$
$$v \in V \mapsto \begin{cases} \tau(v) & \text{if } \exists r \in R_{\mathcal{E}} : \tau(v) = r \\ \bot & \text{otherwise} \end{cases}$$

where $\bot \notin R_{\mathcal{E}}$ represents the case where no property is assigned to a vertex.

Using this definition, we define a new cost function $\kappa_{operators}$ that is based on a characterization of the operators and properties of the corresponding evaluation model. Let $f : O_{\mathcal{E}} \times R_{\mathcal{E}} \cup \bot \to \mathbb{R}_0^+$ be a function specific to the evaluation model $\mathcal{E}$ that assigns to each pair of operator and property a cost for evaluating such an operation. Then the cost of a vertex v is $\kappa_{operators}(v) = f(\pi(v), \tau'(v))$.

Table 8.1 shows a part of an exemplary mapping f based on analytical and experimental evaluation of the SPEX evaluation engine. Note, that the costs are relative, i.e., for comparing operator-property pairs among each other.

8.1.2 Local Cost Functions

Local cost functions allow to take the neighborhood of a vertex, i.e., the incident edges and adjacent ver-

operator o	property r	$f(o,r)$
$\lhd$	$\perp$	5
$\lhd^+$	$\perp$	$5+10$
$\prec$	$\perp$	$5+2$
$\prec^+$	$\perp$	$5+2+10$
label	string s	$\mathsf{s}/8+10$
in	$\perp$	0

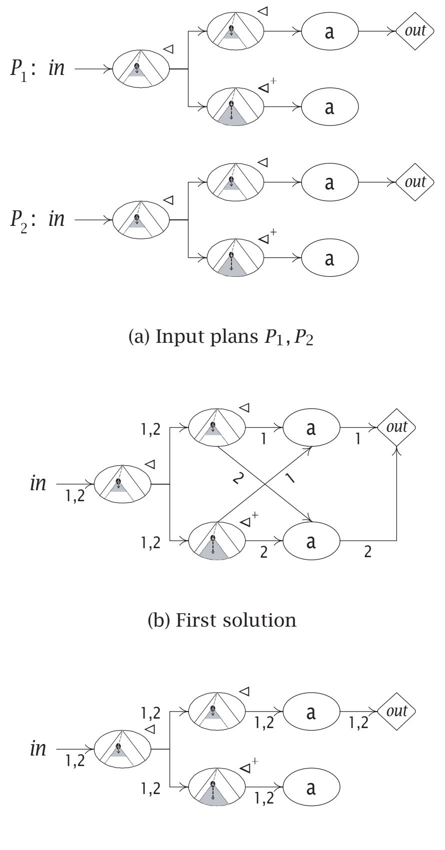
Table 8.1: Relative cost for evaluating operator-property pairs in SPEX

tices, into consideration when computing its cost. As discussed in the previous section, whenever the incoming edges of a vertex are shared among different sets of queries, certain costly measures must be taken in SPEX to ensure the correct evaluation of the query plan. This is likely to be true for many other evaluation engines. Therefore, cost functions that are concerned about the vertices only can in many cases not reflect the expected cost for evaluation of a query plan accurately enough.

Consider, e.g., the two identical query plans P_1 and P_2 from Figure 8.2(a). Cost functions such as $\kappa_{vertices}$ or $\kappa_{operators}$ assign the same cost to the two query plans shown in Figure 8.2(b) and 8.2(c) although for SPEX the second alternative is clearly preferable.

A first attempt, to address this issue, is κ_{edges} that assigns to each vertex a cost based on the number of incident edges, i.e., $\kappa_{edges}(v) = \alpha + \beta|edges(v)|$ where *edges* is defined as in Section 4.3 and α, β are some constants. If $\alpha = \beta = 1$, the query plan from Figure 8.2(b) has a cost of 23, whereas the alternative solution has only a cost of 21.

κ_{edges} allows to affect the cost of a vertex by the *number* of incident edges, but is still not able to cope with the initial problem that the cost of vertex with incoming edges shared among different sets of queries must be clearly higher than for a vertex where the edges are shared among the same set of queries. For illustration, consider the query plans P_1 and P_2 together with two possible common super-plans shown in Figure 8.3. The first solution has actually better cost under κ_{edges} for $\alpha = \beta = 1$. But for the SPEX engine, the alternative solution performs often better since the shared vertices (the $\lhd$ and the *out* operator) are rather cheap, so that sharing them does not justify the cost introduced by the fact that they have now both incoming edges shared among different sets of queries.

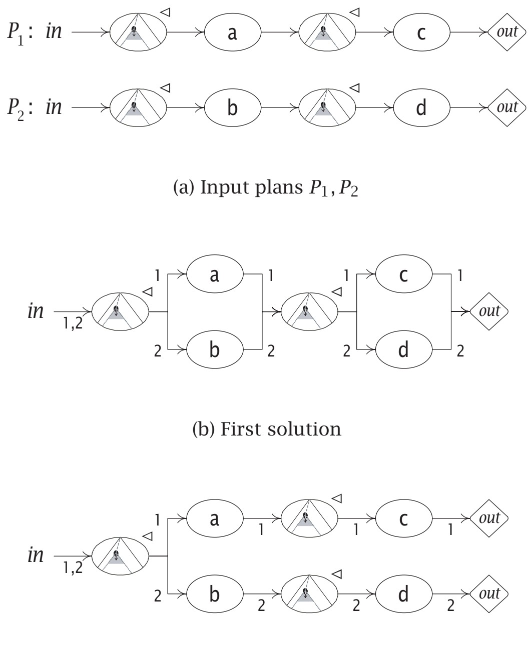


(a) Input plans P_1, P_2

(b) First solution

(c) Alternative solution

Figure 8.2: Relevance of edge count



(a) Input plans P_1, P_2

(b) First solution

(c) Alternative solution

Figure 8.3: Relevance of continuity

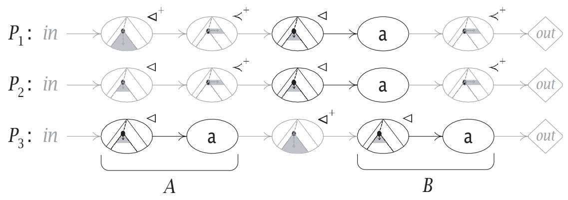


Figure 8.4: Relevance of selectivity

Therefore, we introduce a second local cost function, $\kappa_{merging}$ that penalizes vertices with incoming edges assigned to different sets of queries. More precisely, for a vertex v in a query plan $P = ((V,E),\tau,\pi,q)$, let $\gamma = |\{\, \mathcal{Q} \mid \exists e \in E : \exists w \in V : (w,v) = e \wedge q(e) = \mathcal{Q} \,\}|$ be the number of different sets of queries assigned to an incoming edge of v and α, β some constants, then

$$\kappa_{merging}(v) = \kappa_{operators}(v) + \begin{cases} 0 & \text{if } \gamma = 1 \\ \alpha + \beta \cdot \gamma & \text{otherwise} \end{cases}.$$

8.1.3 Global Cost Functions

The final class of vertex-based cost function may consider the entire graph for computing the cost of a vertex. Here, we present only one example for this class of cost functions, the selectivity based cost function $\kappa_{selectivity}$ that is based on $\kappa_{merging}$ but introduces an additional bias based on the estimated selectivity of a vertex based on the operator-property pair assigned to it. Figure 8.4 shows three query plans P_1, P_2, and P_3. A common super-plan of P_1 and P_3 or P_2 and P_3 can contain the $\lhd$–a subgraph from P_1 respectively P_2 shared with position A or B in P_3 assuming the two $\lhd^+$ operators are not shared. Under $\kappa_{merging}$ both positions yield the same cost for the common super-plan. Observe, that the difference between P_1 and P_2 lies in the second vertex, where P_1 uses a $\lhd^+$ operator and P_2 a $\lhd$. Furthermore, note that in P_3 there is $\lhd^+$ before B. Therefore, the $\lhd$–a subgraph in P_1 and B process both a far large number of a elements in the stream, than the subgraph in P_2 or A. Sharing the subgraph from P_2 with B therefore will affect the processing time more than sharing with A.

The selectivity of an operator-property pair is estimated similar to the cost estimation for relation queries without statistics on the data, cf. [51]. Such an estimation can approximate the selectivity of many operators only very poorly, but has the advantage that it does not require knowledge about the stream and its characteristics during optimization.

Using statistics about the stream is not considered here and left for future work. Based on statistical data collected either a-priori or during the processing, the cost estimation could be further refined as shown in [109; 1; 84; 116; 137].

Based on the cost functions proposed in this chapter, Chapter 9 finally provides an experimental evaluation of the techniques and algorithms proposed.

[51] Garcia-Molina, H., et al. 2001. *Database systems: the complete book*, 1st ed. Prentice Hall, Upper Saddle River, New Jersey.

[109] Ozkan, C., et al. 1995. A heuristic approach for optimization of path expressions. In *Proc. of the International Conference on Database and Expert Systems Applications.* 522–534.

[1] Aboulnaga, A., et al. Estimating the selectivity of XML path expressions for internet scale applications. In *Proc. of the International Conference on Very Large Databases (VLDB).* 2001.

[84] Lim, L., et al. 2002. XPathLearner: An on-line self-tuning markov histogram for XML path selectivity estimation. In *Proc. of the International Conference on Very Large Databases (VLDB).*

[116] Polyzotis, N. and Garofalakis, M. 2002. Statistical synopses for graph-structured XML databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data.*

[137] Wu, Y., et al. 2002. Estimating answer sizes for XML queries. In *Proc. of the International Conference on Extending Database Technology (EDBT).* 590–608.

Chapter 9

Experimental Evaluation

Based on the evaluation model of the SPEX engine and the cost functions defined in previous chapters, the algorithms can now be evaluated thoroughly. In this chapter, it is shown that there are several pair mergers that if combined with a set merger produce good solutions in an acceptable amount of time. In particular, the initial greedy merger proves to construct solutions that are at least as good as the solutions of most of the more expensive pair mergers for the kind of queries considered here. Finally, the results on the complexity of the pair and set mergers discussed in Chapter 6 are confirmed by the presented experiments.

Contents

In this chapter, the algorithms presented in Chapter 6 are finally evaluated for a large set of RPQ query plans, cf. Section 2.2. Extensive test have been performed for most of the proposed pair and set mergers against several query workloads with varying characteristics. To illustrate the results, the following section introduces the setup of the experiments, in particular the different query workloads together with their properties.

9.1 Setup

A prototype implementation of the proposed query plans and algorithms for solving the SMCSP in Java [74] is described in Chapter 10: Based on a graph library specifically optimized for efficient iteration over the incident edges of a vertex, efficient implementations of the most important graph operations for a query plan are provided. The cost functions proposed in the previous chapter are implemented using memorization, allowing an incremental update of the cost of a graph upon change.

The following tests have all been performed using Sun Hotspot JRE 1.4.1 under Linux 2.2 running on a AMD Athlon with 1, 3 GHz and 500 MB of main memory.

The basis for all tests is the SPEX evaluation engine reviewed in Chapter 7. Therefore the evaluation model $\mathcal{X}$, defined in Section 4.3, is employed: Recall, that $\mathcal{X}$ restricts query plans to acyclic graphs. In Chapter 6, an implementation of `validCandidates` for $\mathcal{X}$ is given, that allows, given a valid partial solution, to determine pairs of vertices such that merging one of

[74] Joy, B., et al. 2000. *The Java Language Specification*, 2nd ed. Addison-Wesley.

these pairs does not violate the validity of the partial solution. Combined with one of the vertex-based cost functions in Chapter 8 that all naturally extend to partial solutions, the two requirements for incremental pair mergers are fulfilled by $\mathcal{X}$.

Actually, the initial greedy pair merger, an incremental pair merger, proves to give the best trade-off between solution quality and time for computing a good solution outperforming the more costly local search pair merger. Note, that this result can not be generalized to arbitrary evaluation models, that do not admit to the above requirements for incremental pair mergers.

9.1.1 Workloads

As input for the test, we use five different workloads each consisting in 10.000 query plans based on $\mathcal{X}$. When testing algorithms with high complexity, only smaller subsets of these workloads are considered. The five workloads differ with respect to the characteristics of the contained query plans.

The query plans contained in the workloads have been generated by a query generator based on a DTD. The query generator ensures that all the generated query plans confirm with a given DTD, i.e., contain only structural and label constraints allowed by the DTD. It can be configured with a large number of parameters affecting the structure of the query plans constructed from the generated query plans. In particular, the number and distribution of relation operators and the shape of the query plans can be determined by appropriate parameter settings. If no DTD is provided, random query plans are generated where the label constraints are random strings and the relation operators are chosen according to the specified distribution. In these tests, we have configured the generator to generate tree-shaped query plans with a low (5) maximum degree for a vertex. The relation operators are distributed equally, where possible (i.e., if the DTD allows a choice among several relations, each of these relations is selected with the same probability). Since the generator is based on DTDs only, no text operators are created.

NITF-workload. The first workload is based on the NITF-DTD [71], defining a commonly used format for encoding and exchange of news related information. This DTD is rather large (roughly 200 elements defined), richly structured, highly recursive, and allows very heterogenous instances, with a high degree of freedom similar to the HTML-DTD. Figure 9.1 shows several exemplary query plans from the NITF-workload. In general, these query plans are rather diverse just like the NITF documents they are to be evaluated against. Therefore, the NITF-workload provides a good approximation of querying document-centric XML.

COURSES-workload. In contrast to the NITF-workload, the second workload, referred to as COURSES-workload, is meant to mimic applications involving data-centric XML. The base DTD of this workload is designed for encoding information about university courses (cf. [100]). It is very small (only 16 elements defined), slightly recursive, and very rigid with respect to the allowed structure of an instance. This is reflected in the query plans, cf. Figure 9.2, by a higher similarity within the workload compared to the NITF-workload. Whereas the distribution of the relation operators is comparable to the NITF case, the query plans differ considerably with respect to the label operators. Most of the query plans in this workload contain, for example, several `courses` label operators.

RANDOM-workload. Where the NITF- and the COURSES-workload should provide sets of query plans similar to practical cases, the RANDOM-workload and the following REPEATED-workload represent extreme cases. The query plans in the RANDOM-workload are not based on a DTD but rather generated entirely random based only on the specified distribution of the relation operators, here an equal distribution among $\lhd$, $\lhd^+$, $\prec$, and $\prec^+$. The resulting query plans shown exemplary in Figure 9.3, are extremely diverse and contain almost no label operators with the same label. Therefore, the gain by sharing operators among these query plans is expected to be extremely low.

REPEATED-workload. The converse case is represented by the REPEATED-workload, that exists in two variants. The first variant, REPEATED-1x10000, repeats the same query plan $10,000$ times, the second

[71] International Press Telecommunications Council. News industry text format (NITF). `http://www.nitf.org`.

[100] Miklau, G. XML data repository. `http://www.cs.washington.edu/research/xmldatasets/`, University of Washington.

