Efficient storage and query processing of XML data in relational database systems

Sandeep Prakash

2005

Sandeep, P. (2005). Efficient storage and query processing of XML data in relational database systems. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2438>

<https://doi.org/10.32657/10356/2438>

Nanyang Technological University

Downloaded on 26 Apr 2023 20:24:54 SGT

Efficient Storage and Query Processing of XML Data in Relational Database Systems

Sandeep Prakash

School of Computer Engineering

A thesis submitted to the Nanyang Technological University in fulfilment of
the requirement for the degree of Master of Engineering

2005

Abstract

The popularity of XML has lead to a plethora of data that is less structured and does not follow a pre-defined schema. As a result, there is a growing need for data management systems that allow the storage and querying of such data. One avenue is the use of existing database technology. The goal of this thesis is to investigate relational storage approaches for XML data. To that end we present two schema-oblivious relational storage approaches - SUCXENT and SUCXENT++ that provide tremendous improvement over existing schema-oblivious approaches. Among these two approaches SUCXENT++ presents a more significant improvement in performance. We also present algorithms that translate some types of XQuery queries into SQL based on the these two approaches.

Existing literature indicates that schema-conscious approaches perform better than schema-oblivious approaches. Our experiments inidicate that though this may be true for most types of queries, schema-oblivious approaches (specifically SUCXENT++) perform better when it comes to recursive queries. In addition, the performance of such approaches is hindered by the inability of the relational query optimizer to generate optimal query plans. We propose optimizations to the XML query-to-SQL transaction process that overcome this problem and improve the performance of SUCXENT++ by up to 40 times. We also present a data partitioning strategy that utilizes the query workload to generate partitions which can be queried instead of the entire data set. This results in a performance improvement of up to 450 times. We also present a novel GUI-based query system for the visual formulation of XQuery queries. GUI-latency driven prefetching is employed to optimize query execution further resulting in performance improvements of up to 96%.

Acknowledgments

Firstly, I would like to express my sincere gratitude to my supervisor, Dr. Sourav S. Bhowmick for his support. He allowed me the flexibility to investigate research issues in my own way and at the same time, was always there to assist and advise me when I faced problems. I would also like to thank Mr. Lai Chee Keong and Ms. Tan Lay Choo who are the administrators in my research center for their assistance in providing administrative and technical support. Thanks also to Zhao Qiankun, Chen Ling and Erwin Leonardi for the enlightening discussions I had with them.

My appreciation for all those people who helped out along the way in one way or another. Finally, I would like to thank the School of Computer Engineering of Nanyang Technological University, Singapore for the generous resources and financial support provided for the duration of the course.

Contents

Abstract	ii
Acknowledgments	iii
List of Figures	viii
1 Introduction	1
1.1 Storing and Querying XML data	4
1.1.1 Storage Approaches	6
1.1.2 Querying XML	7
1.2 Research Issues and Challenges	8
1.2.1 Appropriate Relational Schema	9
1.2.2 Query Management	10
1.2.3 Query Optimization	10
1.3 Contributions	11
1.3.1 RDBMS-based Storage of XML	11
1.3.2 Querying XML	11
1.4 Report Organization	12
2 Related Work	13
2.1 XML Storage	13
2.2 XML in a Relational Database System	13
2.2.1 Schema-oblivious Approaches	14
2.2.2 Schema-conscious Approaches	17
2.3 Native XML Storage	18
2.4 Query translation	20
2.5 Optimizing XML query processing	21
2.5.1 Index Structures and Algorithms	21

2.5.2	Query Plan Selection	23
2.6	Partitioning and Prefetching	25
2.7	GUI-Latency Driven Query Optimization	29
3	Relational Schema Design	31
3.1	Introduction	31
3.2	The SUCXENT Approach	32
3.2.1	Schema Description	32
3.2.2	Document Extraction	38
3.2.3	Comparison with Existing Approaches	40
3.2.4	Limitations of SUCXENT	41
3.3	The SUCXENT++ Approach	42
3.3.1	Schema Description	43
3.3.2	Extending the SUCXENT++ Schema	44
3.3.3	Extraction	50
3.4	Performance Evaluation	50
3.4.1	Experimental Setup	51
3.4.2	Insertion/Extraction of XML Documents	51
3.4.3	Storage Size	54
3.5	Summary	54
4	Query Processing	56
4.1	XQuery Syntax Overview	56
4.1.1	Types of XQuery Supported	59
4.2	Overview	60
4.3	Translation in SUCXENT	62
4.3.1	Simple Query Translation	63
4.3.2	Join Query Translation	65
4.3.3	Comparison with Existing Approaches	67
4.4	Translation in SUCXENT++.	67
4.4.1	Query Translation	67
4.4.2	Comparison with SUCXENT	70
4.4.3	Scope of Query Translation	71

4.5	Performance Evaluation	71
4.5.1	Data Set and Queries	72
4.5.2	Won-recursive Query Performance - XBench	73
4.5.3	Query Performance - Recursive queries	81
4.5.4	Query Performance Summary	87
4.6	Summary	87
5	Query Optimization	89
5.1	Query Plan Alteration	89
5.1.1	Issues with the Relational Query Optimizer	90
5.1.2	Optimizing Simple Path Expressions	90
5.1.3	Optimizing Recursive Path Expressions	91
5.1.4	Performance Study	94
5.1.5	Summary	97
5.2	Data Partitioning	98
5.2.1	Overview of Partitioning	99
5.2.2	Problem Description	100
5.2.3	Query Workload-based Partitioning	105
5.2.4	Query Translation	112
5.2.5	Partition Maintenance	113
5.2.6	Performance Evaluation	118
5.2.7	Comparison with Existing Work	124
5.3	GUI-driven Prefetching	126
5.3.1	Visual Query Interface	128
5.3.2	Query Formulation Time (QFT)	130
5.3.3	Visual Query to XQuery Translation	134
5.3.4	Prefetching Model	135
5.3.5	Performance Evaluation	146
5.3.6	Summary	151
5.4	Summary	151

6	Conclusions and Future Work	153
6.1	Contributions	153
6.2	Conclusions	155
6.3	Future Works	156
A	SQL Queries	157
A.1	SQL Queries for Xbench in SUCXENT	157
A.2	SQL Queries for Xbench in SUCXENT++	167
A.3	SQL Queries for Xbench after Partitioning in SUCXENT++	174
A.4	SQL Queries for Recursive XQuery in SUCXENT	182
A.5	S&L Queries for Recursive XQuery in SUCXENT++.	185
B	Query Timings	194
	References	199