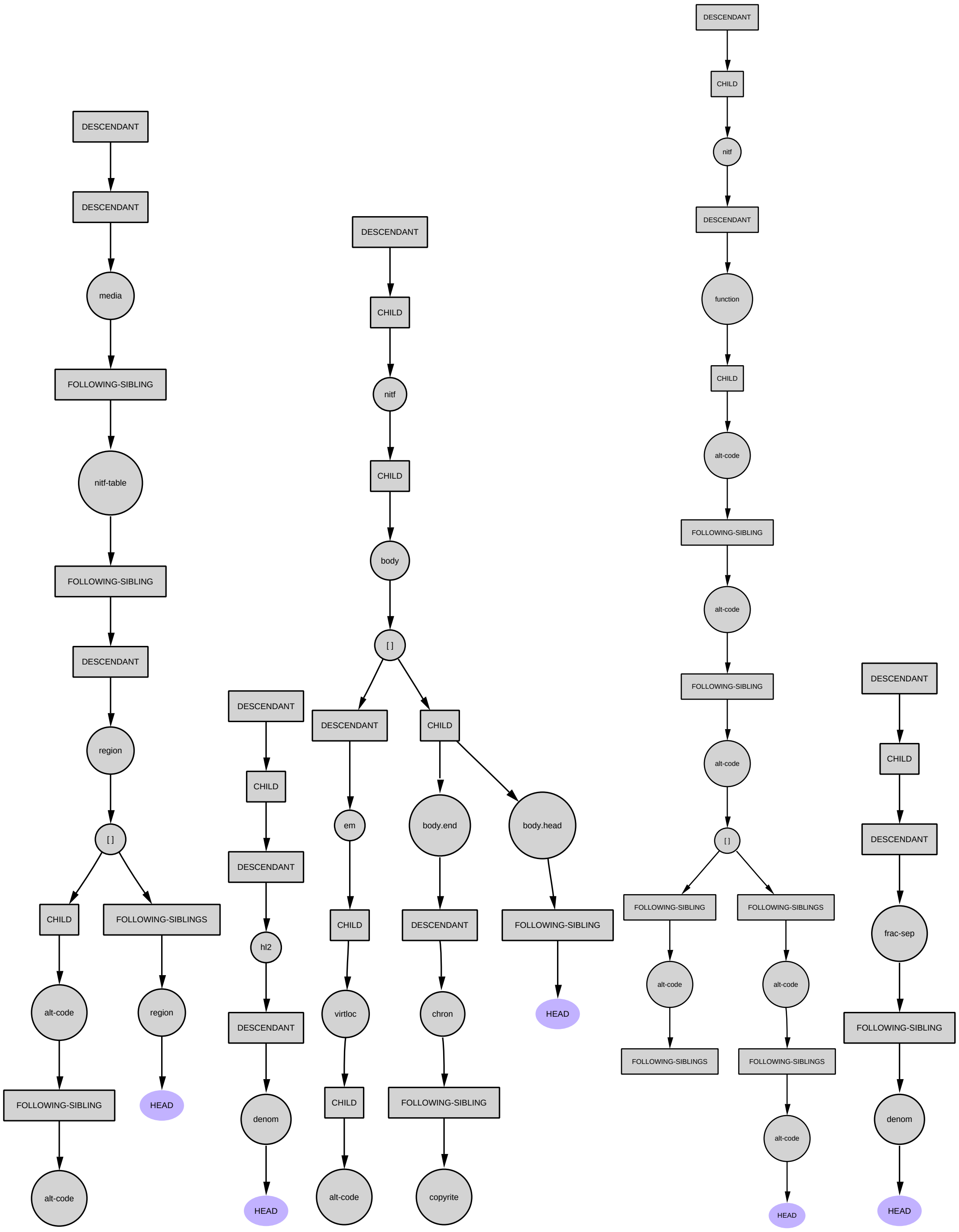


Figure 9.1: Sample query plans from NITF-workload

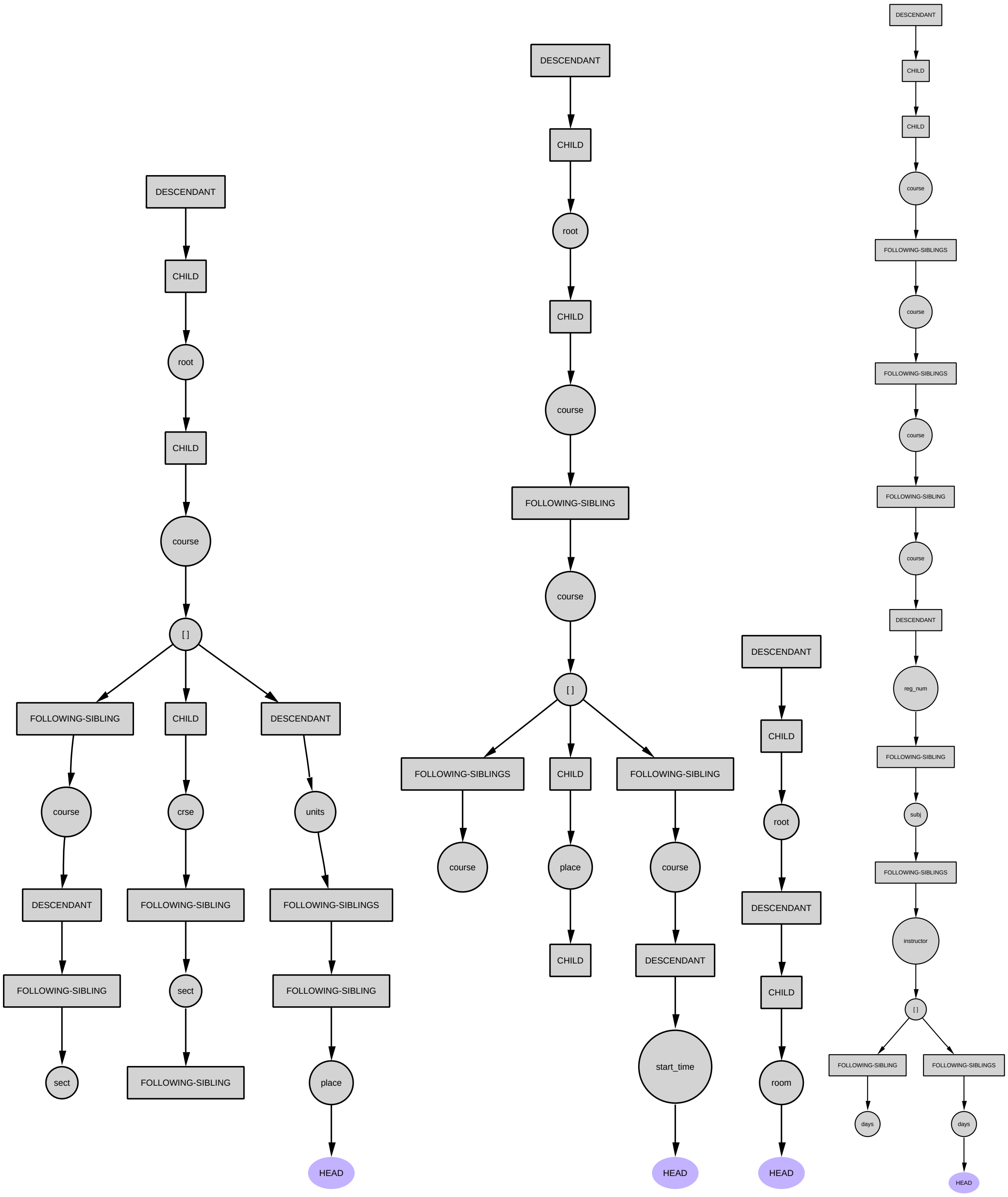


Figure 9.2: Sample query plans from COURSES-workload

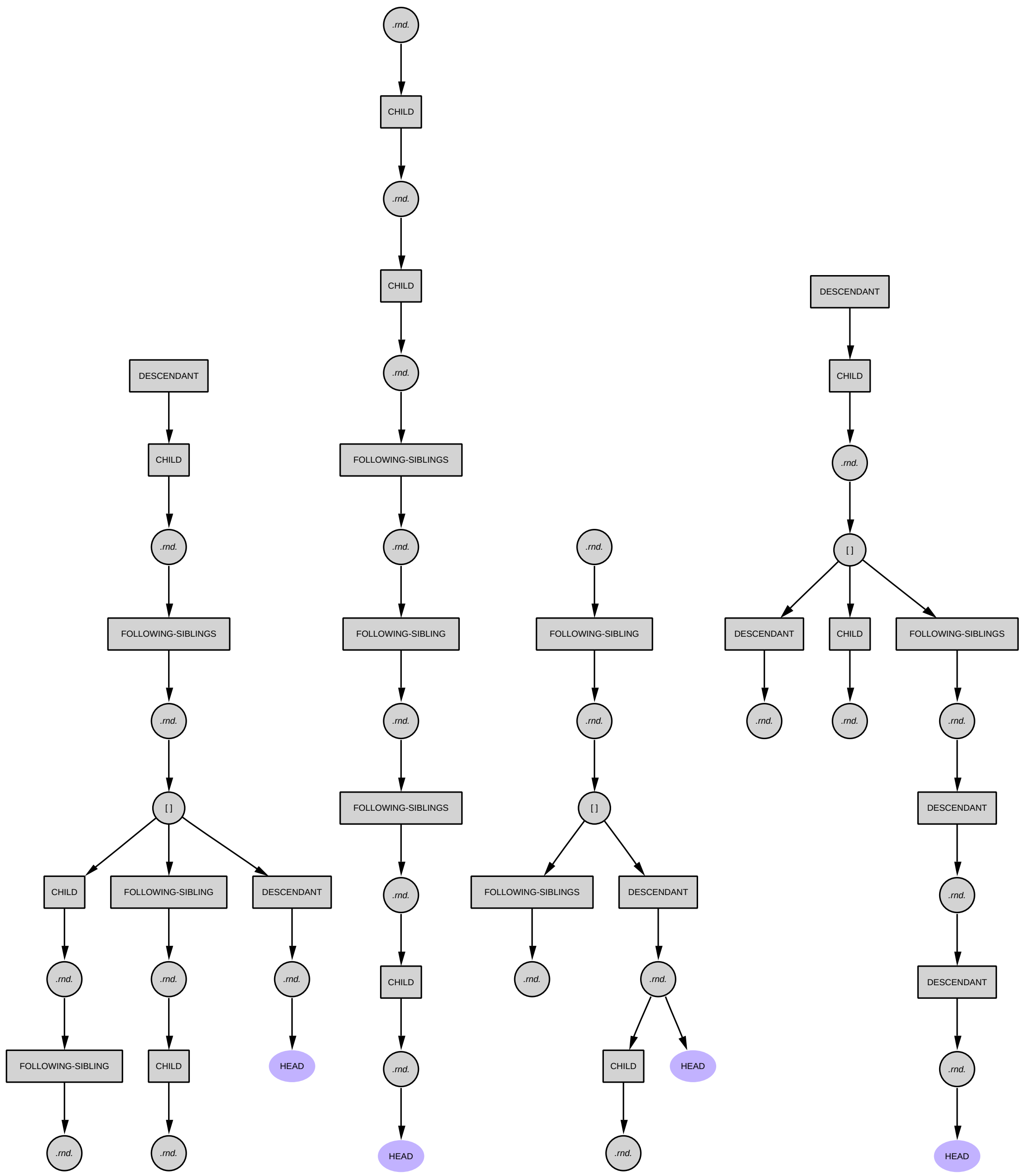


Figure 9.3: Sample query plans from RANDOM-workload (*.rnd.* indicates a random label)

variant, REPEATED-100x100, contains for each of one hundred original query plans one hundred duplicates of that original. In both cases, the original query plans are from the NITF-workload. The REPEATED-workload is therefore used to test how good the different heuristic algorithms recognize identical query plans.

DEVIATED-workload. The final workload is similar to the REPEATED-workload in that it contains repetitions of the same query plans. But this time, the query plans are not repeated exactly, but rather each time slightly deviated. Such a deviation can entail that a relation operator is changed to another relation operator or a label is changed to a random string. The number of deviation from the base query is, for each deviated query, random in the range of $[0, 5]$. Figure 9.4 shows a base query and two deviations. As in the repeated case, there are two variants, DEVIATED-1x10000 and DEVIATED-100x100, with $10,000$ query plans based on the same query and 100 query plans based on 100 different base queries, respectively.

As mentioned above, all workloads have in common that the query plans are tree-shaped and do not contain vertices where two children of that vertex have operator-property pairs that can be merged. Therefore, the queries adhere to all requirements of the tree prefix pair merger. Despite the favorable workload, the experiments show that the initial greedy pair merger, although outperformed by the tree prefix pair merger, can produce clearly superior solutions except on the REPEATED-workload (see Section 9.2) in reasonable time compared to the tree prefix pair merger.

Another commonality among the workloads is that the query plans within a single workload differ only in a limited range. E.g., the queries in all workloads have at most 30 vertices and very seldom less than 10. Furthermore, the maximum degree of the vertices is by choice rather low.

Further studies with more diverse workloads, in particular with graph-shaped queries, where the tree prefix pair merger is presumed to perform far worse, are left for future work.

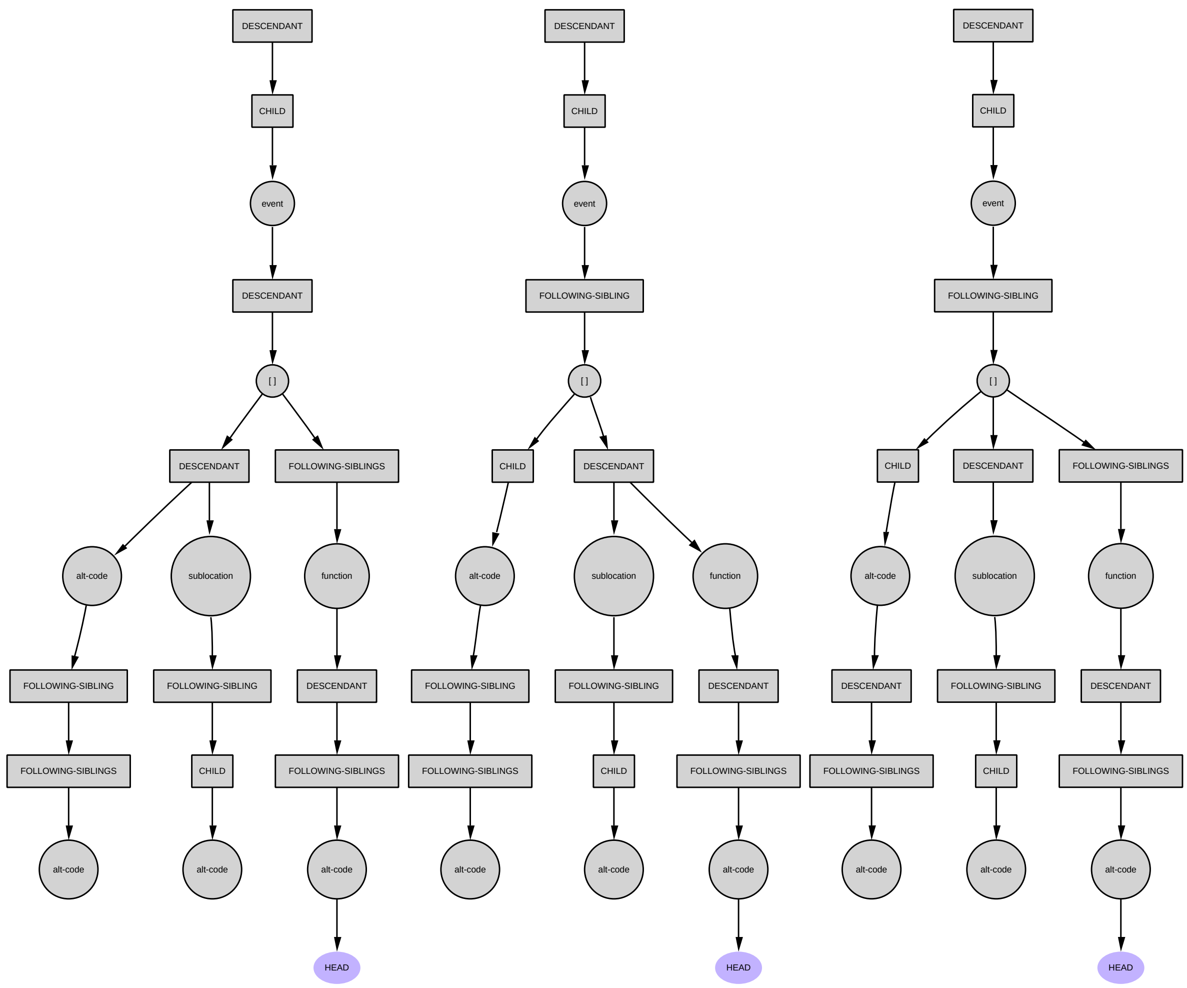


Figure 9.4: Sample deviated queries (first query is base)

9.2 Assessing the Feasibility of the Approach

Based on the experimental setup presented in the previous section, we present here the results of comparing four pair mergers, the plain, the random, the tree prefix, and the initial greedy pair merger, under the arbitrary order set merger and two different local cost functions, $\kappa_{mergings}$ and κ_{edges} with $\alpha = \beta = 1$. The arbitrary order set merger has been chosen, as experimental evaluation on the query workloads presented here indicates that the order in which the query plans are considered does not have a large affect on the quality of the result, cf. Section 9.4. The choice of the cost functions is based on the observation that the independent cost functions fail to provide an acceptable approximation of the actual evaluation time for the SPEX engine and that the global cost function $\kappa_{selectivity}$ is often too expensive to be used, since it has quadratic complexity for computing the cost of the entire query plan in contrast to the (almost) linear complexity for the local cost functions (recall, that the maximum degree of the vertices in all workloads is very low).

Only the above mentioned four pair mergers are considered since the tree prefix and the initial greedy pair merger prove to provide the best time-quality trade-off on the setup considered here. The plain and random pair merger are used for comparison only: The plain pair merger illustrate the cost if there is no sharing at all among the query plans, whereas the random pair merger is used to gauge the other incremental pair mergers, in particular the initial greedy pair merger. In Section 9.3 the remaining pair mergers are evaluated on subsets of the workloads.

9.2.1 Comparing the Cost

Figure 9.5 (9.6) and 9.7 (9.8) show the absolute cost of the solution (the average estimated cost for evaluation per query) computed by the four pair mergers versus the number of queries considered for $\kappa_{merging}$ and κ_{edges} respectively. In both cases, the initial greedy pair merger delivers are very good solutions over all workloads except for the entirely random queries. There, the delivered solution is still better than for the remaining pair mergers tested, but very near in cost to the solution provided by the plain merger where no operators are shared among the query plans.

Interestingly, the relative distance of the quality of the solutions for the initial greedy and the plain pair merger is more than twice under κ_{edges} than under $\kappa_{merging}$. This can be explained by the observation, that the query plans in the RANDOM-workload use random strings as label constraints. Therefore, almost only relation operators can be shared among two query plans. Since in most query plans relation and label operators are intertwined, sharing the relation operators leads to the case where one vertex is shared, its child (or children) are not shared, but their children might again be shared. Recall from Chapter 8, that this case is penalized by $\kappa_{merging}$. The same reason explains to the anomaly shown in Figure 9.5(c), where the solution produced by the random pair merger is worse than the solution where nothing is shared.

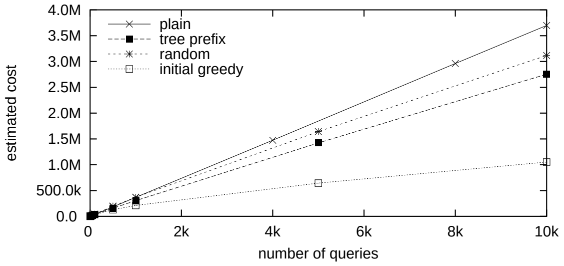
Another expected, but important observation is that the tree prefix pair merger can only outperform the initial greedy pair merger for the two REPEATED-workloads and there only by a small margin. In all other cases, the simple heuristic of the tree prefix pair merger fails to produce solutions that have a comparable quality to the ones constructed by the initial greedy pair merger. The less diverse the query plans, the more likely the tree prefix pair merger can produce a good solution, e.g., it produces solutions that are closer to the solution of the initial greedy pair merger for the COURSES- than for the NITF-workload and similarly for the DEVIATED-1x10000- than for the DEVIATED-100x100-workload.

Note, finally, the performance of the random pair merger: it is consistently outperformed by the initial greedy pair merger with respect to the quality of the solution but can provide solutions almost as good or even better than the prefix pair merger on many workloads, in particular for κ_{edges}. This results from the fact, that the random pair merger, although choosing randomly among the `validCandidates` still considers for all vertices all valid candidates for merging, whereas the tree prefix pair merger only considers the corresponding prefix vertex, cf. Chapter 6.

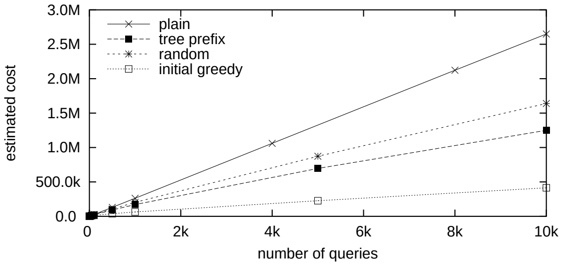
Peeking at Vertices and Edges

These results on the cost of the produced solution are further illustrated by considering the vertices and edges of these solutions, as depicted in Figure 9.9 and 9.10 for $\kappa_{merging}$.

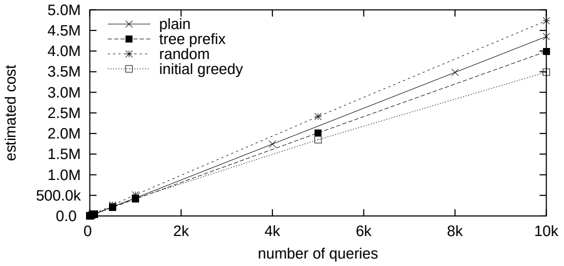
Interestingly, the number of vertices in a solution



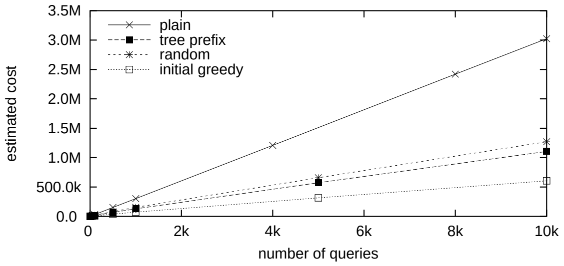
(a) NITF-10000-workload



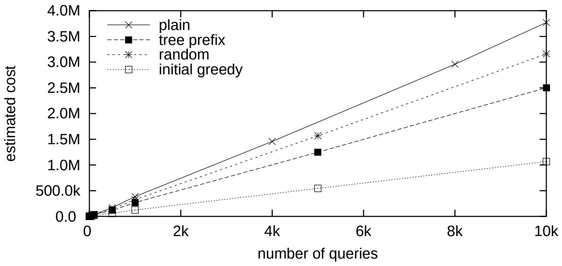
(b) COURSES-10000-workload



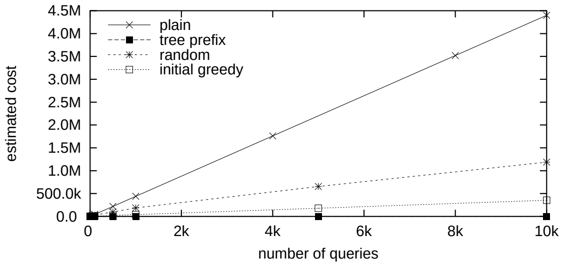
(c) RANDOM-10000-workload



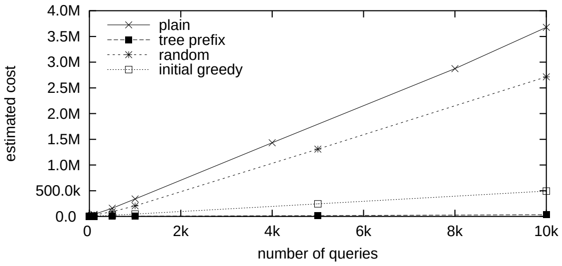
(d) DEVIATED-1x10000-workload



(e) DEVIATED-100x100-workload

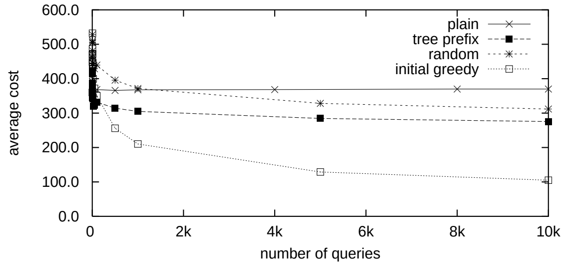


(f) REPEATED-1x10000-workload

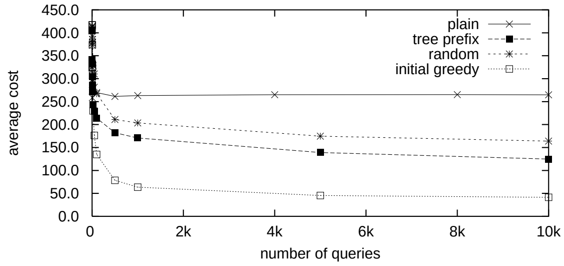


(g) REPEATED-100x100-workload

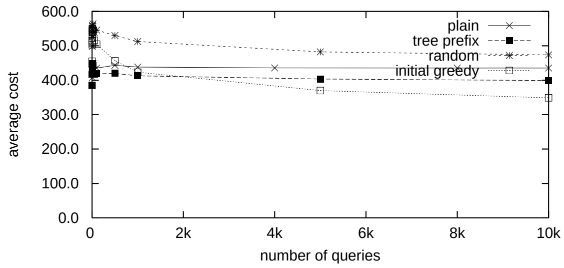
Figure 9.5: Cost of the generated solutions under $\kappa_{merging}$



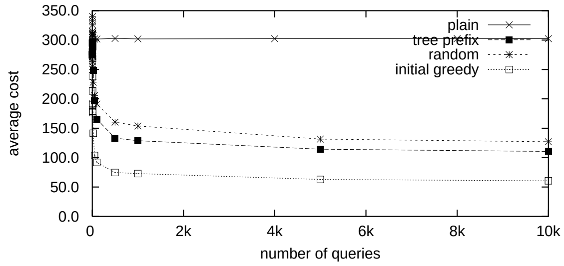
(a) NITF-10000-workload



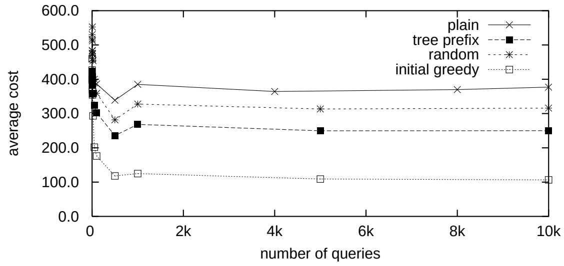
(b) COURSES-10000-workload



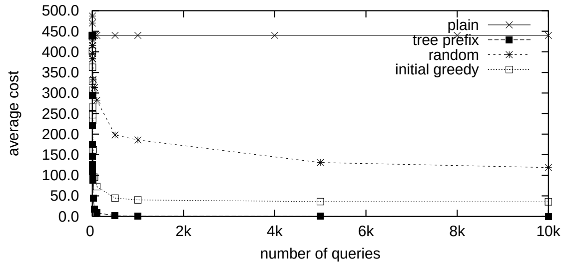
(c) RANDOM-10000-workload



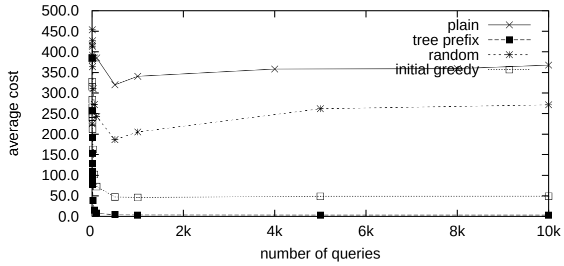
(d) DEVIATED-1x10000-workload



(e) DEVIATED-100x100-workload

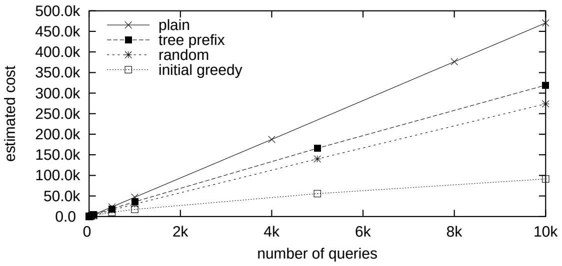


(f) REPEATED-1x10000-workload

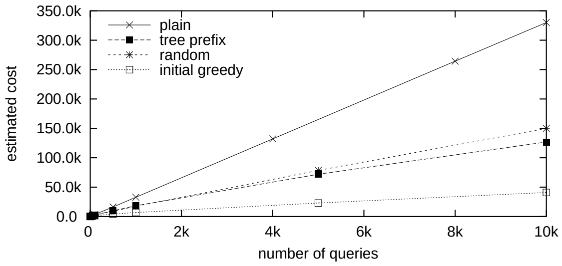


(g) REPEATED-100x100-workload

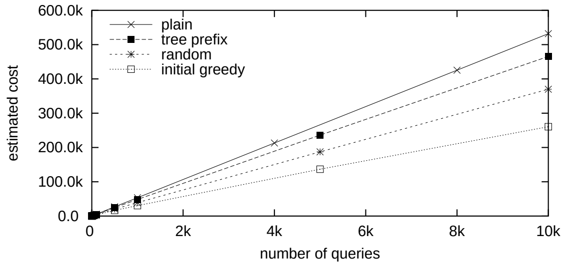
Figure 9.6: Average cost per query for the generated solutions under $\kappa_{merging}$



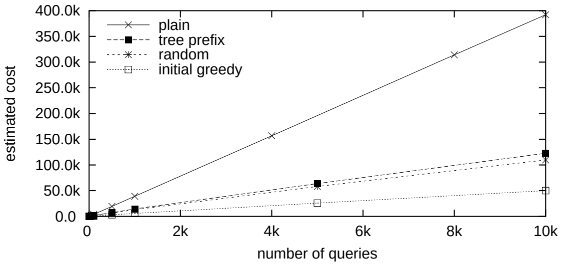
(a) NITF-10000-workload



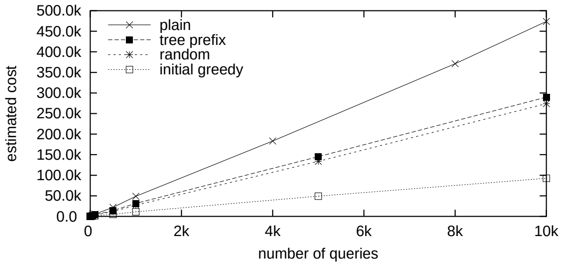
(b) COURSES-10000-workload



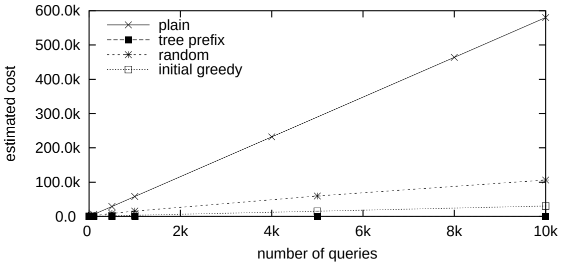
(c) RANDOM-10000-workload



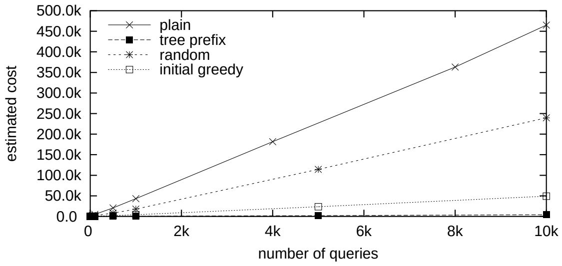
(d) DEVIATED-1x10000-workload



(e) DEVIATED-100x100-workload

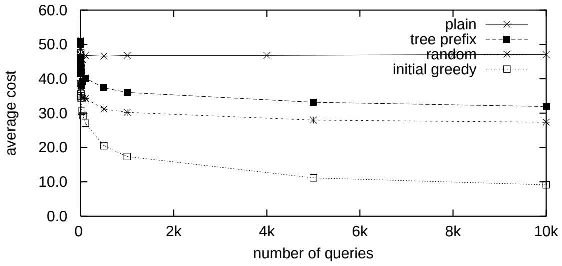


(f) REPEATED-1x10000-workload

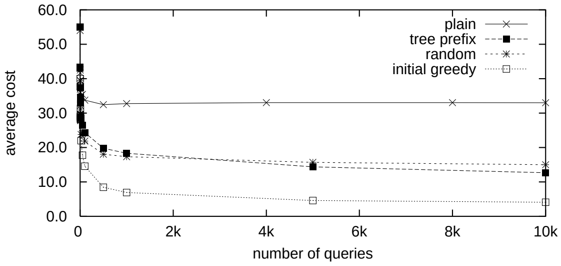


(g) REPEATED-100x100-workload

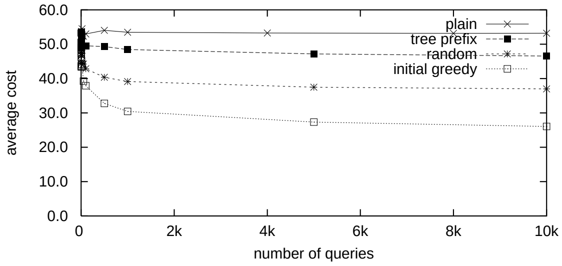
Figure 9.7: Cost of the generated solutions under κ_{edges}



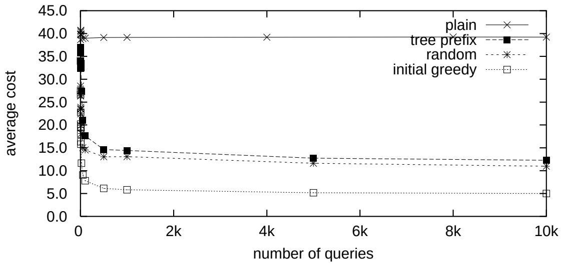
(a) NITF-10000-workload



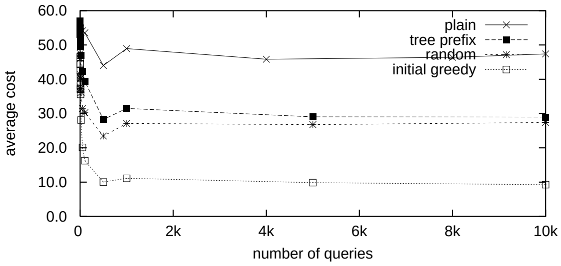
(b) COURSES-10000-workload



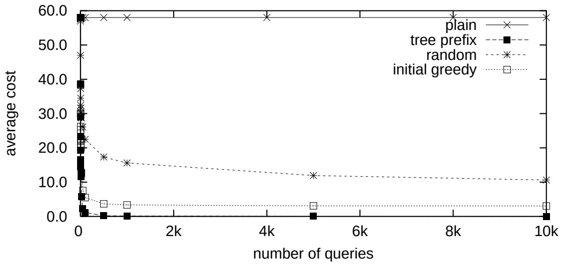
(c) RANDOM-10000-workload



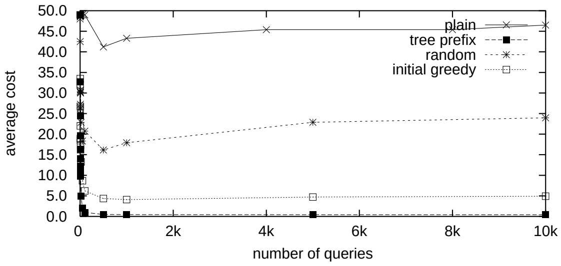
(d) DEVIATED-1x10000-workload



(e) DEVIATED-100x100-workload



(f) REPEATED-1x10000-workload



(g) REPEATED-100x100-workload

Figure 9.8: Average cost per query for the generated solutions under κ_{edges}

produced by the initial greedy pair merger and the random pair merger are always very low and very similar. A solution constructed by the random pair merger includes however vastly more edges, since the discussion which of the valid candidates for merging with a vertex is done without considering the resulting cost. But also a solution produced by the greedy pair merger contains often relatively more edges than vertices, e.g. for the NITF- or RANDOM-workload.

The tree prefix pair merger, on the other hand, does not show this asymmetry between edges and vertices. Since, whenever a vertex in a prefix can not be shared with the corresponding vertex in a prefix of the other query plan, all remaining vertices and edges alike are unshared.

9.2.2 Comparing the Time

The advantage the initial greedy pair merger has in solution cost over the other pair mergers evaluated here is offset to some extend if one considers the time for constructing a solution: Figure 9.11 and 9.13 show the absolute time for constructing a solution versus the number of queries, Figure 9.12 and 9.14 the average time per query.

Confirming the theoretical complexities from Chapter 6, the tree prefix pair merger (and, of course, the constant plain pair merger) outperforms the two remaining incremental pair merger clearly. Except for the RANDOM-workload, the initial greedy pair merger can construct its solution in acceptable time (clearly lower than 1 s per query), in some cases, e.g. for the COURSES-workload, even nearly as fast as the tree prefix pair merger.

Interestingly, the random pair merger performs far worse than the initial greedy pair merger over all workloads. Recall, that the n query plans are compacted by the arbitrary order set merger used here, by merging the first two query plans, than merging the third query plan into the result of the first merging, and so on. Furthermore, the complexity of `validCandidates` is linear in the size of the input query plans, as discussed in Section 6.2.1. For the random pair merger the size of the intermediary results increases considerably more than for the initial greedy pair merger (as seen in the previous sections), that the slight initial advantage is offset.

9.2.3 Comparing the Results

To illustrate this experimental evaluation Figure 9.15 to 9.18 show the resulting query plans constructed by the initial greedy pair merger and the tree prefix pair merger on the COURSES- and DEVIATED-1x10000-workloads after 2, 5, and 10 query plans have been considered. The brightness of a vertex or edge indicates the fraction of query plans shared: the darker a vertex the more queries the vertex is part of. Interestingly, the initial greedy pair merger shares the almost same prefixes as the tree prefix pair merger, but also finds other interesting areas for sharing, in particular towards the end of the query plans.

Clearly, the chance of finding a query plan or part of a query plan that is very similar to a part of a query plan that is to be added to the multi-query plan increases with the number of query plans added. Therefore, these pictures give only a rough indication of the solutions for larger number of query plans.

9.3 Comparison of Local Search Pair Mergers

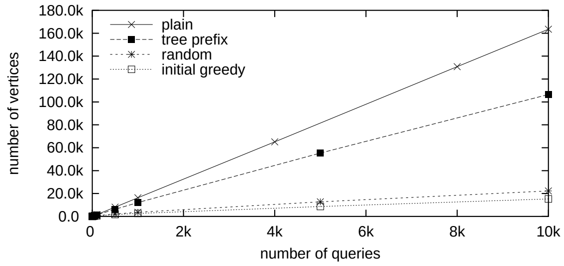
Only four of the pair mergers proposed in Chapter 6 have been evaluated in the previous section. In this section, the local search pair mergers from Section 6.2.2 are compared to the previously shown pair mergers.

For the local search mergers, the following parameters have been used: the maximum number of randomly generated solutions, i.e., number of independent tries, is MAX-TRIES $= 10$, the maximum number of improvement iterations per independent try is MAX-ITERATIONS-PER-TRY $= 15$.

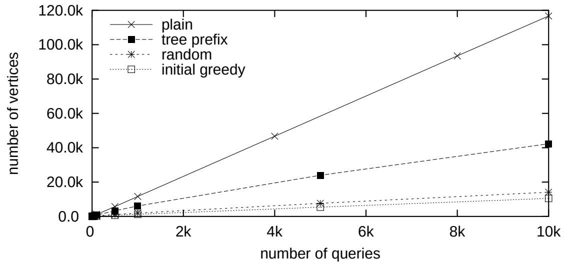
The simulated annealing algorithm has three more parameters, here chosen to be $T_{max} = 0.30$, $T_{min} = 0.01$, $c = 0.05$.

Under these settings, Figure 9.19 to 9.22 show vertices, edges, time, and average time per query for the NITF-, COURSES-, RANDOM-, and REPEATED-workload with only 100 query plans each. The cost function used is κ_{edges}.

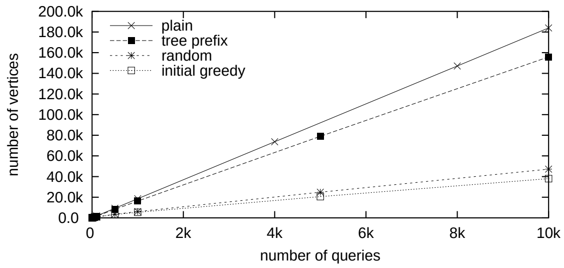
It is striking that for the kind of query plans and for the evaluation model considered here, the initial greedy pair merger not only outperforms the local search mergers with respect to the time for constructing a solution, but also with respect to the quality of



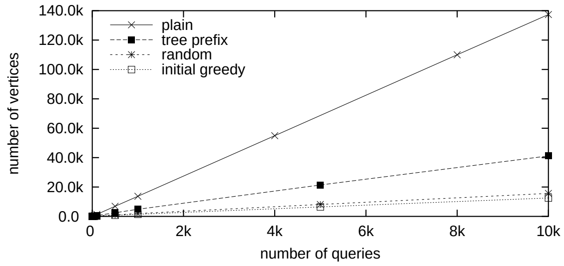
(a) NITF-10000-workload



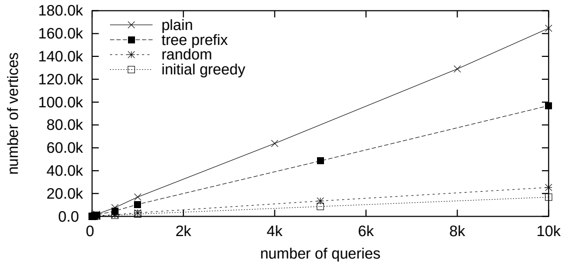
(b) COURSES-10000-workload



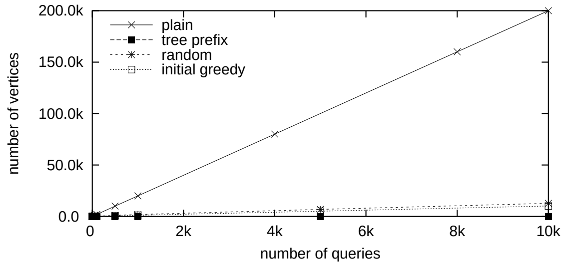
(c) RANDOM-10000-workload



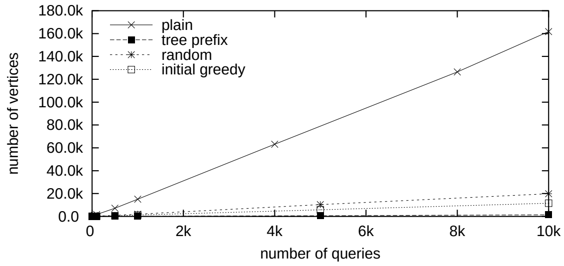
(d) DEVIATED-1x10000-workload



(e) DEVIATED-100x100-workload

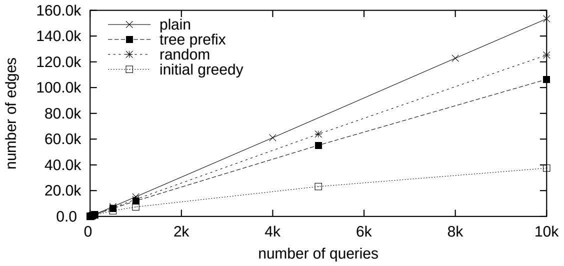


(f) REPEATED-1x10000-workload

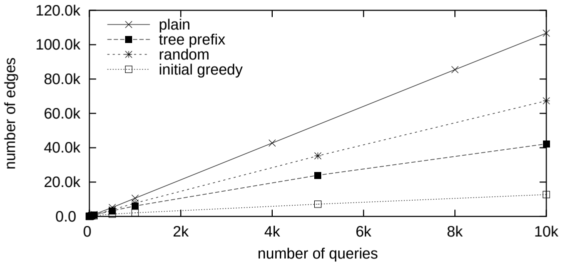


(g) REPEATED-100x100-workload

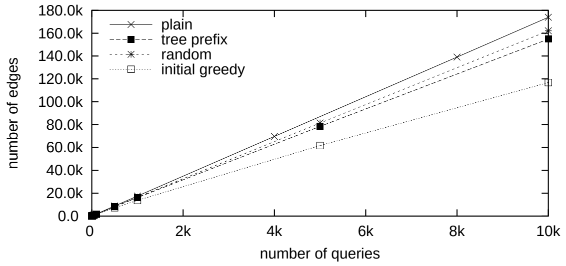
Figure 9.9: Vertices in the generated solutions under $\kappa_{merging}$