List of Figures

1.1	Abstracting away information.	3
1.2	Semistructured data - SWISS-PROT.	4
1.3	Heterogeneous data integration.	5
1.4	Architecture.	5
2.1	Structural-join techniques.	23
3.1	Sample XML data.	32
3.2	Tree representation.	33
3.3	SUCXENT schema.	33
3.4	SUCXENT: XML data in RDBMS.	38
3.5	Extraction algorithm.	39
3.6	Partial tree.	40
3.7	XML data in XParent.	42
3.8	SUCXENT++ schema.	43
3.9	SUCXENT++: XML data in RDBMS.	44
3.10	Query tree and query processing in sc Sucxent++	45
3.11	Data set.	49
3.12	Data characteristics	50
3.13	Insertion times (Data Centric).	50
3.14	Insertion times (Text Centric).	51
3.15	Extraction times (Data Centric).	52
3.16	Extraction times (Text Centric).	52
3.17	Storage size (Data Centric).	55
3.18	Storage size (Text Centric).	55
4.1	XQuery Syntax.	

4.2	Samplequery.	57
4.3	Parse Tree.	58
4.4	Modified XQuery syntax.	61
4.5	Modified parse tree (Abstract Query Tree).	61
4.6	Translation algorithm.	62
4.7	Functions to generate where-clause-predicate of Figure 4.6. . .	63
4.8	Functions to generate where-clause-ixn of Figure 4.6.	64
4.9	Translated SQL query.	65
4.10	Join query.	65
4.11	Join query translation.	66
4.12	Translated SQL query in XParent.	68
4.13	Translated SQL query in SUCXENT++.	68
4.14	Translation algorithm for SUCXENT++.	69
4.15	Join query translation in SUCXENT++.	70
4.16	XBench queries.	72
4.17	XBench queries (contd.).	73
4.18	Data set.	73
4.19	Recursive queries and their features.	74
4.20	Query execution time (Data-Centric)	74
4.21	Query execution time (Text-Centric)	76
4.22	Query execution time (Data-Centric)	76
4.23	Query execution time (Text-Centric)	77
4.24	Query execution time (Data-Centric)	78
4.25	Query execution time (Text-Centric)	80
4.26	Query performance summary.	82
4.27	Sample DTD.	83
4.28	Sample XML document.	83
4.29	Sample recursive query.	84
4.30	Query Tree.	84
4.31	Query performance.	84
4.32	Variation with distance.	85
4.33	Variation with D – depth (D = <i>maximum</i> depth)	85

4.34 Summary	87
5.1 Query and query plan with path joins	91
5.2 Query and query plan without path joins	91
5.3 DTD graph and path numbering	92
5.4 Sample XML data	92
5.5 Modified schema	93
5.6 Relational decomposition for modified schema	93
5.7 Query execution time (Data-Centric)	95
5.8 Query execution time (Text-Centric)	95
5.9 Query execution time (Data-Centric)	96
5.10 Query execution time (Text-Centric)	96
5.11 Query execution time (Data-Centric)	97
5.12 Query execution time (Text-Centric)	97
5.13 Performance summary	98
5.14 Performance summary (contd)	98
5.15 Query performance	99
5.16 Summary	99
5.17 XQuery2 and SQL translation	100
5.18 XQuery3 and SQL translation	100
5.19 Partitions for data in Figure 3.9	102
5.20 Translated queries for naive partitioning	103
5.21 Translated queries for workload-based partitioning	103
5.22 Path expressions resolve to more than one root-to-leaf path	104
5.23 Path expressions resolve to more than one root-to-leaf path	105
5.24 General XQuery and SQL translation	106
5.25 Q-Sets	106
5.26 Global set of Path-sets	107
5.27 Q-Set generation algorithm	107
5.28 Partition generation algorithm	108
5.29 Translation algorithm with partitions	112
5.30 Partition maintenance on data/schema change	113
5.31 Partition maintenance on data addition	114

5.32	Partition maintenance on data removal.	115
5.33	Partition maintenance on query workload change.	116
5.34	Addition of new query to workload.	116
5.35	Partition modification due to query workload change (XQuery 4 added).	117
5.36	Partition modification due to query workload change.	117
5.37	Query execution time (Data-Centric)	118
5.38	Query execution time (Text-Centric)	118
5.39	Query execution time (Data-Centric)	119
5.40	Query execution time (Text-Centric)	119
5.41	Query execution time (Data-Centric)	120
5.42	Query execution time (Text-Centric)	120
5.43	Performance summary.	121
5.44	Performance summary (contd.).	121
5.45	Query performance	122
5.46	Summary.	122
5.47	Partition maintenance cost - Data change.	123
5.48	Increasing workload.	123
5.49	Partition maintenance cost - Query workload change.	123
5.50	Partition change as workload increases.	124
5.51	Partition change as workload decreases.	125
5.52	Query interface.	127
5.53	Query Editor.	128
5.54	Query.	129
5.55	Correcting mistakes.	131
5.56	Examples of QFE.	133
5.57	GUI task times.	135
5.58	Modified schema.	136
5.59	Relational decomposition for modified schema to facilitate prefetching.	136
5.60	Prefetching algorithm.	137
5.61	selectMaterialization().	138
5.62	Algorithm replaceMaterialization.	140
5.63	Prefetching steps.	141

5.64	Prefetching scenarios.	141
5.65	Data set and queries.	146
5.66	Materialization replacement cost.	147
5.67	NET vs. TPT.	147
5.68	NET vs EO_PRT.	148
5.69	EC_PRT vs NET.	149
5.70	EC_PRT vs EO_PRT (1).	149
5.71	EC_PRT vs EO_PRT (2).	150
A-1	Queries and Q-Expression-Sets	193
B-1	Insertion times.	195
B-2	Extraction times.	195
B-3	Query timings (ms) for the 10 MB data set (XBench).	195
B-4	Query timings (ms) for the 100 MB data set (XBench).	196
B-5	Query timings (ms) for the 1 GB data set (XBench).	196
B-6	Query timings (ms) for the recursive data sets.	197
B-7	Maintenance cost (ms) with change in query workload for partitioning.	197
B-8	Prefetching timings (300 MB).	197
B-9	Prefetching timings (600 MB).	198
B-10	Prefetching timings (1200 MB).	198

Chapter 1

Introduction

The tremendous growth of the World Wide Web was initially due to the ease with which authors could share their electronic documents. The Hypertext Markup Language (HTML) has traditionally been used to render documents in a visually appealing manner.