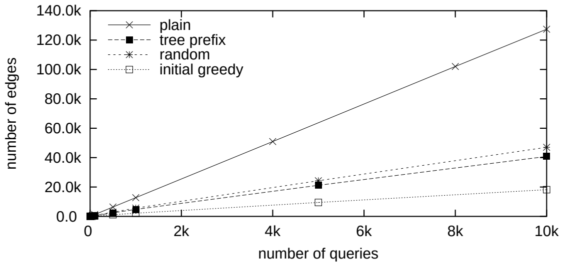
(a) NITF-10000-workload



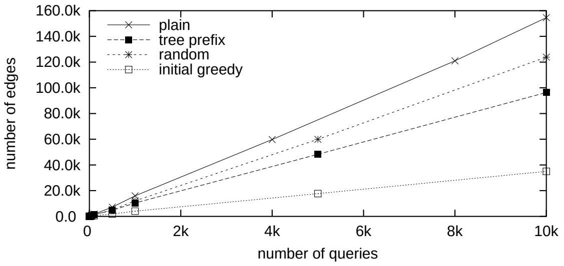
(b) COURSES-10000-workload



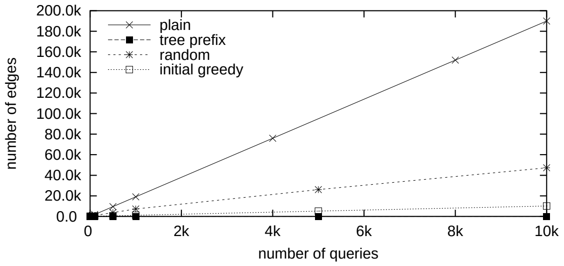
(c) RANDOM-10000-workload



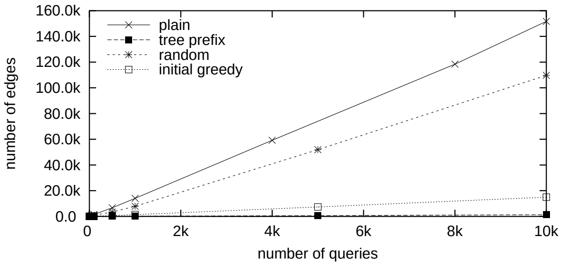
(d) DEVIATED-1x10000-workload



(e) DEVIATED-100x100-workload

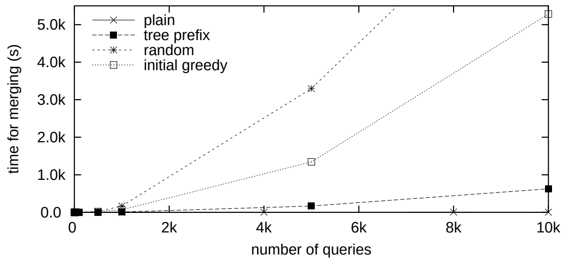


(f) REPEATED-1x10000-workload

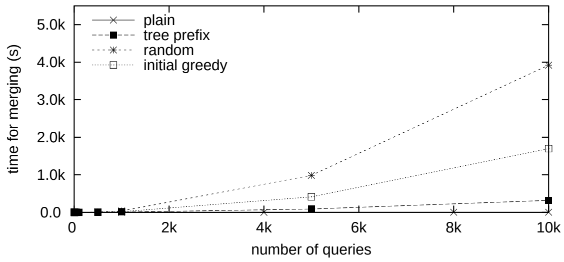


(g) REPEATED-100x100-workload

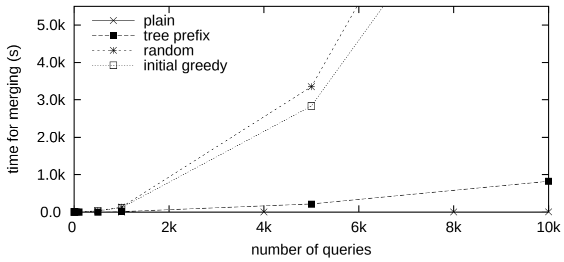
Figure 9.10: Edges in the generated solutions under $\kappa_{merging}$



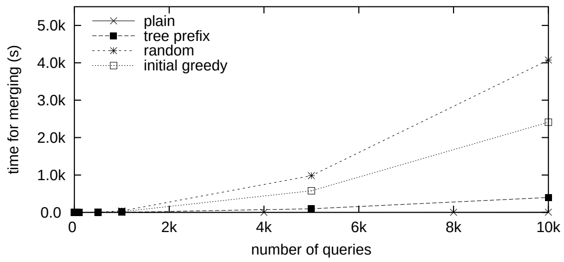
(a) NITF-10000-workload



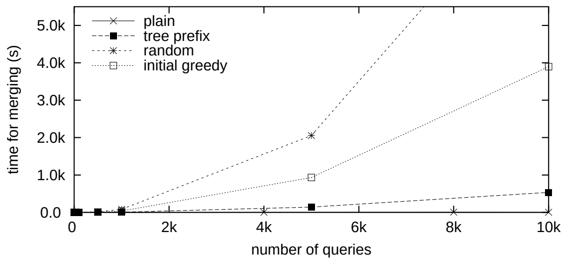
(b) COURSES-10000-workload



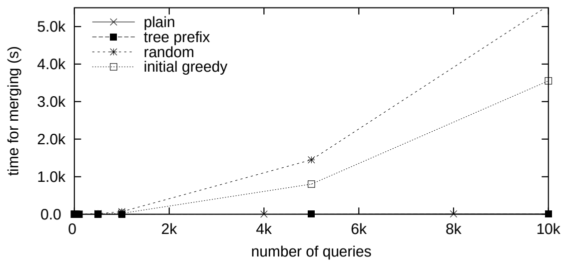
(c) RANDOM-10000-workload



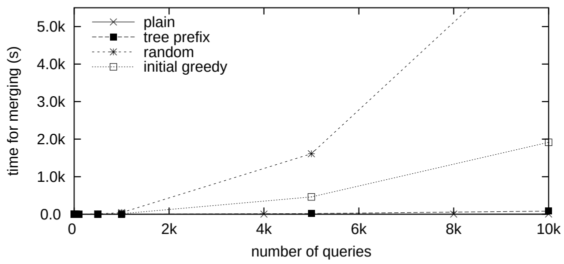
(d) DEVIATED-1x10000-workload



(e) DEVIATED-100x100-workload

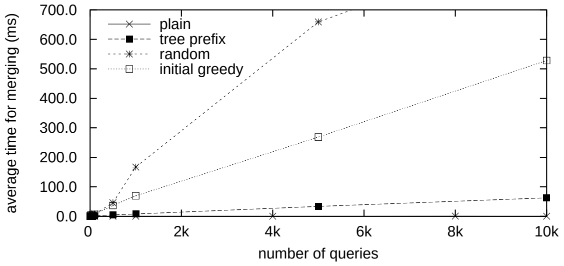


(f) REPEATED-1x10000-workload

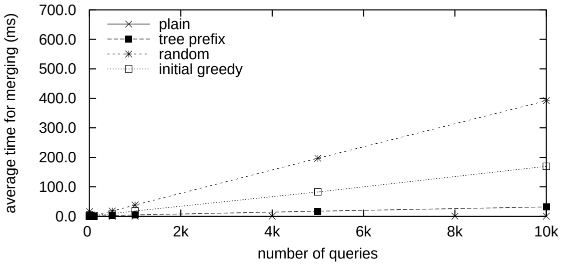


(g) REPEATED-100x100-workload

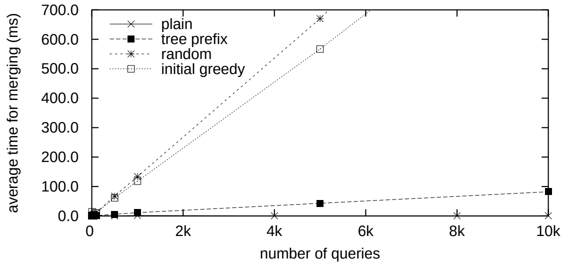
Figure 9.11: Time of the generated solutions under $\kappa_{merging}$



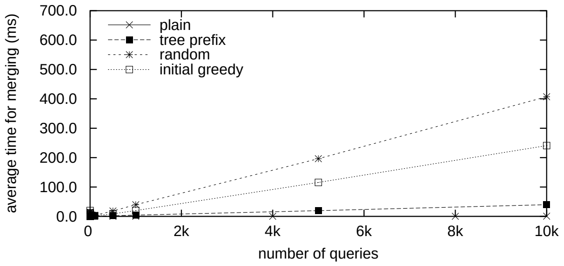
(a) NITF-10000-workload



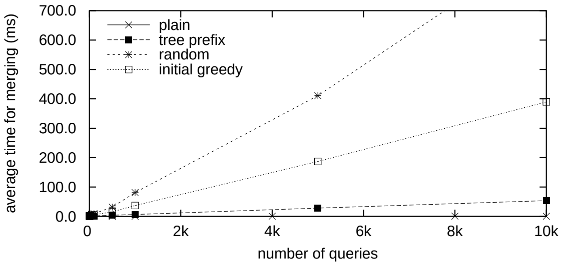
(b) COURSES-10000-workload



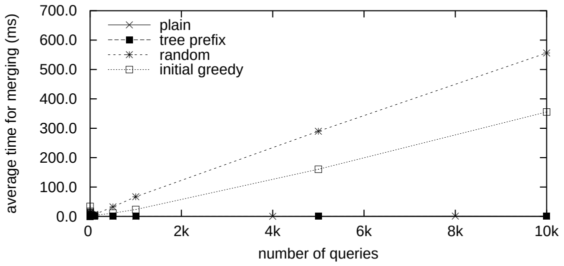
(c) RANDOM-10000-workload



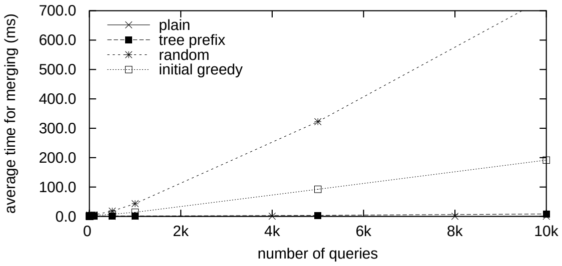
(d) DEVIATED-1x10000-workload



(e) DEVIATED-100x100-workload

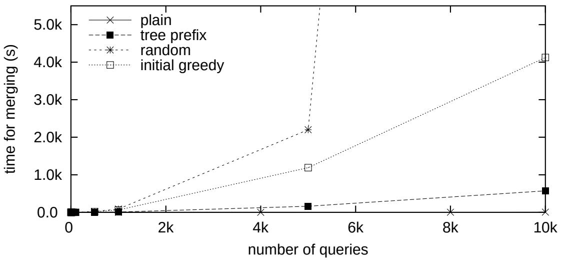


(f) REPEATED-1x10000-workload

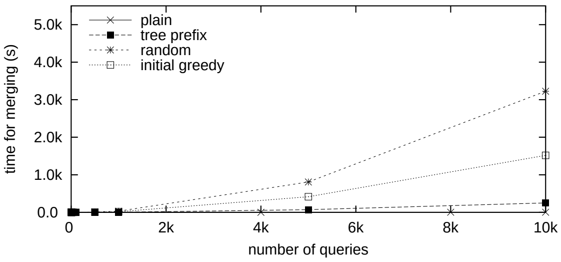


(g) REPEATED-100x100-workload

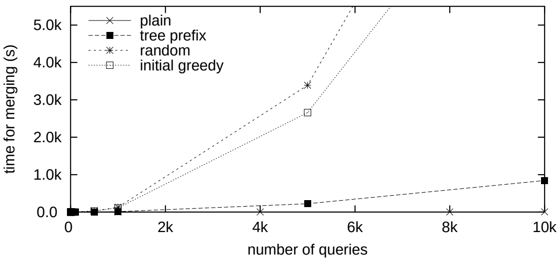
Figure 9.12: Average time per query under $\kappa_{merging}$



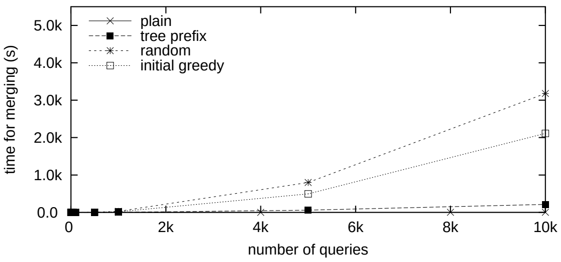
(a) NITF-10000-workload



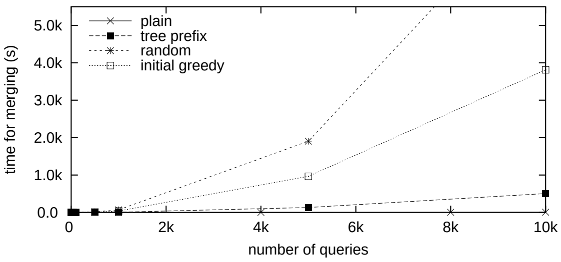
(b) COURSES-10000-workload



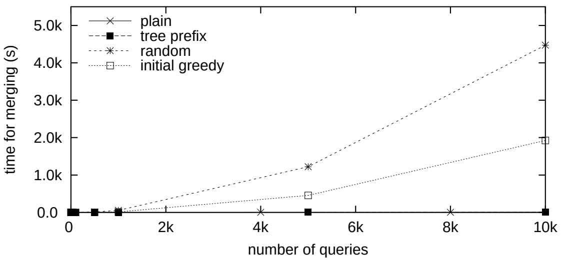
(c) RANDOM-10000-workload



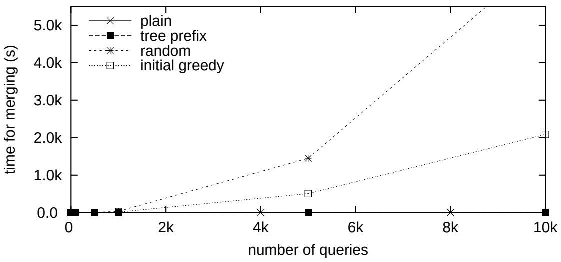
(d) DEVIATED-1x10000-workload



(e) DEVIATED-100x100-workload

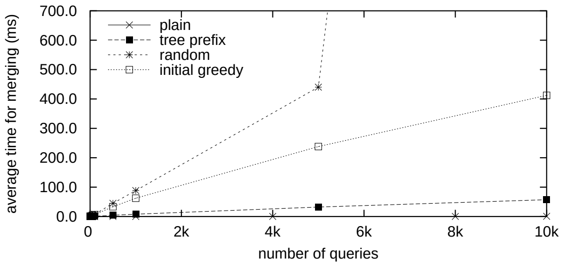


(f) REPEATED-1x10000-workload

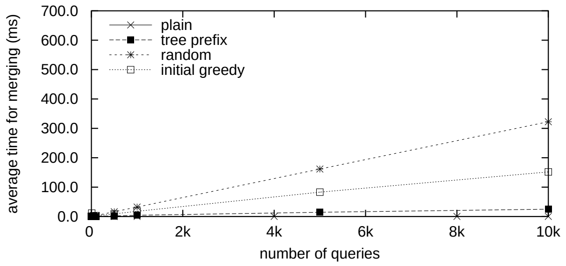


(g) REPEATED-100x100-workload

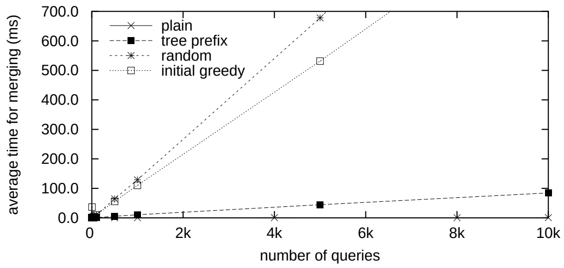
Figure 9.13: Time of the generated solutions under κ_{edges}



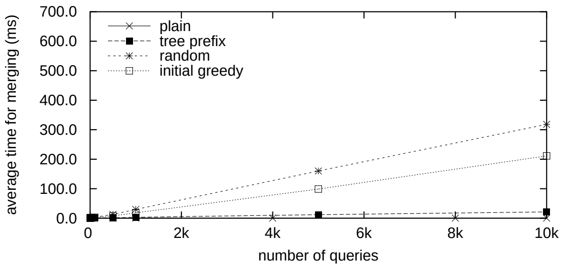
(a) NITF-10000-workload



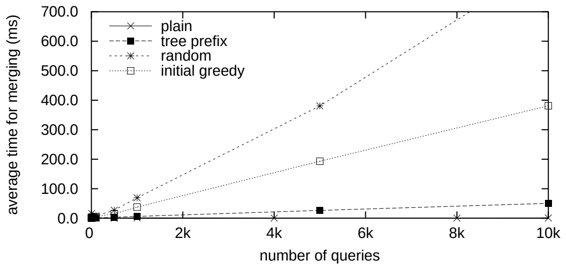
(b) COURSES-10000-workload



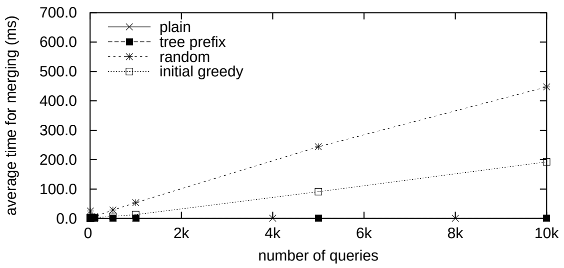
(c) RANDOM-10000-workload



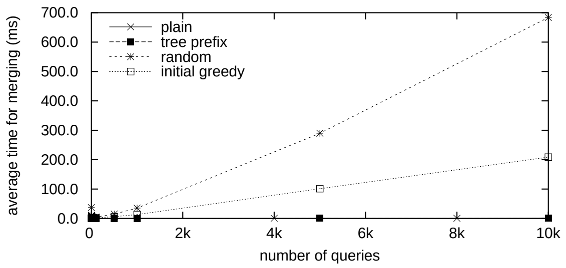
(d) DEVIATED-1x10000-workload



(e) DEVIATED-100x100-workload

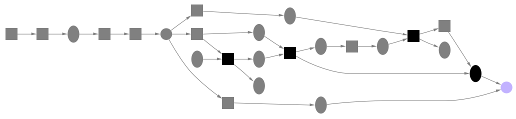


(f) REPEATED-1x10000-workload

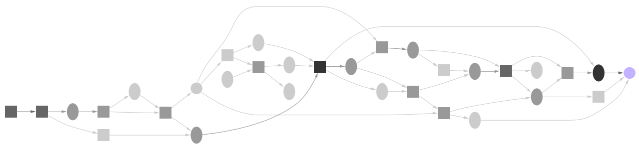


(g) REPEATED-100x100-workload

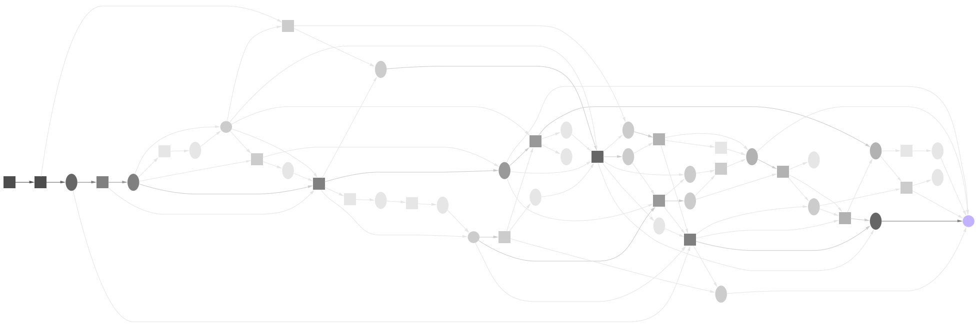
Figure 9.14: Average time per query under κ_{edges}



(a) After two query plans

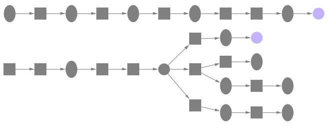


(b) After five query plans

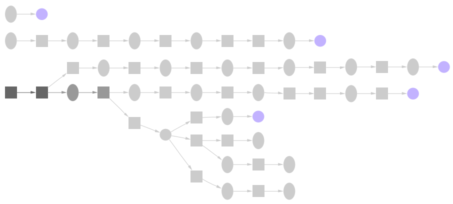


(c) After ten query plans

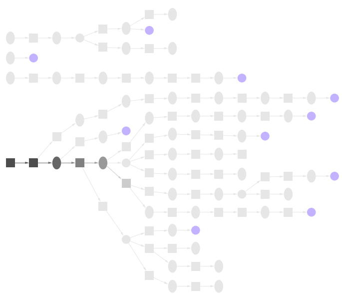
Figure 9.15: Solution for initial greedy pair merger on COURSES-workload



(a) After two query plans

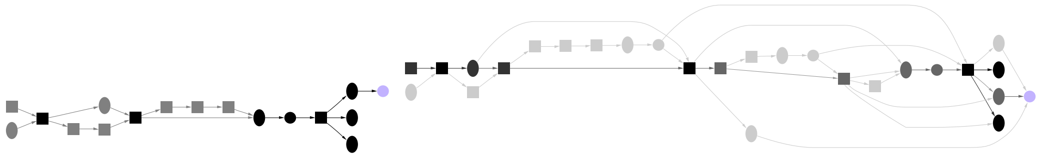


(b) After five query plans

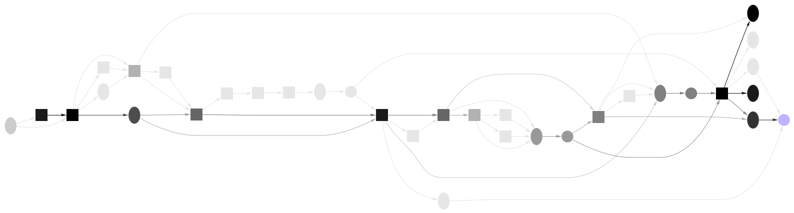


(c) After ten query plans

Figure 9.16: Solution for tree prefix pair merger on COURSES-workload

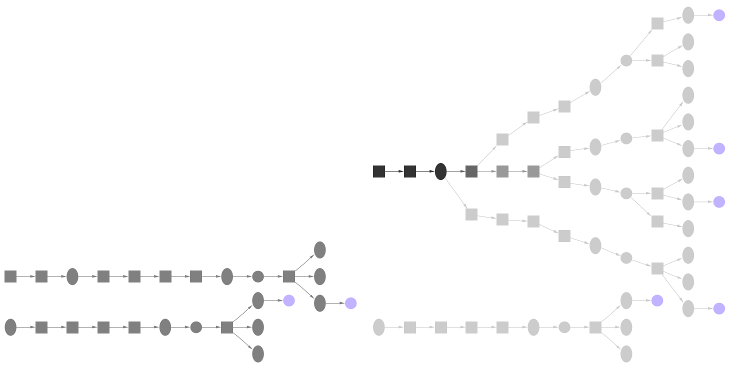


(a) After two query plans (b) After five query plans

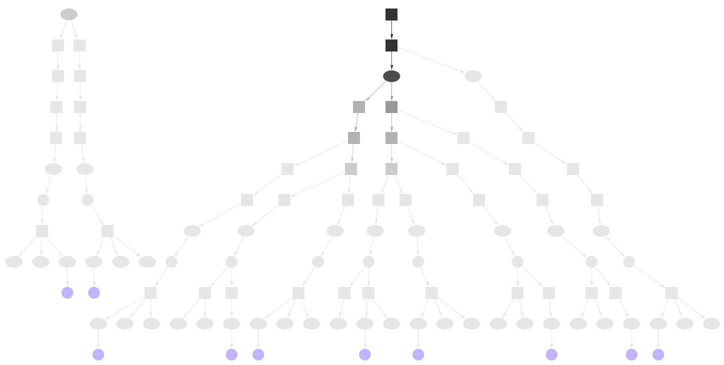


(c) After ten query plans

Figure 9.17: Solution for initial greedy pair merger on DEVIATED-1x10000-workload



(a) After two query plans (b) After five query plans



(c) After ten query plans

Figure 9.18: Solution for tree prefix pair merger on DEVIATED-1x10000-workload

that solution although in most cases only by a small margin.

Among the local search mergers, the deterministic hill-climber seems actually preferable, indicating that κ_{edges} behaves rather monotonic under this setup.

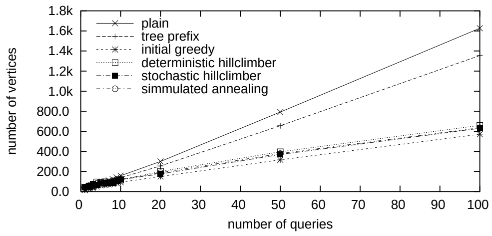
The time for constructing a solution behaves as expected from the theoretical time complexities established in Chapter 6.

9.4 Comparison of Set Mergers

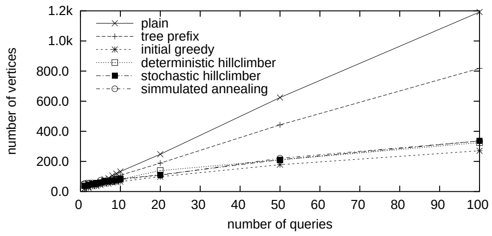
Until now only the arbitrary order set merger has been used for experimental evaluation. In this section, we will compare solution quality and time for different set mergers using the same pair merger, viz. the initial greedy pair merger.

The results shown in Figure 9.23 through 9.26 indicate that the order of mergings in this setup is not affecting the quality of the generated solution at all. Figure 9.23 shows that the solution quality for all set mergers is almost the same.

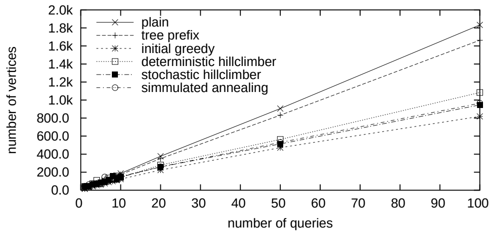
Once again, the theoretical complexities from Chapter 6 are nicely reflected in Figure 9.21 and 9.22: The arbitrary order and the initial separate order optimizer pose nearly no overhead over the pair merger and are linear in the number of queries, whereas the initial pairwise and progressive pairwise order optimizer are clearly polynomial with the second even more expensive than the first one.



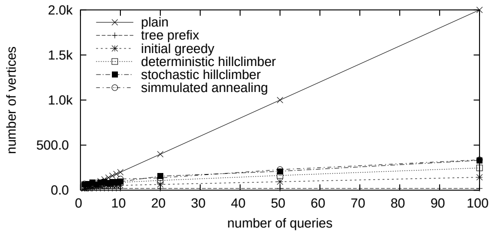
(a) NITF-100-workload



(b) COURSES-100-workload

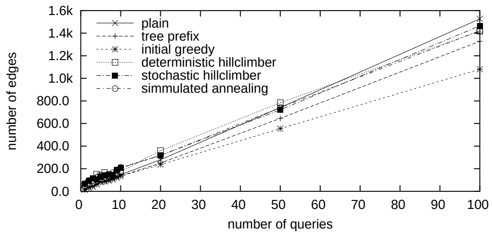


(c) RANDOM-100-workload

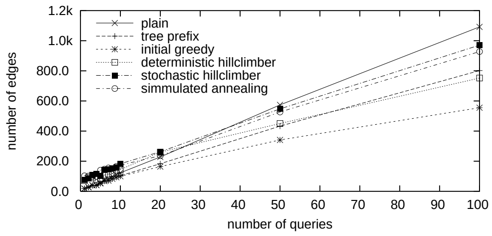


(d) REPEATED-1x100-workload

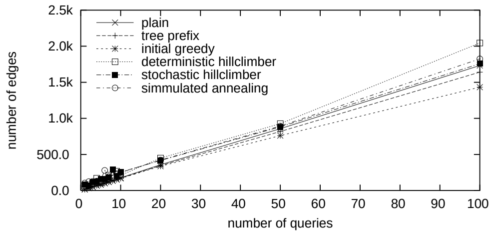
Figure 9.19: Vertices of the generated solutions under κ_{edges}



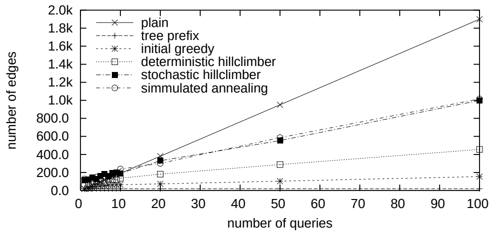
(a) NITF-100-workload



(b) COURSES-100-workload

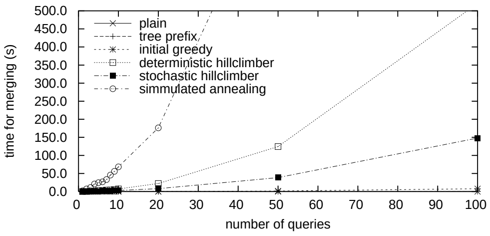


(c) RANDOM-100-workload

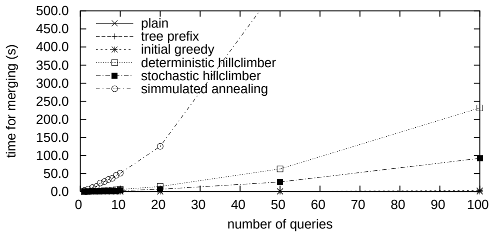


(d) REPEATED-1x100-workload

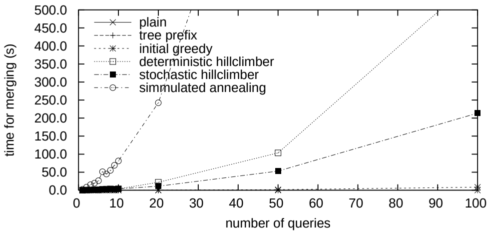
Figure 9.20: Edges of the generated solutions under κ_{edges}



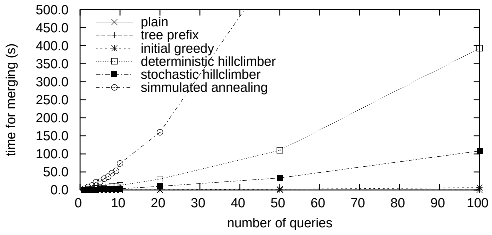
(a) NITF-100-workload



(b) COURSES-100-workload

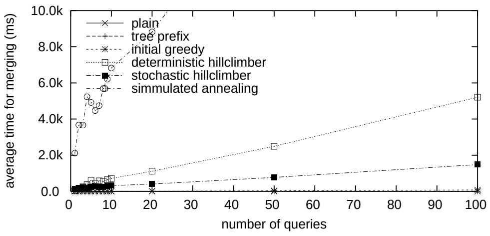


(c) RANDOM-100-workload

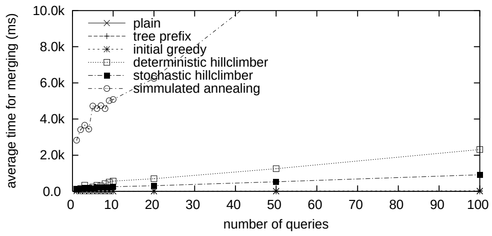


(d) REPEATED-1x100-workload

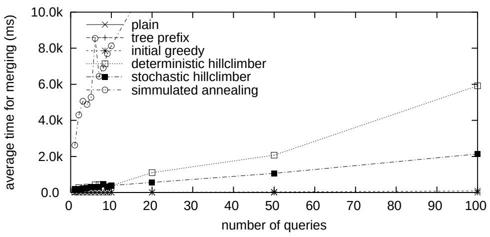
Figure 9.21: Time of the generated solutions under κ_{edges}



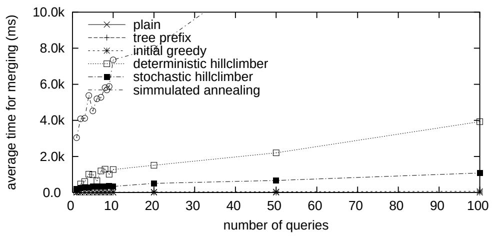
(a) NITF-100-workload



(b) COURSES-100-workload

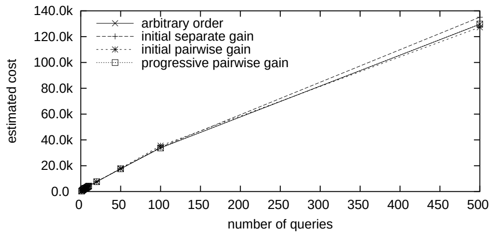


(c) RANDOM-100-workload

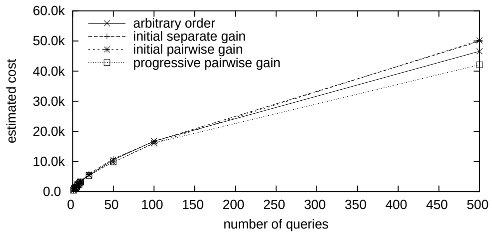


(d) REPEATED-1x100-workload

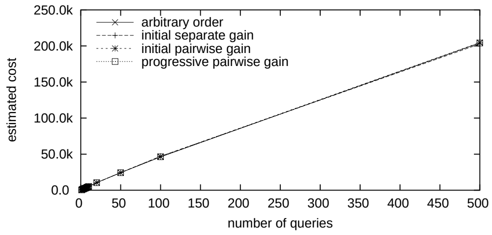
Figure 9.22: Time-Per-Query of the generated solutions under κ_{edges}



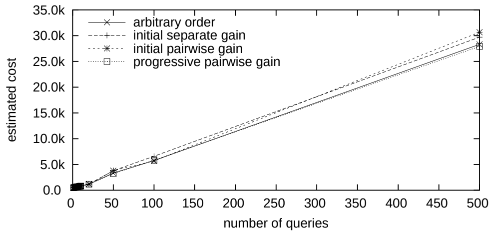
(a) NITF-500-workload



(b) COURSES-500-workload

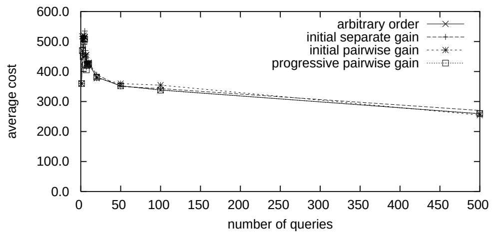


(c) RANDOM-500-workload

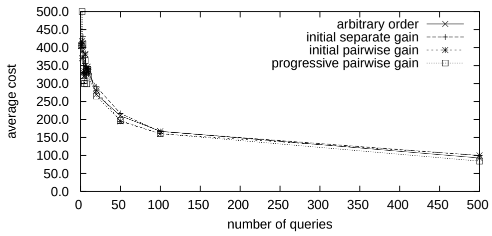


(d) REPEATED-20x25-workload

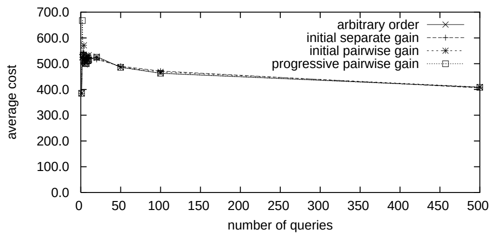
Figure 9.23: Cost of the generated solutions under $\kappa_{merging}$



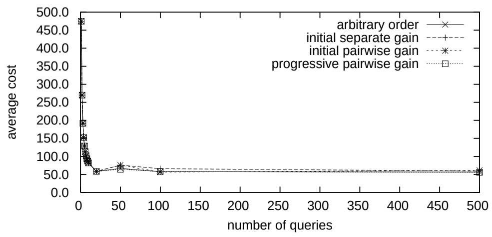
(a) NITF-500-workload



(b) COURSES-500-workload

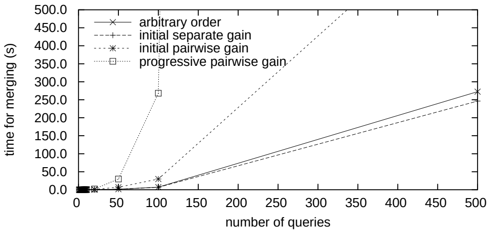


(c) RANDOM-500-workload

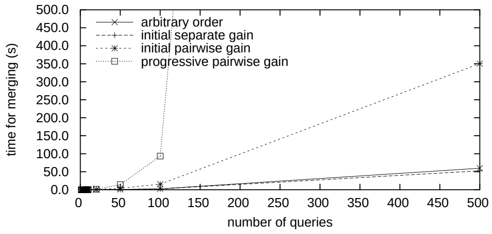


(d) REPEATED-20x25-workload

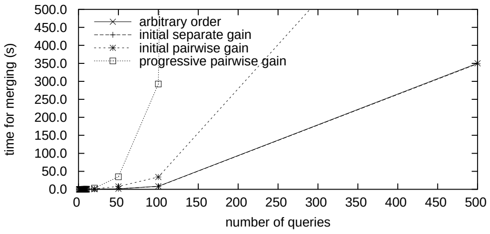
Figure 9.24: Cost-Per-Query of the generated solutions under $\kappa_{merging}$



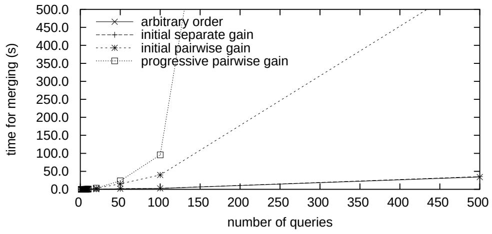
(a) NITF-500-workload



(b) COURSES-500-workload

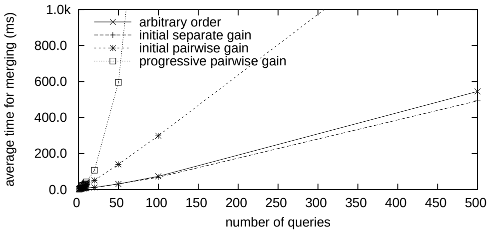


(c) RANDOM-500-workload

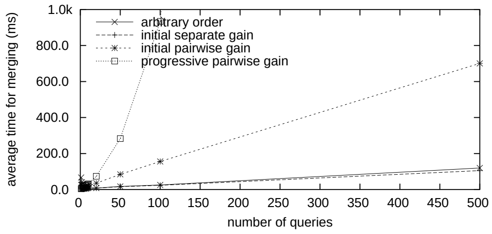


(d) REPEATED-20x25-workload

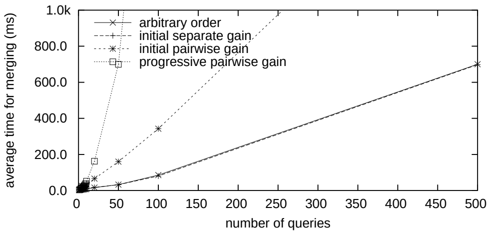
Figure 9.25: Time of the generated solutions under $\kappa_{merging}$



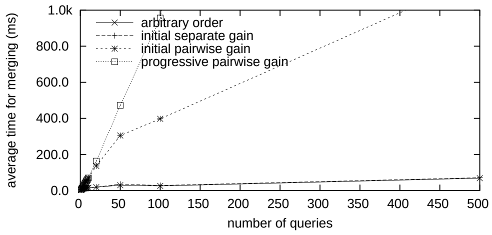
(a) NITF-500-workload



(b) COURSES-500-workload



(c) RANDOM-500-workload



(d) REPEATED-20x25-workload

Figure 9.26: Time-Per-Query of the generated solutions under $\kappa_{merging}$

Chapter 10

Implementation

In this chapter, a brief overview over the implementation of the method proposed in this thesis is given. The overview concentrates on the design of the optimization framework. A more extensive documentation of the implementation is available in form of an documentation of the API [50].

Contents

Based on the prototype of the SPEX engine presented in [79], the optimization framework proposed in this work has been implemented. As implementation language, Java [74] has been selected, in particular to ease the migration from the previous SPEX version. All pair and set mergers proposed in Chapter 6 have been implemented (cf. Sections 10.3 and 10.4) on top of the optimization framework. A new graph library (cf. Section 10.1) optimized for efficient iteration over the edges of a vertex is employed to implement the query plans for SPEX (cf. Section 10.2) and as well as numerous algorithms that are part of the optimization framework (cf. Section 10.5). Finally, several tools for automated testing and performance measurement have been developed (cf. Section 10.6).

[79] Kiesling, T. 2002. Towards a streamed XPath evaluation. M.S. thesis, University of Munich, Institute of Computer Science.

[74] Joy, B., et al. 2000. *The Java Language Specification*, 2nd ed. Addison-Wesley.

10.1 Basic Graph Library: `spex.util.graph`

The core of the implementation, both with respect to the class hierarchy and to performance, is the graph library. From the set and pair merger algorithms and a review of the shape of query plans that are to be optimized based on the graph library, analytical requirements for the graph library can be obtained:

(1) Fast iteration over the edges incident to a vertex is needed for both for the `merge` function as well as for the local cost functions.

(2) `merge` furthermore requires a (almost) constant-time test whether two given vertices are adjacent.

(3) The degree of most vertices is usually very small, in almost every case clearly smaller than the number of vertices. Therefore, an implementation with $O(\mathsf{V} \times \mathsf{E})$ space complexity such as an adjacent list is preferable to an implementation with $O(\mathsf{V}^2)$ space complexity such as an adjacent matrix.

(4) Several algorithms require the ability to attach arbitrary pieces of information to a vertex or edge.

(5) Efficient access to the ancestors and descendants of a vertex.

Several existing graph libraries for Java have been reviewed along this criteria, but none of them could satisfy the given constraints. Most of the graph libraries for Java are tailored to graph visualization [66]. This results not only in relatively heavy-weight libraries introducing a considerable overhead into the optimization framework, but also in their failure to meet most of the criteria presented above as they have other focus, as described in [91].

Therefore, a specialized graph library has been implemented based on the idea of an adjacent list due to the third observation. But instead of a simple list, an associative storage or Map is used that uses vertices as keys and edges as entries. Thereby, the test whether two vertices are adjacent is in most cases constant. To allow fast iteration over the edges, the edges are furthermore linked like in a linked list. The JDK recently introduced such a Map as part of the collections framework, called there LinkedHashMap, as it is based on a hash as associative storage.

Extensive profiling of this implementation has shown however, that (1) iteration over the elements in a LinkedHashMap is still very inefficient compared to iterating over the elements of a list such as LinkedList or array and (2) the test whether two given vertices are adjacent has far less influence on the overall run-time than the iteration. Therefore, a second implementation of the graph library based on LinkedLists for storing the edges incident to a vertex has been provided. It turned out, that switching from the original implementation to this second implementation improved the time for the more expensive algorithms by up to 75%.

Figure 10.5 shows the hierarchy of the most important classes and interfaces that are part of the (second version of the) graph library, implemented as package spex.util.graph.