As the Web has matured, it has become more than a medium to present electronic documents to a wide audience. It has become a medium of communication not just for humans but also for applications. Over time, applications have changed from standalone programs to a collection of related but independent components. Communications between these components has become more and more difficult as each has grown on its own terms without standard interfaces leading to semantic interoperability problems. XML (eXtensible Markup Language) is an attempt to mitigate some of these problems [2].

Semantic interoperability between disparate applications requires that the type and structure of the communicated data be known to the programs. XML has been designed to do just this by making information self-describing. It was originally designed to meet the requirements of large-scale electronic publishing. However, it has found new uses as a standard for data representation and exchange on the internet. The idea behind XML is to have a self-describing format for data. It consists of tags that enclose data and define its meaning. These tags are not predefined and users are allowed to make up new tags for the describing data in their own application. This has given applications a standard mechanism for communication that is independent of the implementation specifics such as operating system, network infrastructure etc. of each application. As a result the use of XML has grown tremendously.

CHAPTER 1. INTRODUCTION

An associated benefit of XML's machine readability is the possibility that the vast amounts of data available on the net today can be turned into useful information. Current search engines are hampered by the fact that HTML, the language used to publish web sites, is intended for *presentation* or more specifically for laying out data in a visual appealing manner in a web browser. XML, on the other hand, allows the document author to prepare the document in a semantically meaningful manner. This opens up a whole range of possible applications in addition to more effective web searches. Consider the example shown in Figure 1.1(a). The visual representation in the form of an HTML page is not suitable for querying or exchanging between applications as it lacks information regarding the data it represents. The XML representation in Figure 1.1(b), on the other hand, abstracts away the underlying *information* making it easy for exchange and querying.

In addition to providing a suitable data exchange format, XML also represents a more flexible *semistructured* data model than the relational model. Relational database systems enforce an explicit and rigid schema on all stored data. This could lead to problems in the implementation of some applications.

- Schema or the structure that the data adheres to may change over a period of time. Data elements or the data types used to represent certain elements may themselves change. Traditional RDBMS do not handle frequent schema changes very well.
- Even if the data follows a fixed schema it could be possible that the schema itself has a very complex representation in terms of sets and relations. Some other structure (e.g. Tree) may be more appropriate. A complex schema would mean that querying the data would require the user to formulate unnecessarily complex queries.

In order to illustrate the above points consider an example from the bioinformatics domain. Figure 1.2 shows a sample entry in the SWISS-PROT database [29]. The relational representation of such an entry would encompass about 150-odd relations [29]. Formulating queries over such a fragmented representation is a daunting task. On the other hand, if XML is used as the data model

CHAPTER 1. INTRODUCTION

1.1a: HTML representation.

1.1b: XML representation.

Figure 1.1: Abstracting away information.

then the representation would be simpler to comprehend and a query language such as XQuery [2] can be used to formulate queries that are lot more intuitive. We will highlight this further in Chapter 5.

In addition to the representation of data with a complex structure, XML can also be used in scenarios that require a "super"-data model. Consider the problem of integrating heterogeneous data sources. In order to provide a common querying mechanism that incorporates these heterogeneous data sources a common data view has to be presented to the user. Such a data view must be flexible enough to represent all the underlying data sources correctly. XML presents an effective solution for this problem [32]. Several data integration applications have been based on the flexibility provided by XML [2]. Figure 1.3 shows the architecture of a hypothetical system integrating data from sources with different data structures.

CHAPTER 1. INTRODUCTION

```

ID 5EAS_TOBAC STANDARD; PRT; 548 AA.
AC Q40577;
DT 15-JUL-1998 (Rel. 36, Created)
DT 15-JUL-1998 (Rel. 36, Last sequence update)
DT 15-JUN-2004 (Rel. 44, Last annotation update)
DE Aristolchene synthase (EC 4.2.3.9) (5-epi-aristolchene synthase)
DE (EAS).
OS Nicotiana tabacum (Common tobacco).
OC Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
OC Spermatophyta; Magnoliophyta; eudicotyledons; core eudicots; asterids;
OC lamiids; Solanales; Solanaceae; Nicotiana.
OX NCBITaxID=4097;
RN [1]
RP SEQUENCE FROM N.A., AND SEQUENCE OF 56-73.
RC STRAIN=cv. NK326;
RX MEDLINE=93066390; PubMed=1438319; [NCBI, ExPASy, EBI, Israel, Japan]
RA Facchini P.J., Chappell J.;
RT "Gene family for an elicitor-induced sesquiterpene cyclase in
RT tobacco.";
RL Proc. Natl. Acad. Sci. U.S.A. 89:11088-11092\(1992\).
CC -!- FUNCTION: Catalyzes the cyclization of trans,trans-farnesyl
CC diphosphate (FPP) to the bicyclic intermediate 5-epi-
CC aristolchene sesquiterpenoid antifungal phytoalexin capsidiol.
CC -!- CATALYTIC ACTIVITY: Trans,trans-farnesyl diphosphate =
CC aristolchene + diphosphate.
DR EMBL; L04680; AAA19216.1; -. [EMBL / GenBank / DDBJ] [CoDingSequence]
DR PIR; T03714; T03714.
KW Lyase; Metal-binding; Magnesium; 3D-structure;
KW Direct protein sequencing.
FT ACT_SITE 273 273 Proton acceptor.
FT HELIX 104 116
FT TURN 117 118
FT HELIX 123 129
SQ SEQUENCE 548 AA; 62973 MW; 9FE1C59CF1A68BF1 CRC64;
MASAAVANYE EEIVRPVADF SPSLWGDQFL SFSIDNQVAE KYAKEIEALK EQTRNMLLAT
HLKSPLREQV THALEQCLHK GVPRVETRFV ISSIYDKEQS KNNVLLRFAK
//

```

Figure 1.2: Semistructured data - SWISS-PROT.

The popularity of XML and its usefulness as a data model have already been highlighted in this section. The next issue, therefore, is the storage and querying of XML. Existing work in this area is quite extensive and will be discussed in detail in Chapter 2. However, there is still tremendous room for improvement in the storage and query performance aspects. This thesis presents novel work in this area.

1.1 Storing and Querying XML data

Here, we will first provide a brief overview XML data management. Then, some of the research issues in XML storage will be presented. Existing research

CHAPTER 1. INTRODUCTION

Figure 1.3: Heterogeneous data integration.

Figure 1.4: Architecture.

related to XML storage and querying will be presented in Chapter 2.

Based on the architecture of existing database systems [22] the following can be identified as the main components of an XML database system. Figure 1.4 shows an overview of the architecture.

The *Storage Manager* implements the storage structures such as the page files etc. and the index structures to speed up the process of accessing data. It manages the interaction between the underlying secondary storage and main memory buffers.

The *Query Processor* consists of two sub-components. The *query compiler* translates a textual XQuery query to a query plan. It parses the query, checks for syntactic correctness and optimizes the initial query plan. The *execution engine* executes each step in the query plan by sending commands to the storage

manager.

The *Application Interface* provides an interface for applications to access the database system. This could be as simple as a *text console* to enter an XQuery query or a *complex GUI* to construct visual queries that are translated to the textual representation and then sent to the query processor.

There are three main approaches to the storage and querying of XML data - using native XML storage, using relational database system and using object-oriented database systems. The main differences across these approaches are in the storage manager and query processor.

1.1.1 Storage Approaches

We present a brief overview of the native and relational storage approaches. A more detailed discussion will be presented in Chapter 2 where existing work will be highlighted. OODBMS-based approaches are not discussed as OOBDBMS-based XML storage has not proven popular due to the problems associated with object-oriented database systems [37].

Native XML Systems

Native XML database systems are implemented specifically for XML data. They generally use a proprietary storage format such as indexed files or specialized index structures some of which will be discussed in Chapter 2.

The main advantage of using native XML systems from a storage perspective is that document insertion and retrieval is much faster. There is no overhead associated with mapping XML to suite the needs of the underlying storage model.

However, native XML systems are a fairly new development and therefore, are not as scalable or robust as existing RDBMS technology. Also, most existing IT infrastructure is built on relational database systems. Reusing this technology would be a much better option, commercially, than building a native XML system from scratch.

Some examples of research and commercial Native XML products can be found at [11].

CHAPTER 1. INTRODUCTION

Storing XML in an RDBMS

Relational systems first convert XML data to a format that can be stored and queried as a group of relations. Again, there are two approaches within this classification - schema-conscious and schema-oblivious. These will be discussed in Chapter 2.

The storage manager component is provided by the RDBMS. This is a major advantage of this approach. The simplicity, stability, and expressiveness of existing relational database technology greatly adds to the scalability and robustness of the XML storage system. The main issue that needs to be explored here is the representation of XML in a relational format that allows for efficient querying.

The main disadvantage is the slow insertion and retrieval of XML documents. This is because XML documents have to be mapped to suite the relational model first, thus introducing an performance overhead. In addition, the storage size can be a lot more than the size of the original XML documents as most relational approaches store every node/ledge in the XML document and introduce additional index structures to enable efficient query processing. Another disadvantage of most relational approaches is their inability to adapt to change in either the structure or content of the stored XML document. Furthermore, relational storage approaches may not perform as well as native XML systems for some queries [56].

Several relational storage approaches have been discussed in literature. The major ones are XRel [57], XParent [25], Shared-Inlining [49], Stored [48] and LegoDB [10]. These are discussed in greater detail in Chapter 2.

1.1.2 Querying XML

A brief overview of query processing in native and relational storage systems is presented. A detailed discussion with examples of existing systems will be presented in Chapter 2.

Native XML Systems

In native XML systems, the query processor generates query plans in terms of the operations allowed by the storage manager. Query optimization by enumerating query plans and selecting the optimal plan is also handled by this module.

Native XML systems present the most flexible option for query processing. Since they are not hindered by a model-mismatch between the storage and query processing layer, implementing all query constructs should be possible. In addition, query optimization can be tailored to XML data resulting in faster query execution.

However, native XML systems are not suited for transaction based queries i.e., queries that involve inserts and updates. Also, query optimization techniques are nowhere as mature as in relational systems. The tremendous effort required to implement a query system from scratch is another disadvantage.

RDBMS-based approaches

The query processor has to essentially translate the incoming XML based query to one or more SQL queries depending on the storage approach in use.

The main advantage is the simplicity of implementation as compared to a native XML approach. In addition, the underlying relational query processor can handle the task of query optimization to a certain extent.

The main disadvantage is that most translation approaches do not generate an optimal translation [52]. In addition, once the translation is done the underlying query optimizer may not be able to generate an optimal query plan as the SQL query is "non-standard" as compared to traditional relational queries [25] due to the model-mismatch between the XML and relational data models. The translation of some types of queries may become unnecessarily complex [48] as well. These issues are highlighted further in Chapters 4 and 5.

1.2 Research Issues and Challenges

Based on the architecture of an XML database management system and current approaches to storing and querying XML data the following research issues can be identified.

CHAPTER 1. INTRODUCTION

(i) Issues in storing XML.

- Development of scalable storage managers and suitable index structures if a native XML implementation is followed.
- Development of an appropriate relational schema that allows for efficient storage in an RDBMS and is amenable to efficient query processing.

(ii) Issues in querying XML.

- Query optimization in the context of a native XML system.
- Suitable translation algorithms to translated queries in an XML query language to SQL.
- Query optimization in the context of a relational approach to storing XML that overcomes the inadequacies of the relational query optimizer.

The focus of this thesis is the storage and querying of XML in an RDBMS. Therefore, we will further elaborate on the associated research issues of *designing an appropriate relational schema, query management and query optimization*.

1.2.1 Appropriate Relational Schema

Several schema to store XML have been proposed and some of these will be discussed in Chapter 2. Essentially, the following aspects need to be considered when designing an appropriate schema.

- Efficient insertion and retrieval of XML documents.* The schema should minimize the amount of shredding (conversion of an XML document into tuples of a relation or set of relations) needed to store an XML document. This would reduce the performance overhead when storing or retrieving XML.
- Size.* When XML data is converted to a relational format there is a possibility that the final storage size will be more than the size of the original data. This depends on the schema and the indexes required. A well-designed schema will try to minimize the storage size.

CHAPTER 1. INTRODUCTION

- (iii) *Adaptability*. Stored XML data may undergo change over a period of time. This could be change in the structure or just the values of elements and attributes. The relational schema should adapt well to these changes.

The development of an appropriate schema can be investigated by following the above issues as guidelines.

1.2.2 Query Management

The users of an XML storage system will pose queries in an XML query language such as XQuery. Therefore, query translation needs to be considered when designing the storage mechanism. The relational schema should allow for translation to the corresponding SQL query in a flexible manner. In addition, query translation algorithms need to be developed. XQuery [2] has emerged as the standard XML query language. Therefore, efforts should focus on the translation of XQuery queries to SQL.

In spite of an efficient schema and translation algorithm, it is possible that the inadequacies of the relational query optimizer could effect query performance. This brings us to the issue of query optimization beyond what is provided by the underlying RDBMS.