Three interfaces are at the center of the graph library: the *DirectedAcyclicGraph*, *Vertex*, and *Edge* interface, that represent a DAG, a vertex in a DAG and a directed edge.

```
DirectedAcyclicGraph
----------------------------------------------------------------
+ removeVertex(v : Vertex) : boolean
+ removeVertex(v : Vertex, plain : boolean) : boolean
+ addEdge(e : Edge)
+ addEdge(source : Vertex, sink : Vertex)
+ removeEdge(e : Edge) : boolean
+ removeEdge(source : Vertex, sink : Vertex) : boolean
+ getVertices() : ListIterator
+ areAdjacent(v : Vertex, w : Vertex) : boolean
+ areAdjacent(e : Edge, f : Edge) : boolean
+ numVertices() : int
+ numEdges() : int
+ degree(v : Vertex) : int
+ degree() : int
+ isEmpty() : boolean
+ containsVertex(v : Vertex) : boolean
+ containsEdge(source : Vertex, sink : Vertex) : boolean
+ getVertexFactory() : VertexFactory
+ getEdgeFactory() : EdgeFactory
+ deepClone() : DirectedAcyclicGraph
+ shallowClone() : DirectedAcyclicGraph
+ resetProcessedFlag()
+ getVerticesList() : List
+ isIsomorphic(other : DirectedAcyclicGraph) : boolean
+ getRoots() : List
+ getLeaves() : List
+ getRootsIterator() : Iterator
+ getLeavesIterator() : Iterator
```

Figure 10.1: Interface *DirectedAcyclicGraph*

The *DirectedAcyclicGraph* interface provides access to the vertices in the graph and several graph related functions, such as test for graph isomorphism and cloning of graphs. All edge related functions are convenient wrappers for the corresponding functions of the vertex interface. The *Vertex* interface is the most extensive interface in the optimization framework and allows the manipulation of a vertex and its incident edges by a plethora of operations as shown in Figure 10.2. Most notably, the edges of a vertex can be traversed by an ListIterator.

The *Edge* is simple in comparison to the *Vertex* interface, since it provides essentially only access to the vertices adjacent to it and a convenient function for testing whether adding that edge is part of a cyclic path. The latter function is implemented using established methods for dynamic cycle detection [125].

All these interfaces are inherited from *Decorable*, a software pattern used to allow arbitrary attribute-value pairs to be associated with an object, as required by the specifications for the graph library discussed above.

[66] Herman, I., et al. 2000. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics 6,* 1, 24–43.

[91] Marshall, M. S., et al. 2001. An object-oriented design for graph visualization. *Software Practice and Experience 31,* 8, 739–756.

[125] Shmueli, O. 1983. Dynamic cycle detection. *Information Processing Letters 17,* 4, 185–188.

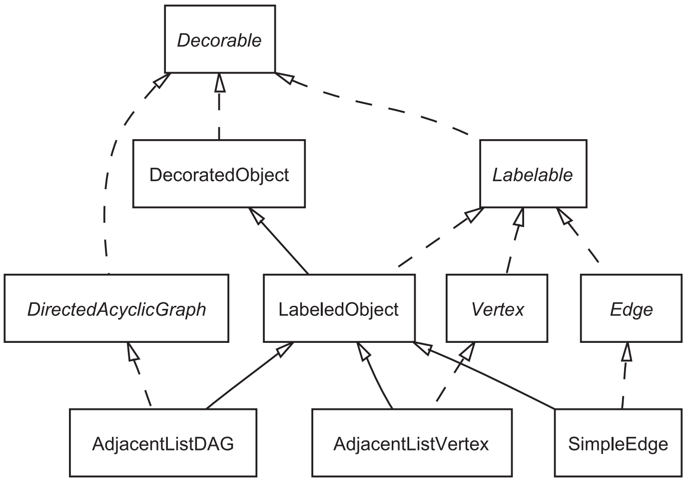


Figure 10.5: Hierarchy of important classes and interfaces in spex.util.graph

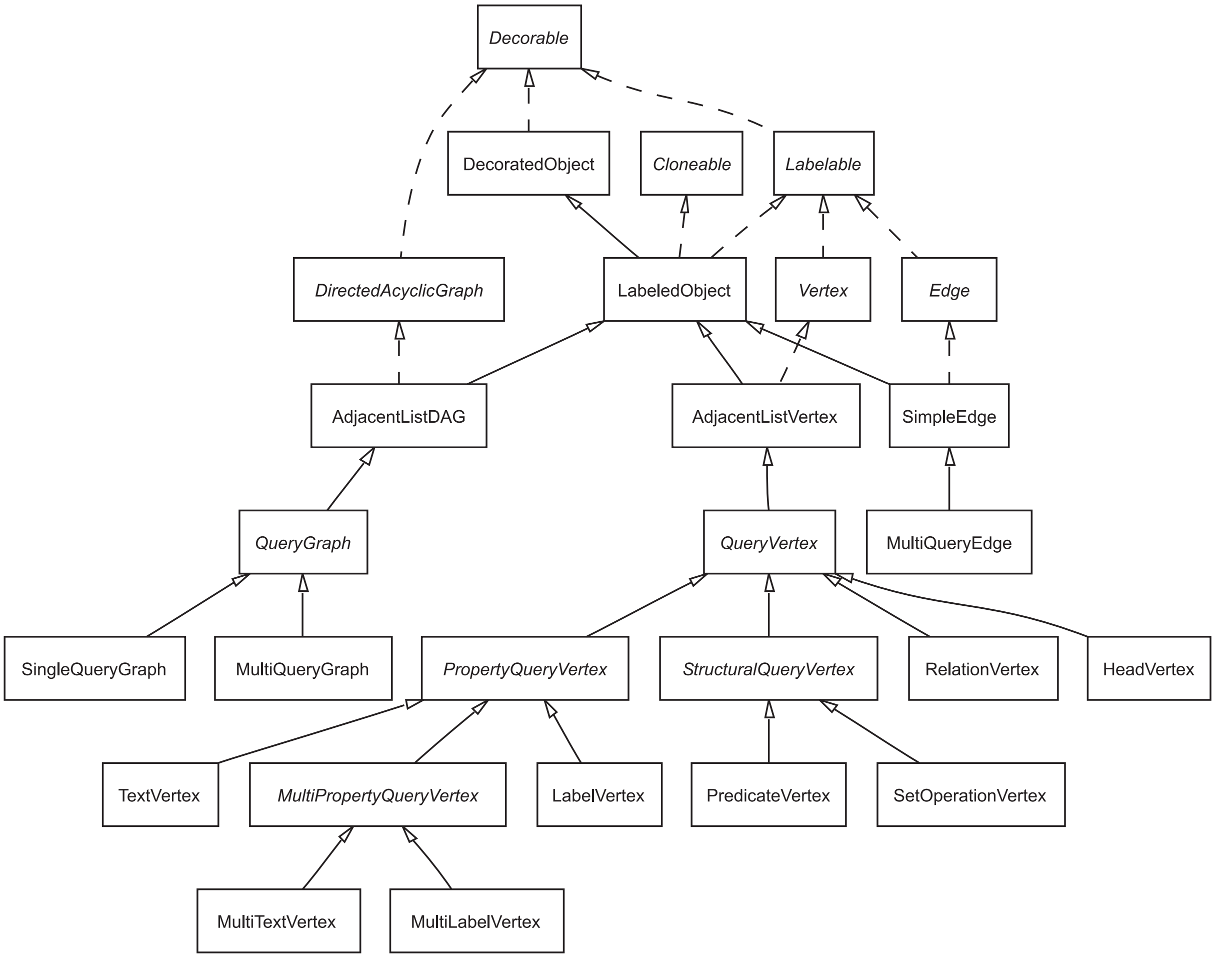


Figure 10.6: Hierarchy of important classes and interfaces in spex.queries

Vertex
+ addEdgeTo(v : Vertex)
+ addEdgeFrom(v : Vertex)
+ addEdge(e : Edge, plain : boolean)
+ addEdgeTo(v : Vertex, plain : boolean)
+ addEdgeFrom(v : Vertex, plain : boolean)
+ addEdge(e : Edge)
+ removeEdge(e : Edge) : boolean
+ removeEdgeTo(v : Vertex) : boolean
+ removeEdgeFrom(v : Vertex) : boolean
+ removeEdge(e : Edge, plain : boolean) : boolean
+ removeEdgeTo(v : Vertex, plain : boolean) : boolean
+ removeEdgeFrom(v : Vertex, plain : boolean) : boolean
+ getOutgoingEdges() : ListIterator
+ getIncomingEdges() : ListIterator
+ getParents() : ListIterator
+ getChildren() : ListIterator
+ getAdjacentVertices() : ListIterator
+ getIncidentEdges() : ListIterator
+ isAdjacent(v : Vertex) : boolean
+ hasParent(v : Vertex) : boolean
+ hasChild(v : Vertex) : boolean
+ existsPathTo(v : Vertex) : boolean
+ createsCycle(v : Vertex) : boolean
+ getEdgeTo(v : Vertex) : Edge
+ getEdgeFrom(v : Vertex) : Edge
+ degree() : int
+ numOutgoingEdges() : int
+ numIncomingEdges() : int
+ getGraph() : DirectedAcyclicGraph
+ getPathFrom(v : Vertex) : LinkedList
+ getPathTo(v : Vertex) : LinkedList
+ shallowClone(graph : DirectedAcyclicGraph) : Vertex
+ deepClone(target : DirectedAcyclicGraph) : Vertex
+ getDescendantsTraversal(order : char) : TraversalIterator
+ getAncestorsTraversal(order : char) : TraversalIterator
+ getDescendantsTraversal(order : char, f : Filter) : TraversalIterator
+ getAncestorsTraversal(order : char, f : Filter) : TraversalIterator
+ getDescendantsTraversal() : TraversalIterator
+ getAncestorsTraversal() : TraversalIterator
+ getDescendantsTraversal(f : Filter) : TraversalIterator
+ getAncestorsTraversal(f : Filter) : TraversalIterator
+ getChildrenTraversal(f : Filter) : TraversalIterator
+ getParentsTraversal(f : Filter) : TraversalIterator
+ getChildrenTraversal() : TraversalIterator
+ getParentsTraversal() : TraversalIterator
+ getChild() : Vertex
+ getParent() : Vertex
+ getParent(f : Filter) : Vertex
+ getChild(f : Filter) : Vertex
+ hasIdenticalType(other : Vertex) : boolean
+ isIsomorphic(vm : Vertex) : boolean
+ includes(vm : Vertex) : boolean
+ isIncludedIn(vm : Vertex) : boolean
+ isProcessed() : boolean
+ getProcessedValue() : Object
+ setProcessed()
+ setProcessed(value : Object)
+ clearProcessed()

Figure 10.2: Interface *Vertex*

Edge
+ getSourceVertex() : Vertex
+ getSinkVertex() : Vertex
+ getOppositeVertex(v : Vertex) : Vertex
+ isIncident(v : Vertex) : boolean
+ createsCycle() : boolean
+ clone(source : Vertex, sink : Vertex) : Edge

Figure 10.3: Interface *Edge*

Decorable
+ remove(key : Object) : Object
+ get(key : Object) : Object
+ has(key : Object) : boolean
+ set(key : Object, value : Object)
+ setAttributes(attributes : HashMap)

Figure 10.4: Interface *Decorable*

10.2 From Graphs to Query Plans: `spex.queries`

Based on this graph library, query plans for the SPEX engine are implemented. Figure 10.6 shows the full hierarchy of classes and interfaces in the `spex.queries` package. This hierarchy can be divided into classes and interfaces that are used to represent query plans or query graphs, as they are called in the implementation, vertices that occur in query plans, and edges that occur in query plans.

Once again, the case of the edges is the simplest: The interface *Edge* is implemented by SimpleEdge that is part of the graph library. Instances of SimpleEdge are used in general graphs but also in query plans as long as the query plan evaluates a single query only. The class MultiQueryEdge extends SimpleEdge by several operations for handling the queries assigned to an edge in a query plan (as represented by the q function in the formal specification of a query plan).

Vertices occurring in query graphs are classified by the operator they are assigned to. For each operator, there is a corresponding class as shown in Figure 10.7. The *out* operator is represented by the HeadVertex and the vertices for property operators are further divided in vertices that can have a single property assigned to them and vertices that can carry multiple properties. The QueryVertex class shown in Figure 10.8 extends the *AdjacentListVertex* of the *Vertex* interface provided as part of the graph library by operations specific to a vertex in a query, such as operations for accessing the queries a vertex is part of. Furthermore, means to store the cost of a vertex are provided, allowing efficient implementation of the vertex-based cost functions discussed in Chapter 8 using memorization.

Finally, for efficiency reasons, there are two kinds of query plans (called query graphs): query plans that evaluate a single query only as instances of the class SingleQueryGraph, and query plans that evaluate multiple queries as instances of the MultiQueryGraph that provide additional methods for accessing the queries evaluated by a query plan. Furthermore, MultiQueryGraph allows efficient access to its vertices by the operator they are assigned to. This operation is crucial for implementing `validCandidates` in the AbstractMerger (cf. Section 10.3). Finally, efficient access to the mappings from vertices of one MultiQueryGraph to another is provided as support for the pair mergers

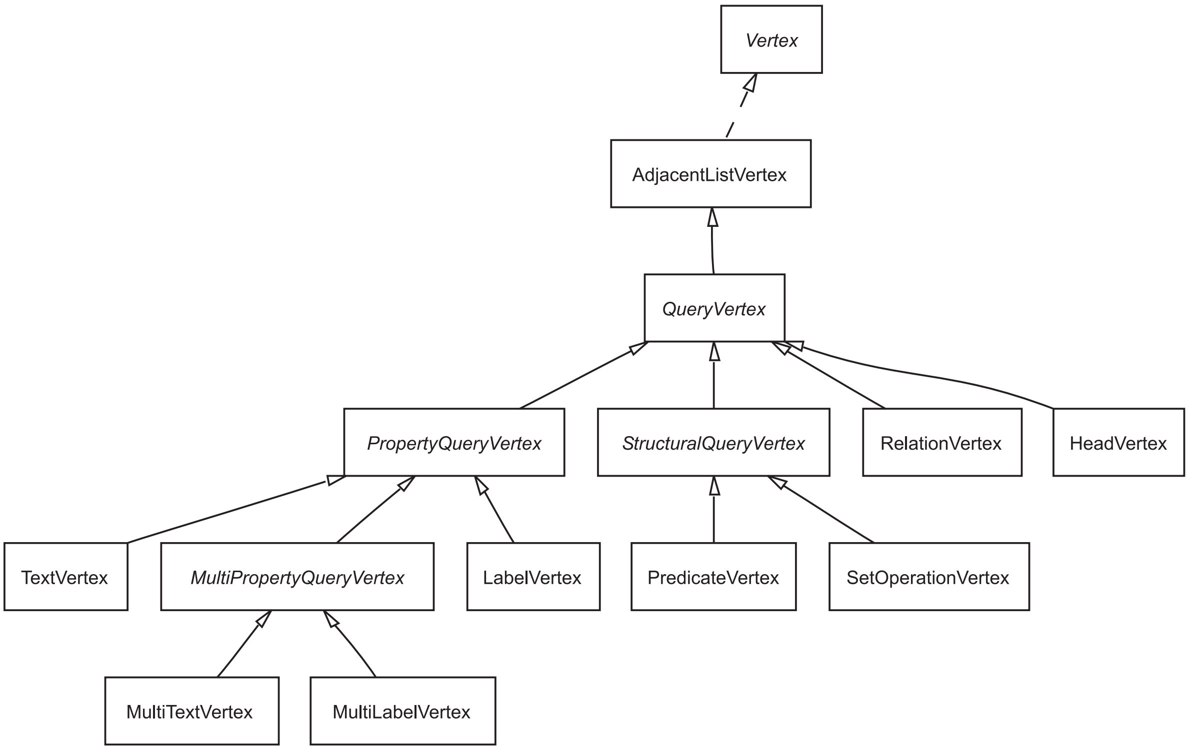


Figure 10.7: Class hierarchy for query vertices

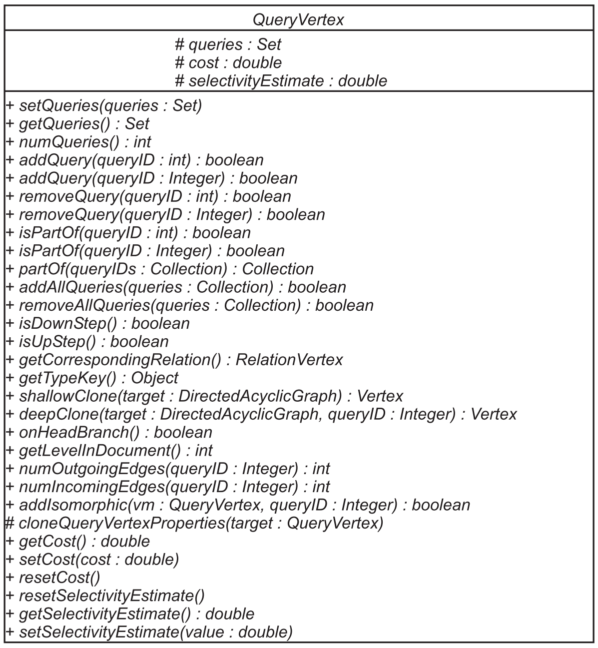


Figure 10.8: Class QueryVertex

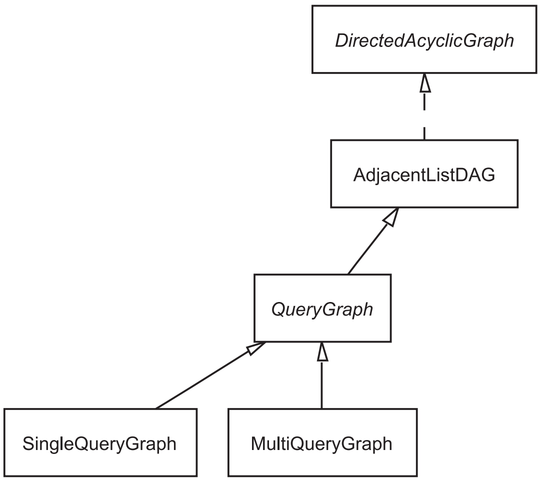


Figure 10.9: Class hierarchy for query graphs

shown in Section 10.3 as well as for the pairwise set merger.

10.2.1 Computing the cost of a query plan: `spex.queries.optimizers.cost`

All vertex-based cost functions proposed in Chapter 8 are implemented as shown in Figure 10.10 based on the common interface *CostFunction* providing means for computing the cost of a graph or a vertex. All these implementations are required to provide means to update the cost of a graph upon changes to that graph. Thus, if e.g. a vertex is added to the graph, an independent cost function can simply increase the cost of the graph by the cost of the new vertex. For a local cost function, also all now adjacent vertices of the new vertex might change their cost and therefore have to be considered. In the case of a global cost function, all vertices might actually be affected. Nevertheless, this optimization together with the memorization of the cost of a vertex allow very efficient implementations of the cost functions, in particular of the independent and local cost function.

The three cost functions $\kappa_{operators}$, implemented by the class ProcessingCostFunction, $\kappa_{merging}$, implemented by MergingCostFunction, and $\kappa_{selectivity}$, implemented by SelectivityCostFunction, can be configured with an instance of the class EvaluatorCharacteristics that describes the mapping from operator-property pairs to relative costs discussed in Chapter 8.

10.3 Pair mergers: `spex.queries.optimizers.merger`

All pair mergers proposed in Chapter 6 have been implemented including the various variants. Figure 10.11 shows the hierarchy of the corresponding classes and interfaces. The incremental mergers are mostly implemented slightly more efficient than shown in Chapter 6, but without considerable change to the worst-case time complexity. The two variants of the greedy pair merger are implemented in the GreedyIncrementalMerger that can be configured by the variant to select.

The central interface of this package is the *Merger*, that is implemented by the abstract class AbstractMerger, as shown in Figure 10.12, that is in turn extended by all the concrete pair mergers. The *Merger* interface provides the single method `merge` with two query plans as input that returns a MultiQueryGraph representing the result of merging the two input plans. The AbstractMerger contains implementations of operations used by all mergers, such as `validCandidates` implemented by `getValidMergeCandidates` or operations for adding and removing a merging from one query plan into another.