1.2.3 Query Optimization

Query optimization for relational data is a well-explored topic and current relational database systems perform an excellent job of optimizing queries. So, using an RDBMS to store XML data should mean that query optimization will be taken care of. However, this is not the case. Query optimization in the context of a relational storage approach entails the following issues.

- (i) *Optimal translation*. Each XML query could possibly be translated to more than one SQL query. The translation mechanism should select the one that can be executed most optimally.
- (ii) *Optimal execution*. The relational query optimizer may be unable to optimize the translated SQL query. There is a need to formulate execution strategies that overcome this inadequacy.

With the above research issues identified we move on to a brief discussion of the contributions of our work and how they mitigate some of the problems discussed above.

1.3 Contributions

This thesis presents work in two main areas - storage of XML in an RDBMS and efficient query execution based on this storage strategy.

1.3.1 RDBMS-based Storage of XML

Realizing the problems associated with existing relational storage strategies we present two systems for the efficient storage of XML - SUCXENT and SUCXENT++. These are discussed in detail in Chapter 3.

SUCXENT is a preliminary attempt at developing a suitable relational schema. Our experiments show that it performs better than most existing approaches such as XRel [57] and XParent [25]. However, it performs significantly worse than some other approaches such as Shared-Inlining [49]. Also, its storage space requirements and insertion times, though less than XRel and XParent, are still significantly high.

SUCXENT++ incorporates a schema that is significantly more efficient than SUCXENT both in terms of storage, insertion/extraction and querying. In addition, our experiments indicate that it performs substantially better than most existing relational storage approaches including Shared-Inlining.

1.3.2 Querying XML

We have implemented algorithms to translate XQuery queries to SQL queries in SUCXENT and SUCXENT++.

In addition to translation, we have also implemented optimization techniques to overcome the inadequacies of the relational query optimizer. One approach has been to generate multiple SQL queries instead of a single query. Details of this approach will be presented in Chapter 5. Another approach we have followed is query workload-based partitioning. Traditionally, data partitioning has been done manually and has been independent of the queries executed on

the system. We present an approach in Chapter 5 that partitions data based on the query workload.

We also present a non-traditional query optimization technique inspired by the work on speculative query processing in [40]. There are application scenarios where the Application Interface needs to be a GUI. One example would be a system to query biological data as an average biologist cannot be expected to learn a query language. The latency offered by GUI-based query formulation can be utilized to pre-fetch portions of the query results. We present our work in this direction in Chapter 5.

1.4 Report Organization

The rest of the report is organized as follows. Chapter 2 presents a survey of current work in the field of XML Database Systems. We will highlight the high-level differences and similarities with respect to our work deferring detailed comparisons to later chapters. Chapter 3 will present the relational schemas of SUCXENT and SUCXENT++ highlighting the relative merits of SUCXENT++. We will also present results of a performance evaluation to compare these systems against existing approaches in terms of storage size and insertion/extraction times of XML documents. Our experiments indicate an improvement of up to 4 times for storage space requirements and up to 11 times of document insertion time.

Chapter 4 will present our work on querying XML. Query translation approaches in SUCXENT and SUCXENT++ will be presented. Comparisons with existing systems with respect to query performance will also be presented. Our experiments indicate an improvement of up to 37 times over existing schema-oblivious approaches. Chapter 5 will extend the discussion in Chapter 4 by presenting our work on query optimization. Query optimization results in an improvement in query performance of up to 40 times. We will also present our work on GUI-driven prefetching using a system to query biological data as an example.

We conclude in Chapter 6 by summarizing the contributions of this dissertation and highlighting promising directions for future work in this field.

Chapter 2

Related Work

In this chapter we discuss prior works in topics related to this dissertation. Section 2.2 presents existing work on relational storage of XML and Section 2.3 discusses existing native XML storage systems. Section 2.4 discusses work on query translation. In Section 2.5, existing work on XML query optimization is briefly described. This includes a brief overview of data partitioning, prefetching and GUI-latency speculative query processing. The chapter only provides an overview of related work and the detailed comparisons with our work can be found in the relevant chapters.

2.1 XML Storage

We begin by presenting existing work on storage of XML using a relational database system. We then discuss work related to systems built from scratch specifically for the storage of XML i.e., native XML storage systems.

2.2 XML in a Relational Database System

There is a substantial body of work on using relational databases to store XML documents. The various approaches differ in which meta-data they use (i.e., schema or schemaless); how the relational configuration is generated; and which information is preserved in the relational side. Table 2.1 gives the summary of existing approaches.

CHAPTER 2. RELATED WORK

Techniques	Schema-oblivious	Cost-based	Order Pre-serving	Class of XML schema considered	Class of XML Query considered
STORED [48]	Yes	No	Yes	All	STORED
Edge [20]	Yes	No	Yes	All	Path expressions
Interval [45]	Yes	No	Yes	All	XQuery
XRel [57]	Yes	No	Yes	All	Path expressions
XParent [25]	Yes	No	Yes	All	Path expressions
[51]	Yes	No	Yes	All	Order-based queries
[58]	Yes	No	Yes	ll	Path expressions
SUCXENT++	Yes	No	Yes	All	XQuery
Inlining [49]	No	No	No	Recursive	-
LegoDB [10]	No	Yes	No	tree	XQuery
Oracle XML DB [47]	Yes	No	Yes	Recursive	SQL/XML restricted XPath
DB2 XML Extender [46]	Yes	No	Yes	non-recursive	SQL extensions through UDFs

Table 2.1: Summary of XML storage and querying techniques.

2.2.1 Schema-oblivious Approaches

Techniques which store XML documents in *generic* (pre-defined) relational tables are called schema-oblivious. One of the first proposals for schema-oblivious mapping of XML documents was the Edge approach [20]. In this approach, the input XML document is viewed as a graph and each edge of the graph is represented as a tuple in a single table. In a variant known as the Attribute approach, the edge table is horizontally partitioned on the tag name yielding separate table for each element/attribute. Two other alternatives, the Universal table approach and the Normalized Universal approach are proposed but shown to be inferior to the other two. In this approach, resolving ancestor-descendant relationships requires the traversal of all the edges from the ancestor to the descendant (or vice-versa). Thus it is an expensive approach as it typically requires many joins for navigating and/or reconstructing the document. Note that the Edge approach uses *recursive SQL queries* using the SQL99 *With* construct to evaluate recursive XML queries.