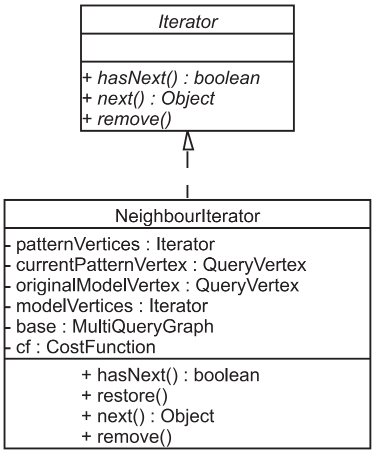


Figure 10.13: Class NeighbourIterator

Another important class is the NeighbourIterator that provides efficient iteration over the neighbors of a solution. This iteration is implemented incrementally, i.e., on each call to the `next` method the next neighbor is generated on-the-fly thus decreasing the space complexity dramatically.

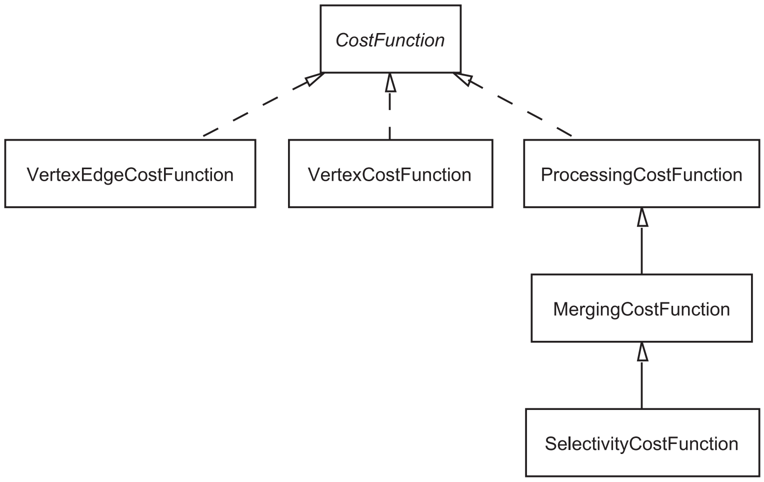


Figure 10.10: Hierarchy of important classes and interfaces in spex.queries.optimizers.cost

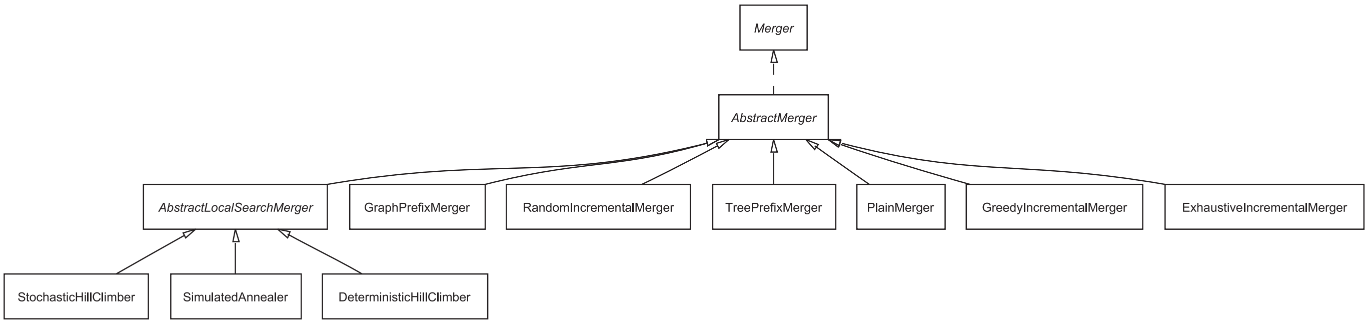


Figure 10.11: Hierarchy of important classes and interfaces in spex.queries.optimizers.mergers

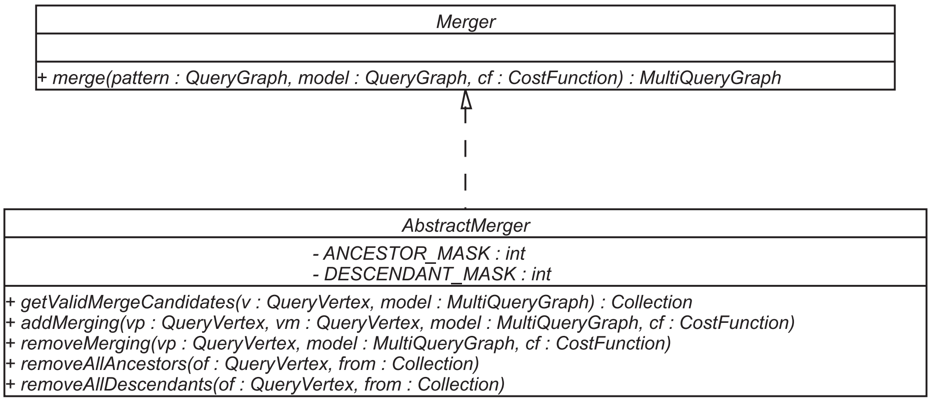


Figure 10.12: Interface *Merger*

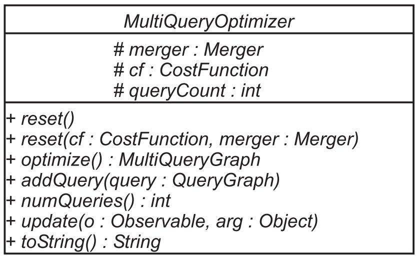


Figure 10.15: Interface *Optimizer*

10.4 Set mergers: `spex.queries.optimizers`

As the pair mergers, also all set mergers discussed in Section 6.3 have been implemented. The implementation refers to a set merger as an optimizer since it is the interface from the multi-query optimization subsystem to the rest of the optimization framework.

The hierarchy of classes and interfaces realizing the set mergers from Section 6.3 is shown in Figure 10.14. The class ExhaustiveBestOrderOptimizer provides an exhaustive implementation of a set merger. The arbitrary order set merger is implemented in two variants by ArbitraryOrderOptimizer and AlternativeArbitraryOrderOptimizer where only the latter is extended from AbstractBestOrderOptimizer. AbstractBestOrderOptimizer provides means to store and access sets of queries and is extended by all the optimizers that care about the order of queries (as they use the set of queries to determine the best order), whereas the ArbitraryOrderOptimizer merges each query immediately into a multi-query graph and stores only that one. The GreedyBestOrderOptimizer implements the initial separate, initial pairwise and progressive pairwise order set merger. Slightly more efficient specializations of the two initial set mergers are provided in the classes InitialSeparateGainGreedyOptimizer and InitialPairwiseGainGreedyOptimizer.

The interface *Optimizer* allows to add queries to a set merger, to reset the set of queries, and to optimize the current set into a multi-query graph as shown in Figure 10.15. It extends the *Observer* interface enabling push-based query addition, i.e., whenever another component of the optimization framework such as a query parser of query generator, has a new query available, the `update` method of all *Observers* registered with that component (extending Observable) are called.

10.5 Other Components of the Optimization Framework

The actual optimization framework implemented as part of this thesis, entails numerous packages and classes that are not discussed in this thesis. Notable among these are classes for generating and translating query plans. There are several kinds of translators: in particular, parser from a serial form into a query plan and serializer that create a serial form such as an RPQ or XPath query from a query plan. Several such translators have been implemented, in particular for serializing to and parsing from RPQ, XPath, and the dot-graph visualization language.

The second kind of translators are responsible for generating a physical query plan from a logical query plan represented by a QueryGraph. Two such translators with different capabilities are implemented that generate a SPEX network for a QueryGraph.

Rewriters are components that transform one query plan into another one. Important rewriters are e.g., the InverseJoinRewriter that implements one of the rewriting algorithms for removing inverse relations described in [107] and the BranchPrefixCompacter that implements the prefix compaction in query plans discussed in Section 2.3.3.

For a more extensive description of the optimization framework please refer to the API documentation [50] provided as part of this thesis.

10.6 Testing: `spex.tests`

For testing and evaluation of the method as well as the implementation proposed in this work, several tools for automated testing have been developed. Most notably, these tools allow the generation of large sets of query plans based on a DTD. These query plans can then be evaluated with an arbitrary number of combinations of set merger, pair merger, cost function, serializer etc. in an automated process. The generation

[107] Olteanu, D., et al. 2002. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*. Lecture Notes on Computer Science (LNCS), vol. 2490. Springer Verlag, 109–125.

[50] Furche, T. MQ-SPEX: Multi-query optimization framework for SPEX, API documentation. `http://www.pms.informatik.uni-muenchen.de/forschung/xpath-eval.html`.

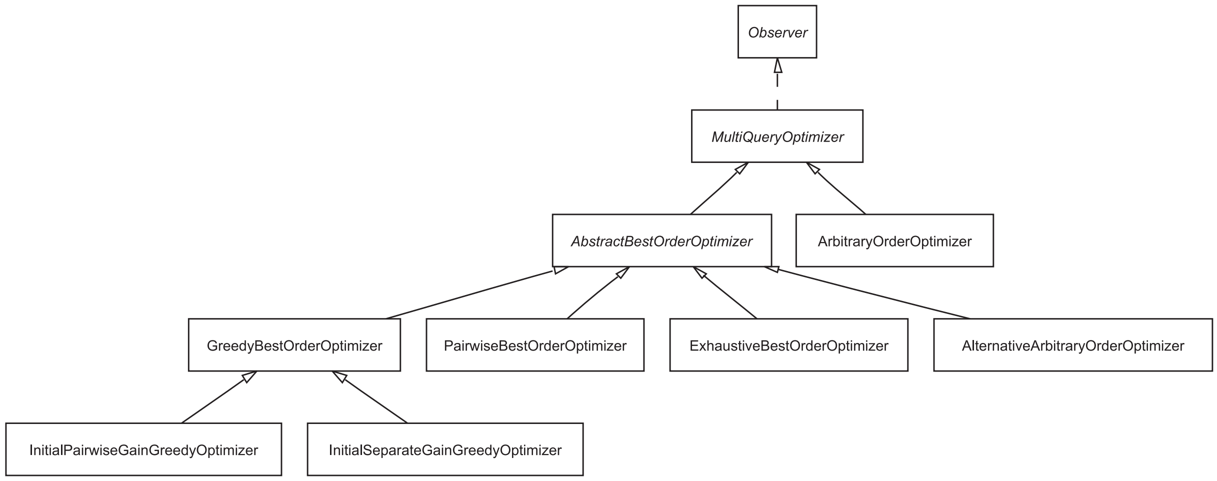


Figure 10.14: Hierarchy of important classes and interfaces in spex.queries.optimizers

as well as the evaluation itself are highly parameterizable. For a more extensive documentation, please refer to the documentation of the API [50] of the optimization framework provided as part of this thesis.

Chapter 11

Conclusion and Future Work

Starting from a discussion about query plans for evaluating queries against an XML stream, an extensive investigation of the optimization of multiple queries to be executed *simultaneously* is provided in this thesis. Query plans are introduced as graphs of operators that specify the flow of data in an evaluation engine. It is emphasized, that sharing of operators becomes a critical issue for optimization against streams, in particular for XML query plans as proposed here. Where previous work in this field is concerned, if at all, only with operator sharing among common prefixes of a query plan, we show that it is feasible to consider sharing of arbitrary operators.

The notion of a query plan and an evaluation model is formalized to facilitate a precise definition of the optimization problem considered. In Chapter 5 we define the minimum common super-plan problem and its more feasible variant, the *stable minimum common super-plan*, formally as optimization problems: The objective is to find, given a set of query plans, a query plan that contains all the original query plans as subgraphs and is cost optimal. The intuition behind this problem definition is that such a query plan evaluates all the queries that are also evaluated by the input query plans using the same evaluation strategy for that query as the corresponding query plan from the input but shares operators among queries wherever the underlying cost function justifies that.

By reducing the maximum common connected subgraph problem to the general *stable minimum common super-plan* problem it is proven that the *stable minimum common super-plan* is NP-hard and NPO PB-complete, i.e., it can not be approximated within n^ϵ for any $\epsilon > 0$.

Chapter 6 proposes several heuristic algorithms for solving this problem. Algorithms (called pair mergers) for two query plans as well as algorithms that consider arbitrary sets of query plans are investigated and several different heuristics are presented for both cases. In particular, two different classes of pair mergers are identified differentiated by what operations are supported by the kind of query plans considered. The incremental pair merger operate on partial solutions and therefore require that there is some way to determine the cost and validity of a partial solution. Local search pair mergers, on the other hand, are often less efficient but require only a transformation function that commutes from one solution to another one.

To evaluate these algorithms, an overview over the SPEX engine, used as basis for the evaluation, is provided: SPEX is a novel evaluation engine proposed in [79; 105] and extended in [106], based on networks of deterministic push-down transducers. We show how to extended this evaluation engine to the query plans for multiple queries generated by the heuristics discussed above.

Based on this evaluation engine and one of the appropriate cost functions, discussed in Chapter 8, the heuristics are evaluated against seven diverse workloads of query plans mimicking common application scenarios as well as the extreme cases. For each of these workloads, the most promising heuristics are

[79] Kiesling, T. 2002. Towards a streamed XPath evaluation. M.S. thesis, University of Munich, Institute of Computer Science.

[105] Olteanu, D., et al. 2003. An evaluation of regular path expressions with qualifiers against XML streams. In *Proc. of the International Conference on Data Engineering (ICDE).*

[106] Olteanu, D., et al. 2003. Advanced techniques for streamed and progressive evaluation of XPath. Research report, University of Munich, Institute for Computer Science.

tested with up to 10,000 query plans with an average size of 15 as input, whereas the test for the remaining heuristics are limited to smaller input sizes.

The experimental evaluation shows, that under the considered cost functions and query plans, the proposed method for optimization of multiple queries can provide with the right combination of heuristics a distinctively better cost than conventional techniques based on prefix compaction only.

Concluding, this work shows that sharing of operators among multiple query plan can, at least for the SPEX evaluation engine, be extended from common prefixes to arbitrary operators in a query plan. Furthermore, a simple greedy heuristic can compute solutions that are clearly superior to solutions constructed if one considers only common prefixes as in previous work in reasonable, albeit longer time. We believe, that the best combination of heuristics identified in Chapter 9 can be employed in practial cases justifying the larger time required for query optimization by considerably reducing the query execution time.

Nevertheless, several open issues remain: In particular, only one of three strategies for solving the stable minimum common super-plan problem has been investigated extensively. For example, adapting techniques for frequent sub-structure discovery [68] in biological data to the problem at hand might prove very beneficial. Furthermore, clustering of the queries, either in a preprocessing step or during the construction of the super-plan, can drastically reduce the complexity of the problem. Related work on graph clustering, reviewed in [21], could provide hints for such an extension.

Genetic algorithms have shown considerable potential as heuristics for solving several graph theoretical problems [37; 98]. Although finding a crossover operation might not be trivial, it could prove very beneficial to investigate such a heuristic.

Finally, we have not considered combining the proposed heuristics in a reasonable way, e.g., to use certain incremental pair mergers to provide starting points for the local search pair merger.

Aside of improving the heuristics, there is one other important open question: Can we find classes of cost functions that are still interesting but for which the stable minimum common super-plan problem becomes easier to approximate or even easier to solve precisely? Although we do not believe, that there is a class of interesting cost functions where the stable minimum common super-plan problem becomes easier to solve, i.e., not NP-hard, there might be classes, where it is easier to approximate, i.e., one can find a heuristic such that a solution constructed by that heuristic has a performance ratio bounded by n^ϵ for some $\epsilon > 0$.

[68] Inokuchi, A., et al. 2000. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the European Conference on Principles and Practice of Knowledge Discovery and Data Mining (PKDD2000)*. 13–23.

[21] Bunke, H. 2000. Recent developments in graph matching. In *Proc. of the International Conference on Pattern Recognition (ICPR)*. Vol. 2.

[37] Cross, A. D. J., et al. 1996. Genetic search for structural matching. In *Computer Vision – ECCV '96*, R. C. B. Buxton, Ed. LNCS 1064. Springer Verlag, 514–525.

[98] Michalewicz, Z. 1996. *Genetic Algorithms + Data Structures = Evolution Programs*, 2nd ed. Springer Verlag.

Appendix A

Bibliography

Contents

[1] Aboulnaga, A., Alameldeen, A. R., and Naughton, J. F. Estimating the selectivity of XML path expressions for internet scale applications. In *Proc. of the International Conference on Very Large Databases (VLDB)*. 2001.

[2] Aguilera, M. K., Strom, R. E., Sturman, D. C., Astley, M., and Chandra, T. D. 1999. Matching events in a content-based subscription system. In *Proc. of the ACM Symposium on Principles of Distributed Computing*. ACM Press, 53–61.

[3] Alex C. Snoeren, Kenneth Conley, D. K. G. 2001. Mesh-based content routing using XML. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*. 160–173.

[4] Altinel, M. and Franklin, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the International Conference on Very Large Databases (VLDB)*.

[5] Armen, C. and Stein, C. 1994. A $2\frac{3}{4}$-approximation algorithm for the shortest superstring problem. Tech. Rep. PCS-TR94-214, Department of Computer Science, Dartmouth College, Hannover (NH).

[6] Arora, S. 1998. The approximability of NP-hard problems. In *Proc. of the ACM Symposium on Theory of Computing*. 337–348.

[7] Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., and Protasi, M. 1999. *Complexity and Approximation: Combinatorial Optimization Problems and their Approximability Properties*. Springer Verlag, Berlin.

[8] Avila-Campillo, I., Gupta, A., Onizuka, M., Raven, D., and Suciu, D. 2002. XMLTK: An XML toolkit for scalable XML stream processing. In *Proc. of the Workshop on Programming Language Technologies for XML (PLAN-X)*. Proc. available at `http://www.research.avayalabs.com/user/wadler/planx/planx-eproceed/proc%eed.html`.

[9] Avnur, R. and Hellerstein, J. M. 2000. Eddies: Continuously adaptive query processing. In *Proc. of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 261–272.

[10] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. 2002. Models and issues in data stream systems. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*.

[11] Babu, S. and Widom, J. 2001. Continuous queries over data streams. *SIGMOD (ACM Special Interest Group on Management of Data) Record*, 109–120.

[12] Banavar, G., Chandra, T. D., Mukherjee, B., Nagarajarao, J., Strom, R. E., and Sturman, D. C. 1999. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*. 262–272.

[13] Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M., and Josifovski, V. 2002. An algorithm for streaming XPath processing with forward and backward axes. In *Proc. of the Workshop on Programming Language Technologies for XML (PLAN-X)*. Proc. available at `http://www.research.avayalabs.com/user/wadler/planx/planx-eproceed/proc%eed.html`.

[14] Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M., and Josifovski, V. 2003. Streaming XPath processing with forward and backward axes.

In *Proc. of the International Conference on Data Engineering (ICDE)*.

[15] Berlund, A., Boag, S., Chamberlin, D., Fernandez, M. F., Kay, M., Robie, J., and Siméon, J., Eds. 2002. XML path language (XPath) 2.0. Working draft, World Wide Web Consortium. `http://www.w3.org/TR/xpath20/`.

[16] Blum, A., Jiang, T., Li, M., Tromp, J., and Yannakakis, M. 1994. Linear approximation of shortest superstrings. *Journal of the ACM 41*, 630–647.

[17] Bonnet, P., Gehrke, J., and Seshadri, P. 2001. Towards sensor database systems. In *Proc. of the International Conference on Mobile Data Management (ICMDM)*. 3–14.

[18] Botts, M., Ed. 2002. Sensor model language (SensorML) for in-situ and remote sensors specification. discussion paper 02-026r4, Open GIS Consortium. `http://www.opengis.org/techno/discussions/02-026r4.pdf`.

[19] Botts, M. and Reichardt, M. 2003. Sensor web enablement. white paper, Open GIS Consortium. `http://www.opengis.org/pressrm/summaries/SensorWebWhPpr030512.doc`.

[20] Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E., Eds. 2000. Extensible markup language (XML) 1.0 (second edition). Recommendation, World Wide Web Consortium. `http://www.w3.org/TR/REC-xml`.

[21] Bunke, H. 2000. Recent developments in graph matching. In *Proc. of the International Conference on Pattern Recognition (ICPR)*. Vol. 2.

[22] Bunke, H., Jiang, X., and Kandel, A. 2000. On the minimum common supergraph of two graphs. *Springer Computing 65,* 1, 13–25.

[23] Calvanese, D., Giacomo, G. D., Lenzerini, M., and Vardi, M. Y. 2000. Containment of conjunctive regular path queries with inverse. In *Proc. of the International Conference on the Principles of Knowledge Representation and Reasoning (KR)*. 176–185.

[24] Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. 2002. Monitoring streams: A new class of data management applications. In *Proc. of the International Conference on Very Large Databases (VLDB)*.

[25] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 2000. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. of the ACM Symposium on Principles of Distributed Computing*. ACM Press, 219–227.

[26] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 2001. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS) 19,* 3, 332–383.

[27] Carzaniga, A. and Wolf, A. L. 2001. Fast forwarding for content-based networking. Tech. Rep. CU-CS-922-01, Department of Computer Science, University of Colorado.