In STORED [48], given a semistructured database instance, STORED mapping is generated automatically using data mining techniques - STORED is a declarative query language proposed for this purpose. This mapping has two parts: a relational schema and an *overflow graph* for the data not conforming to the relational schema. STORED can be considered as schema-oblivious approach as the data inserted in the future is not required to conform to the

CHAPTER 2. RELATED WORK

derived schema. Thus, if an XML document with completely different structure is added to the database, the system sticks to the existing relational schema without any modification whatsoever. In STORED, an algorithm is outlined for translating an input STORED query into SQL. The algorithm uses inversion rules to create a single canonical data instance, intuitively corresponding to a schema. The structure component of the STORED query is then evaluated on this instance to obtain a set of results, for each of which a SQL query is generated incorporating the rest of the STORED query. However, similar to the Edge, in this approach it is necessary to perform one join per step in the path expression during query translation.

The system proposed by Zhang et al in [58] labels each node with its pre-order and postorder traversal numbers. Then, ancestor-descendant relationships can be resolved in constant time using the property $\text{preorder}(\text{ancestor}) < \text{preorder}(\text{descendant})$ and $\text{postorder}(\text{ancestor}) > \text{postorder}(\text{descendant})$. However, it still results in as many joins as there are path separators.

To solve the problem of multiple joins, XRel [57] stores the path of each node in the document. For each element, the path id corresponding to the root-to-leaf path as well as an interval representing the region covered by the element are stored. Then, the resolution of path expressions only requires the paths (which can be represented as strings) to be matched using string matching operators. However, the query translation algorithm in XRel is correct for nonrecursive data sets - it turns out that it does not give the correct result when the input XML data has an ancestor and descendant element with the same tag name [31]. Moreover, this approach still makes use of the containment property mentioned above to resolve ancestor-descendant relationships. It involve joins with θ ($<$ or $>$) operators that have been shown to be quite expensive due to the manner in which an RDBMS processes joins [58]. In fact, special algorithms such as the Multi-predicate merge sort join algorithm [58] have been proposed to optimize these operations. However, to the best of our knowledge there is no off-the-shelf RDBMS that implements these algorithms as the issue of how we extend the relational engine to identify the use of these strategies is an open problem. In particular, the question of how the optimizer maps SQL operations into these strategies needs to be addressed.

CHAPTER 2. RELATED WORK

XParent [25] solves the problem of 0-joins by using an **Ancestor** table that stores all the ancestors of a particular node in a single table. It then replaces 19-joins with equi-joins over this set of ancestors. However, this approach results in an explosion in the database size as compared to the original document. The number of relational joins is also quite substantial. XParent requires a join between the **LabelPath**, **DataPath**, **Element** and **Ancestor** tables for each path in the query expression. The joins are quite expensive especially when the Ancestor table is involved as it can be quite large in size. Note that XParent and XRel handle recursive queries like any other query.

In [51], the focus is on supporting order based queries over XML data. The schema assumed is a modified Edge relation where the path id is stored as in XRel, and an extra field for order is also stored. Three schemes for supporting order are discussed. Algorithms for translating order-based path expression queries into SQL are also provided. As this approach is based on the Edge and XRel, it suffers from the same limitations as discussed above.

In dynamic intervals approach [45], all XML data is stored in a single table containing a tuple for each element, attribute and text node. For an element, the element name and an interval representing the region covered by the element is stored. Analogous information is stored for attributes and text nodes. In order to distinguish children from descendants, a level number is recorded with each node. This approach supports a larger fragment of XQuery with arbitrarily nested FLWR expressions, element constructors and built-in functions including structural comparisons. Special purpose relational operators are proposed for better performance. We note that without these operators, the performance is likely to be inferior even for simple path expressions. As an example, using their technique, the path expression `/site/people` is translated to an SQL query involving five temporary relations created using the *With* clause in SQL99, three of which involve correlated subqueries [31]. Hence, without modifications to the relational engine, its performance may not be acceptable.

In Oracle XML DB [47] and IBM DB2 XML Extender [46], a schema oblivious way of storing XML data is provided, where the entire XML document is stored using the CLOB data type. Hence, evaluating XML queries in this case is similar to XML query processing in a native XML database. Also, many

CHAPTER 2. RELATED WORK

types of XML queries suffer from poor performance due to the treatment of XML documents as CLOB.

SUCXENT++ is different from existing approaches in that it only stores leaf nodes and their associated paths. For each level in an XML document, we store an attribute called *RValue*. Rather than storing the ancestor-descendant and parent-child of all nodes in the XML document, we store only the leaf nodes and their corresponding values along with the root-to-leaf paths in the document. Additionally, for each leaf node we store two additional attributes called *BranchOrder* and *BranchOrderSum*. These attributes along with the *RValue* enable us to efficiently check whether the level of the nearest common ancestor of a pair of relevant leaf nodes satisfies the query constraints. This reduces the storage size significantly as well as the number of joins needed to be executed in the translated SQL queries. In addition, we propose optimization techniques that enable the underlying relational query optimizer to generate near-optimal query plans for our approach, resulting in a substantial performance improvement.

2.2.2 Schema-conscious Approaches

Departing from generic mapping as discussed above, several specialized strategies have been proposed which make use of schema information to generate efficient mappings. These approaches are called schema-conscious approaches.

In [49], three techniques for using a DTD to choose a relational schema are proposed - basic inlining, shared inlining, and hybrid inlining. The main idea is to inline all elements that occur at most once per parent element in the parent relation itself. This is extended to handle recursive DTDs.

LegoDB [10] takes a cost-based approach to derive a mapping that best suits a given application - characterized by a schema, query workload and document samples. LegoDB uses the information in the XML schema to derive several possible mapping alternatives, and selects the one which leads to the lowest cost for executing a given query workload over sample documents. Compared to Shared Inlining, LegoDB system exploits a richer set of mapping primitives. In addition to parent-child relationships, LegoDB also takes into account additional schema constructs such as choice and repetition, and it allows multiple mapping functions for a given construct.

CHAPTER 2. RELATED WORK

Unlike the schema-oblivious approaches, schema-conscious techniques have focused on structural and constraint mapping, often ignoring the order among elements. Because these techniques ignore order, the resulting mapping are lossy [31]. For example, the mapping strategies in [49] do not allow mapped documents to be faithfully reconstructed. Furthermore, schema-conscious strategies have to treat recursion in both schema and queries as special cases. In [30], the authors propose a generic algorithm to translate recursive XML queries for schema-conscious approaches using the SQL99 `WITH` construct. However, no performance evaluation of the resulting SQL queries is presented and it is assumed that schema-conscious approaches will outperform schema-oblivious approaches. SUCXENT++ maintains document order and also treats recursive XML queries like any other queries.