[28] Chan, C.-Y., Felber, P., Garofalakis, M., and Rastogi, R. 2002a. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal (Special Issue on XML Data Management)*.

[29] Chan, C.-Y., Felber, P., Garofalakis, M., and Rastogi, R. 2002b. Efficient filtering of XML documents with XPath expressions. In *Proc. of the International Conference on Data Engineering (ICDE)*. 235–244.

[30] Chandrasekaran, S. and Franklin, M. J. 2002. Streaming queries over streaming data. In *Proc. of the International Conference on Very Large Databases (VLDB)*.

[31] Chen, J., DeWitt, D. J., , and Naughton, J. F. 2002. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proc. of the International Conference on Data Engineering (ICDE)*.

[32] Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In Proc. of the ACM SIGMOD International Conference on Management of Data. *SIGMOD Record 29,* 2, 379–390.

[33] Cisco Systems. 2000. Cisco IOS netflow – technology data sheet. `http://www.cisco.com/warp/public/cc/pd/iosw/prodlit/iosnf_ds.pdf`.

[34] Clark, J. and DeRose, S., Eds. 1999. XML path language (XPath) version 1.0. Recommendation, World Wide Web Consortium. `http://www.w3.org/TR/xpath`.

[35] Crescenzi, P. and Panconesi, A. 1991. Completeness in approximation classes. *Information and Computation 93,* 2, 241–262.

[36] Crespo, A., Buyukkokten, O., and Garcia-Molina, H. 2003. Query merging: Improving query subscription processing in a multicast environment. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*.

[37] Cross, A. D. J., Wilson, R. C., and Hancock, E. R. 1996. Genetic search for structural matching. In *Computer Vision – ECCV '96*, R. C. B. Buxton, Ed. LNCS 1064. Springer Verlag, 514–525.

[38] Dayal, U., Hanson, E. N., and Widom, J. 1995. Active database systems. In *Modern Database Systems*. 434–456.

[39] Deering, S. E. and Cheriton, D. R. 1990. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems (TOCS) 8,* 2, 85–110.

[40] Desai, A. 2001. Introduction to sequential XPath. In *Proc. of the IDEAlliance XML Conference.* Electronic Proc. available at `http://www.idealliance.org/papers/xml2001/`.

[41] Diao, Y., Altinel, M., Franklin, M. J., Zhang, H., and Fischer, P. 2002. Path sharing and predicate evaluation for high-performance XML filtering. Submitted for publication, `www.cs.berkeley.edu/~diaoyl/publications/yfilter-public.ps`.

[42] Diao, Y., Fischer, P., Franklin, M. J., and To, R. 2002. YFilter: Efficient and scalable filtering of XML documents. In *Proc. of the International Conference on Data Engineering (ICDE).*

[43] Douglass, R., Mork, J., and Suresh, B. 1997. Battlefield awareness and data dissemination (BADD for the warfighter. In *Proc. of the SPIE*, B. R. Suresh, Ed. Vol. 3080. SPIE – The International Society for Optical Engineering, 18–24.

[44] Duffield, N. G. and Grossglauser, M. 2001. Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Networking (TON) 9,* 3, 280–292.

[45] Eric, H., Al-Fayoumi, N., Carnes, C., Kandil, M., Liu, H., Lu, M., Park, J., and Vernon, A. 1997. TriggerMan: An asynchronous trigger processor as an extension to an object-relational DBMS. Tech. Rep. 97-024, University of Florida, CISE Department.

[46] Eric N. Hanson, T. J. 1996. Selection predicate indexing for active databases using interval skip lists. *Information Systems 21,* 3, 269–298.

[47] Fabret, F., Jacobsen, H.-A., Llirbat, F., Pereira, J., Ross, K. A., and Shasha, D. 2001. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of the ACM SIGMOD International Conference on Management of Data.* ACM Press, 115–126.

[48] Finkelstein, S. J. 1982. Common subexpression analysis in database applications. In *Proc. of the ACM SIGMOD International Conference on Management of Data.* 235–245.

[49] Franklin, M. J., Ed. 1996. *Special Issue on Data Dissemination.* Data Engineering Bulletin, vol. 19, 3. IEEE Computer Society.

[50] Furche, T. MQ-SPEX: Multi-query optimization framework for SPEX, API documentation. `http://www.pms.informatik.uni-muenchen.de/forschung/xpath-eval.html`.

[51] Garcia-Molina, H., Ullmann, J. D., and Widom, J. 2001. *Database systems: the complete book*, 1st ed. Prentice Hall, Upper Saddle River, New Jersey.

[52] Gore, P., Cytron, R., Schmidt, D., and O'Ryan, C. 2001. Designing and optimizing a scalable CORBA notification service. *ACM SIGPLAN Notices 36,* 8, 196–204.

[53] Gottlob, G., Koch, C., and Pichler, R. 2003. The complexity of XPath query evaluation. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS).* 179–190.

[54] Gough, J. and Smith, G. 1995. Efficient recognition of events in a distributed system. In *Proc. of the Australasian Computer Science Conference.*

[55] Graefe, G. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys 25,* 2, 73–170.

[56] Green, T. J., Miklau, G., Onizuka, M., and Suciu, D. 2003. Processing XML streams with deterministic automata. In *Proc. of the International Conference on Database Technology (ICDT).* 173–189.

[57] Gruber, R. E., Krishnamurthy, B., and Panagos, E. 1999. The architecture of the READY event notification service. In *Proc. of the ICDCS Workshop on Electronic Commerce and Web-Based Applications.*

[58] Gruber, R. E., Krishnamurthy, B., and Panagos, E. 2000. READY: A high performance event notification service. In *Proc. of the International Conference on Data Engineering (ICDE).* 668–669.

[59] Gupta, A. and Nishimura, N. 1998. Finding largest subtrees and smallest supertrees. *Algorithmica 21,* 2, 183–210.

[60] Gupta, A. K. and Suciu, D. 2003. Stream processing of XPath queries with predicates. In *Proc. of the Proc. of the ACM SIGMOD International Conference on Management of Data.*

[61] Hanson, E. N. 1991. The interval skip list: A data structure for finding all intervals that overlap a point. In *Proc. of Workshop on Algorithms and Data Structures, Ottawa, Canada.* Springer Verlag, 153–164.

[62] Hanson, E. N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parthasarathy, S., Park, J. B., and Vernon, A. 1999. Scalable Trigger Processing. In *Proc. of the International Conference on Data Engineering (ICDE).* IEEE Computer Society Press, 266–275.

[63] Hanson, E. N. and Chaabouni, M. 1990. The IBS-tree: A data structure for finding all intervals that overlap a point. Tech. Rep. WSU-CS-90-11, Dept. of Computer Science and Engineering, Wright State University. Available at `ftp://ftp.cis.ufl.edu/cis/tech-reports/tr94/tr94-040.ps`.

[64] Hanson, E. N., Chaabouni, M., Kim, C.-H., and Wang, Y.-W. 1990. A predicate matching algorithm for database rule systems. In *Proc. of the ACM SIGMOD International Conference on Management of Data.* ACM Press, 271–280.

[65] Harnden, F. R., Primini, F. A., and Payne, H. E., Eds. 2001. *Astronomical Data Analysis Software and Sys-*

tems X: Science Data Pipelines. ASP (Astronomical Society of the Pacific) Conference Series, vol. 238.

[66] Herman, I., Melançon, G., and Marshall, M. S. 2000. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics 6,* 1, 24–43.

[67] Hochbaum, D., Ed. 1996. *Approximation Algorithms for NP-hard Problems*, 1st ed. Brooks Cole.

[68] Inokuchi, A., Washio, T., and Motoda, H. 2000. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the European Conference on Principles and Practice of Knowledge Discovery and Data Mining (PKDD2000)*. 13–23.

[69] Inokuchi, A., Washio, T., and Motoda, H. 2003. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning 50,* 3, 321–354.

[70] Inokuchi, A., Washio, T., Nishimura, Y., and Motoda, H. 2002. General framework for mining frequent patterns from structures. In *Proc. of the International Workshop on Active Mining (AM 2002)*. 23–30.

[71] International Press Telecommunications Council. News industry text format (NITF). `http://www.nitf.org`.

[72] Ives, Z. G., Halevy, A. Y., and Weld, D. S. 2001. Integrating network-bound XML data. *IEEE Data Engineering Bulletin 24,* 2, 20–26.

[73] Ives, Z. G., Halevy, A. Y., and Weld, D. S. 2002. An XML query engine for network-bound data. *VLDB Journal Special Issue on XML Data Management*.

[74] Joy, B., Steele, G., Gosling, J., and Bracha, G. 2000. *The Java Language Specification*, 2nd ed. Addison-Wesley.

[75] Kann, V. 1992. On the approximability of the maximum common subgraph problem. In *Proc. 9th Symp. Theoretical Aspects of Computer Science*. Number 577 in Lecture Notes in Computer Science. Springer Verlag, 377–388.

[76] Kantor, B. and Lapsley, P., Eds. 1986. Network news transfer protocol – a proposed standard for the stream-based transmission of news. RFC 977, IETF. `http://www.ietf.org/rfc/rfc0977.txt`.

[77] Keidl, M., Kreutz, A., Kemper, A., and Kossmann, D. 2002. A publish & subscribe architecture for distributed metadata management. In *Proc. of the International Conference on Data Engineering (ICDE)*. 309–320.

[78] Khanna, S., Motwani, R., Sudan, M., and Vazirani, U. 1999. On syntactic versus computational views of approximability. *SIAM Journal on Computing 28,* 1, 164–191.

[79] Kiesling, T. 2002. Towards a streamed XPath evaluation. M.S. thesis, University of Munich, Institute of Computer Science. Description and diploma thesis at `http://www.pms.informatik.uni-muenchen.de/lehre/projekt-diplom-arbeit/s%treamedxpath.html`.

[80] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. 1983. Optimization by simulated annealing. *Science 220,* 4598, 671–680.

[81] Krishnamurthy, B. and Rosenblum, D. S. 1995. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering (TSE) 21,* 10, 845–857.

[82] Kuramochi, M. and Karypis, G. 2002. An efficient algorithm for discovering frequent subgraphs. Tech. Rep. 02-026, Computer Science Departement, University of Minnesota.

[83] Lakshmanan, L. V. and Parthasarathy, S. 2002. On efficient matching of streaming XML documents and queries. In *Proc. of the International Conference on Extending Database Technology (EDBT)*. 142–160.

[84] Lim, L., Wang, M., Padmanabhan, S., Vitter, J. S., and Parr, R. 2002. XPathLearner: An on-line self-tuning markov histogram for XML path selectivity estimation. In *Proc. of the International Conference on Very Large Databases (VLDB)*.

[85] Liu, L., Pu, C., Barga, R., and Zhou, T. 1996. Differential evaluation of continual queries. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*. 458–465.

[86] Liu, L., Pu, C., and Tang, W. 1999. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 11,* 4, 610–628.

[87] Ludäscher, B., Mukhopadhyay, P., and Papakonstantinou, Y. 2002. A transducer-based XML query processor. In *Proc. of the International Conference on Very Large Databases (VLDB)*.

[88] Madden, S. and Franklin, M. J. 2002. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of the International Conference on Data Engineering (ICDE)*.

[89] Madden, S., Shah, M., Hellerstein, J. M., and Raman, V. 2002. Continuously adaptive continuous queries over streams. In *Proc. of the ACM SIGMOD International Conference on Management of Data*.

[90] Maier, D. and Storer, J. A. 1977. A note on the complexity of the superstring problem. Tech. Rep. 233, Princeton University. Oct.

[91] Marshall, M. S., Herman, I., and Melançon, G. 2001. An object-oriented design for graph visualization. *Software Practice and Experience 31,* 8, 739–756.

[92] Martínez, J. M., Ed. 2002. Mpeg-7 overview. Tech. Rep. N4980, ISO/IEC JTC1/SC29/WG11. http://mpeg.telecomitalialab.com/standards/mpeg-7/mpeg-7.htm.

[93] McGregor, J. J. 1982. Backtrack search algorithms and the maximal common subgraph problem. *Software-Practice and Experience 12*, 23–34.

[94] Megginson, D. and Brownell, D. 2002. SAX: The simple API for XML. http://www.saxproject.org/.

[95] Mehringer, D. M., Plante, R. L., and Roberts, D. A., Eds. 1999. *Astronomical Data Analysis Software and Systems VIII: Data Pipelines*. ASP (Astronomical Society of the Pacific) Conference Series, vol. 172.

[96] Meuss, H. and Schulz, K. 2001. Complete answer aggregates for tree-like databases: A novel approach to combine querying and navigation. *ACM Transactions on Information Systems (TOIS) 19,* 2, 161–215.

[97] Mühl, G., Fiege, L., and Buchmann, A. 2002. Filter similarities in content-based publish/subscribe systems. In *Proc. of the International Conference on Architecture of Computing Systems (ARCS).* Lecture Notes in Computer Science, vol. 2299. Springer Verlag, 224–238.

[98] Michalewicz, Z. 1996. *Genetic Algorithms + Data Structures = Evolution Programs*, 2nd ed. Springer Verlag.

[99] Michalewicz, Z. and Fogel, D. B. 2000. *How to Solve It: Modern Heuristics*, 1st ed. Springer Verlag.

[100] Miklau, G. XML data repository. http://www.cs.washington.edu/research/xmldatasets/, University of Washington.

[101] Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R. 2003. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the Conference on Innovative Data Systems Research (CIDR).*

[102] Nguyen, B., Abiteboul, S., Cobena, G., and Preda, M. 2001. Monitoring XML data on the Web. *SIGMOD (ACM Special Interest Group on Management of Data) Record 30,* 2, 437–448.

[103] Object Management Group, Inc. 2001. *Event Service Specification*, 1.1 ed. Object Management Group, Inc. http://www.omg.org/technology/documents/formal/event_service.htm.

[104] Object Management Group, Inc. 2002. *Notification Service Specification*, 1.0.1 ed. Object Management Group, Inc. http://www.omg.org/technology/documents/formal/notification_service.htm%.

[105] Olteanu, D., Kiesling, T., and Bry, F. 2003. An evaluation of regular path expressions with qualifiers against XML streams. In *Proc. of the International Conference on Data Engineering (ICDE).*

[106] Olteanu, D., Kiesling, T., Furche, T., and Bry, F. 2003. Advanced techniques for streamed and progressive evaluation of XPath. Research report, University of Munich, Institute for Computer Science. http://www.pms.informatik.uni-muenchen.de/forschung/xpath-eval.html.

[107] Olteanu, D., Meuss, H., Furche, T., and Bry, F. 2002. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM).* Lecture Notes on Computer Science (LNCS), vol. 2490. Springer Verlag, 109–125.

[108] Ozen, B., Kilic, O., Altinel, M., and Dogac, A. 2001. Highly personalized information delivery to mobile clients. In *Proc. of ACM International Workshop on Data Engineering for Wireless and Mobile Access.*

[109] Ozkan, C., Dogac, A., and Evrendilek, C. 1995. A heuristic approach for optimization of path expressions. In *Proc. of the International Conference on Database and Expert Systems Applications.* 522–534.

[110] Papadimitriou, C. H. and Yannakakis, M. 1991. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences 43*, 425–440.

[111] Peng, F. and Chawathe, S. S. 2003a. XPath queries on streaming data. In *Proc. of the Proc. of the ACM SIGMOD International Conference on Management of Data.*

[112] Peng, F. and Chawathe, S. S. 2003b. XSQ: Streaming XPath queries. In *Proc. of the International Conference on Data Engineering (ICDE).*

[113] Pereira, J., Fabret, F., Jacobsen, H.-A., Llirbat, F., and Shasha, D. 2001. WebFilter: A high-throughput XML-based publish and subscribe system. In *Proc. of the International Conference on Very Large Databases (VLDB).* 723–724.

[114] Pereira, J., Fabret, F., Llirbat, F., Preotiuc-Pietro, R., Ross, K. A., and Shasha, D. 2000. Publish/subscribe on the web at extreme speed. In *Proc. of the International Conference on Very Large Databases (VLDB).* 627–630.

[115] Pereira, J., Fabret, F., Llirbat, F., and Shasha, D. 2000. Efficient matching for web-based publish/subscribe systems. In *Proc. of the International Conference on Cooperative Information Systems.* Lecture Notes in Computer Science, vol. 1901. Springer Verlag, 162–173.

[116] Polyzotis, N. and Garofalakis, M. 2002. Statistical synopses for graph-structured XML databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data.*

[117] Ramakrishnan, S. and Dayal, V. 1998. The pointcast network. In *Proc. of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 520.

[118] Reiss, S. P. 1990. Connecting tools using message passing in the field environment. *IEEE Software 7,* 4, 57–66.

[119] Rosenthal, A. and Chakravarthy, U. S. 1988. Anatomy of a modular multiple query optimizer. In *Proc. of the International Conference on Very Large Databases (VLDB)*. 230–239.

[120] Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. 2000. Efficient and extensible algorithms for multi query optimization. *SIGMOD (ACM Special Interest Group on Management of Data) Record 29,* 2, 249–260.

[121] Segall, B. and Arnold, D. 1997. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proc. of AUUG (Australian Unix and Open Systems User Group) '97*.

[122] Segall, B., Arnold, D., Boot, J., Henderson, M., and Phelps, T. 2000. Content based routing with elvin4. In *Proc. of AUUG2K (Australian Unix and Open Systems User Group)*.

[123] Sellis, T. K. 1988. Multiple-query optimization. *ACM Transactions on Database Systems (TODS) 13,* 1, 23–52.

[124] Sellis, T. K. and Ghosh, S. 1990. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 2,* 2, 262–266.

[125] Shmueli, O. 1983. Dynamic cycle detection. *Information Processing Letters 17,* 4, 185–188.

[126] Spannagel, M. 2003. SPEX Viewer: A graphical user interface for SPEX. Project thesis, University of Munich, Institute for Computer Science. `http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten%/Markus.Spannagel/SPEX_Viewer.html`.

[127] Spears, W. M. 1996. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26. American Mathematical Society, Chapter Simulated Annealing for Hard Satisfiability Problems, 533–558.

[128] Sullivan, M. and Heybey, A. 1998. Tribeca: A system for managing large databases of network traffic. In *Proc. of the USENIX Annual Technical Conference*.

[129] Sun Microsystems, Inc. 2001. *Jini^{TM} Technology Core Platform Specification*, 1.2 ed. Sun Microsystems, Inc. `http://wwws.sun.com/software/jini/specs/jini1.2html/core-title.html`.

[130] Sun Microsystems, Inc. 2002. *Java Message Service API Specification*, 1.1 ed. Sun Microsystems, Inc. `http://java.sun.com/products/jms/`.

[131] Terry, D. B., Goldberg, D., Nichols, D., and Oki, B. M. 1992. Continuous queries over append-only databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 321–330.

[132] Turner, J. S. 1989. Approximation algorithms for the shortest common superstring problem. *Information and Computation 83,* 1 (Oct.), 1–20.

[133] Ukkonen, E. 1990. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica 5*, 313–323.

[134] Ullmann, J. R. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM 23,* 1, 31–42.

[135] Wang, K. and Liu, H. 1999. Discovering structural association of semistructured data. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*.

[136] Widom, J. and Ceri, S., Eds. 1996. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann.

[137] Wu, Y., Patel, J. M., and Jagadish, H. V. 2002. Estimating answer sizes for XML queries. In *Proc. of the International Conference on Extending Database Technology (EDBT)*. 590–608.

[138] Xu, L. and Oja, E. 1990. Improved simulated annealing, boltzmann machine, and attributed graph matching. L. Almeida, Ed. LNCS 412. Springer Verlag, 151–161.

[139] Yamaguchi, A., Nakano, K., and Miyano, S. 1997. An approximation algorithm for the minimum common supertree problem. *Nordic Journal of Computing 4,* 3, 303–316.

[140] Yan, T. W. and Garcia-Molina, H. 1999. The sift information dissemination system. *ACM Transactions on Database Systems (TODS) 24,* 4, 529–565.

[141] Yannakakis, M. 1978. The node-deletion problem for hereditary properties. Tech. Rep. 240, Computer Science Laboratory, Princeton University.

[142] Yannakakis, M. 1979. The effect of a connectivity requirement on the complexity of maximum subgraph problems. *Journal of the ACM 26,* 4, 618–630.

[143] Yu, H., Estrin, D., and Govindan, R. 1999. A hierarchical proxy architecture for internet-scale event services. In *Proc. of the IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*.

[144] Yu, P. S., Ed. 2003. *IEEE Transactions on Knowledge and Data Engineering: special section on online analysis and querying of continuous data streams*. Vol. 15. IEEE Computer Society.