2.3 Native XML Storage

The term "native XML database" (NXD) is deceiving in many ways. In fact many so-called NXDs aren't really standalone databases at all, and don't really store the XML in true native form (i.e. text). To get a better idea of what an NXD really is, let's take a look at the NXD definition offered by the XML:DB Initiative [4]. A native XML database:

- Defines a (logical) model for an XML document – as opposed to the data in that document – and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PC-DATA, and document order. Examples of such models are the XPath data model, the XML Infoset, and the models implied by the DOM and the events in SAX 1.0.
- Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
- Is not required to have any particular underlying physical storage model. For example, it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files.

CHAPTER 2. RELATED WORK

By this definition some of the storage techniques discussed in the previous section would also qualify as Native XML Databases. However, we will use the term to solely represent storage techniques that do not make use of traditional relational database systems.

There are several commercial native XML storage systems. A comprehensive listing is provided at [11]. We will enumerate some of the research projects that deal with native storage for XML.

- Lore [36]. The Lightweight Object Repository (LORE) is a Database Management System designed specifically to manage semistructured data. In general, LORE attempts to take advantage of structure where it exists, but also handles irregular data as gracefully as possible. LORE'S data model is the Object Exchange Model (OEM) discussed in Chapter 1. Another novel aspect of LORE is the use of DataGuides [35].
- TIMBER [24]. The system is based on a bulk algebra for manipulating trees. New access methods have been developed to evaluate XML queries. These access methods are discussed in greater detail in Section 2.5 as they relate more to access algorithms rather than storage. The authors have also proposed new cost estimation and query optimization techniques some of which are discussed in Section 2.5. The key intellectual contribution of this system is a comprehensive set-at-a-time query processing ability in a native XML store with all the standard components of relational query processing, including algebraic rewriting and a cost-based optimizer.
- Kweelt (<http://kweelt.sourceforge.net/>) is a modular and extensible framework to query XML. It provides an implementation of Quilt Query Language. The main objective of this system was to decouple the query engine from the storage module. The authors have implemented two different backends to demonstrate this - a DOM backend and a relational database backend.
- Xyleme [5] is a dynamic warehouse for XML data. It uses the Natix [28] native XML storage manager as its data repository. Natix uses a hybrid

approach to store XML data, where some data is stored as trees and the rest is kept as byte streams. Xyleme implements some tree matching mechanisms to accelerate query processing.

We have only discussed complete native storage systems. There is significant work on individual components that form parts of a storage system such as index structures and access algorithms, query optimization etc which is discussed in the next section.

2.4 Query translation

Several XML-ro-SQL query translation approaches have been proposed. Some of these deal with XPath expressions, subsets of XQuery, XML-QL etc. Here, we will focus on a few representative approaches.

- (i) *Query translation in XRel*. XRel identifies a core part of XPath called XPathCore. A detailed algorithm is presented to translate such expressions into SQL. The translated SQL query conforms to the XRel schema. The main advantage the XRel has over previous translation schemes is the fact that a path id corresponding to the root-to-leaf path is stored for each element (as discussed in Section 2.2). There is no need to evaluate a join for each step of the path expression. Our work also follows a similar approach to storage and query translation.
- (ii) *Query translation in ordered XML*. In [51], the authors present an algorithm to translate order based XPath expressions into SQL. Again, this translation conforms to the schema-oblivious relational schema proposed by the authors. However the authors do highlight how their translation approach could be applied to schema-conscious relational schema. The algorithm allows for the translation of each axis in XPath and positional predicates.
- (iii) *Translation of recursive XML queries*. In [30], the authors propose an approach to translate recursive XPath expressions (with '//')over recursive schema when a schema-conscious storage approach is followed. However,

the proposed algorithm is not limited to one particular schema-conscious approach. It is able to translate recursive XPath expressions for a range of schema-conscious techniques. The main feature is that the algorithm produces a single query irrespective of the complexity of the query or the schema. It uses the SQL99 with clause to generate recursive SQL queries and demonstrates that this is sufficient to handle recursive path expressions over recursive (or non-recursive) XML schema.

2.5 Optimizing XML query processing

Work in this areas has focused on novel index structures/algorithms, optimal query plan selection, and selectivity estimation.

2.5.1 Index Structures and Algorithms

XML queries typically specify patterns of selection predicates on multiple elements that have some specified tree structured relationships. The primitive tree structured relationships are parent-child and ancestor-descendant, and finding all occurrences of these relationships in an XML database is a core operation for XML query processing. Several index structures and algorithms have been proposed to optimize this operation. Most of these follow the numbering scheme discussed in [58].

Multi-predicate merge join

Zhang et al. established in [58] that relational join algorithms are not appropriate for containment detection as containment detection involves θ -joins. Instead, they proposed the MPMJN algorithm that skips over several comparisons that traditional merge-sort join performs and present an implementation using Berkeley DB [1] as the repository. They demonstrate that this results in a tremendous gain in performance of several orders of magnitude.

Structural join algorithms

The authors propose two families of structural join i.e., set-at-a-time evaluation of ancestor-descendant containment, algorithms in [9]. The Tree-Merge

CHAPTER 2. RELATED WORK

algorithms are an extension of the MPMJN. The time complexity of these algorithms is $O(|AList| \times |DList|)$ where $|AList|$ and $|DList|$ are the sizes of the ancestor and descendant node lists respectively. One key point to note here is the set-at-a-time detection of containment. Lists of possible ancestor and descendant nodes are involved in the process instead of single ancestor-descendant nodes.

The authors also propose another family of algorithms, Stack-Tree, that extend Tree-Merge algorithms using a stack. The time complexity of these algorithms is $O(|AList| + |DList| + |OutputList|)$ where $|OutputList|$ is the size of the final list of detected ancestor-descendant pairs. Note that Tree-Merge algorithms perform fewer comparisons than the traditional merge-sort algorithm for containment detection and Stack-Tree algorithms perform even fewer.

The above algorithms do not make use of any special index structures. They work on the assumption that a list of possible ancestors and descendants has been provided. Their objective is to optimize the detection of ancestor-descendant pairs in this list.

Structural join indexes

The work discussed here builds on the structural join algorithms by enhancing them with the use of index structures such as B+-trees and R-trees to reduce the number of θ comparisons between the ancestor/descendant lists even further.

Non-indexed structural joins algorithms sequentially scan the ancestor/descendant lists. In [17] the authors propose the use of B+-trees to index the ancestor/descendant list based on their number assignment. Then, it is shown that a large number of θ comparisons can be skipped, i.e., several nodes in the ancestor/descendant lists not present in the final output list can be skipped over by using the B-t-tree index. The authors show an improvement in performance of up to 60% over the non-indexed structural joins algorithms.

Jiang et al. propose XR-tree (XML Region Tree), a dynamic external memory index structure specially designed for strictly nested XML data in [26]. The unique feature of XR-tree is that, for a given element, all its ancestors (or descendants) in an element set indexed by an XR-tree can be identified with

CHAPTER 2. RELATED WORK

Name	Uses index?	Time complexity	Remarks
Multi-predicate Merge Join	No	$O(AList \times DList)$	First work in this area
Tree-Merge	No	$O(AList \times DList)$	Extends MPMJN.
Stack-Tree	No	$O(AList + DList + OutputList)$	
B+	Yes	$O(AList + DList + OutputList)$	Uses B+-trees
XR-Tree	Yes	$O(\log_F N + R)$	Special index structure
TwigStack	Yes	$O(AList + DList + OutputList)$	Independent of the sizes of intermediate results.

Figure 2.1: Structural-join techniques.

optimal worst case I/O cost of $O(\log_F N + R)$ where F is the fan-out the XR-Tree, N is the number of elements and R is the output size. They also propose a new structural join algorithm that can evaluate the structural relationship between two XR-tree indexed element sets by effectively skipping ancestors and descendants that do not participate in the join. The authors show that the XR-tree based join algorithm significantly outperforms previous algorithms.

In [27] the authors propose a novel holistic twig join algorithm, **TwigStack**, for matching an XML query twig pattern. The technique uses a chain of linked stacks to compactly represent partial results to root-to-leaf query paths, which are then composed to obtain matches for the twig pattern. When the twig pattern uses only ancestor-descendant relationships between elements, TwigStack is linear in the sum of sizes of the input lists and the final result list (much like Stack-Tree), but independent of the sizes of intermediate results. This work is extended using B+-trees to perform twig matching in sub-linear time.

The next section will briefly describe some of the techniques used to optimize query execution in a manner orthogonal to the approaches discussed till now.

2.5.2 Query Plan Selection

McHugh and Widom present optimization techniques used in Lore in [35]. Evaluation of regular expressions at runtime can be expensive. The authors propose compile-time expansion of regular path expressions based on a structural summary called DataGuides. This eliminates unnecessary database exploration at run-time. Two strategies are applied - path expansion and alternation elimination. Basically, the query is precomputed over the DataGuide, extracting all

CHAPTER 2. RELATED WORK

labelled paths conforming to the regular path expression. This work is similar to [34]. Here, the authors define a graph schema that describes partial knowledge of the graph structure. It is used to restrict a search to certain fragments of the graph.

Structural joins can be considered the core operation in XML query processing making structural join plan selection an important aspect of query optimization in an XML database. In [54], the authors introduce five algorithms for structural join order optimization for XML tree pattern matching. The Dynamic Programming (DP) algorithm is similar to the traditional relational optimization technique and selects the optimal solution. The Dynamic Programming with Pruning (DPP) algorithm prunes unpromising join plans in early stage. It finds a solution, identical to that found by DP, but at substantially lower cost. The authors demonstrate that more aggressive pruning is possible at the cost of the optimality. They propose two DPAP (Dynamic Programming with Aggressive Pruning) algorithms, each based on a different heuristic. A Fully Pipelined (FP) algorithm that considers the solution space of only fully pipelined plans is also proposed. Extensive experimental evaluation with a range of queries over a variety of XML data sets is presented. The authors demonstrate that many optimization assumptions valid for relational join selection do not hold in this case.

Estimating query result sizes

An important precursor to any query optimization approach is the derivation of accurate statistics about the data being queried.

In [16], the authors describe a technique to obtain query result size estimates. They propose a solution based on a novel histogram encoding of element occurrence position. Together with a novel position histogram join (pH-join) algorithm, the estimates of sizes for complex pattern queries, as well as simpler intermediate patterns can be easily evaluated.

In [41], the authors extend their earlier work on structural XSKETCH synopses [42] and propose an (augmented) XSKETCH synopsis model that exploits localized stability and value distribution summaries (e.g., histograms) to accurately capture the complex correlation patterns that can exist between and

CHAPTER 2. RELATED WORK

across path structure and element values in the data graph. A systematic XSKETCH estimation framework for complex path expressions with value predicates is developed and an efficient heuristic algorithm based on greedy forward selection for building an effective XSKETCH is proposed.

In [21], the authors define an XML statistics model. A system, StatiX, that implements this model is also proposed. StatiX leverages the XML Schema data model, schema rewriting transformations, and histograms to provide simple, concise, flexible, and scalable data summaries for query selectivity estimation. In addition, the authors propose an implementation that integrates with existing XML technology such as schema validators. The authors demonstrate the effectiveness of StatiX by using it in the implementation of LegoDB (discussed earlier).

Our work differs from the above query optimization techniques in two critical aspects. First, we focus on optimization in the context of a relational storage approach for XML. Second, one of our optimization techniques - GUI-based prefetching - is orthogonal to the optimization techniques presented above.

2.6 Partitioning and Prefetching

Several commercial database implementations use partitioning to improve the performance of relational data warehouses [18]. However, the focus of our work is workload-based partitioning. There is a significant body of work in existing literature that deals with automatic query-based partitioning [14, 38, 39, 23, 8].

Partitioning of relational data can be done in two ways. *Vertical partitioning* splits a table into two or more tables each of which has a subset of *columns* in the original table. As most queries access only a subset of the columns, vertical partitioning can help by reducing the data that needs to be scanned to answer a query. *Horizontal partitioning* splits a table into two or more tables each of which has a subset of *rows* in the original table. The evaluation of predicates of a query requires the scan of only a subset of the rows. Horizontal partitioning helps by reducing the number of rows that need to be scanned.

In [38] the authors present a graph-based algorithm to determine vertical partitions based on the transactions executed on the database. First, an

CHAPTER 2. RELATED WORK

attribute-usage matrix is generated. This matrix represents the use of the attributes of a relation in important transactions. The importance of a transaction is determined by its frequency. From this matrix an attribute-affinity matrix representing the access of a *set* of *attributes* is generated. Attribute-affinity measures the extent to which two attributes in a relation are related based on their collective usage in transactions. The authors treat this matrix as a graph and generate the vertical partitions by identifying cycles. The complexity of the algorithm is $O(n^2)$ which is an improvement over previous work in identifying vertical partitions.

The authors have taken a novel graph-based approach and demonstrate the superior time-complexity characteristics of their algorithm. However, the authors do not consider intra-query dependencies. Attribute-affinity is determined solely on the basis of inter-query usage. Also, no performance evaluation is presented to quantify the gains offered by this partitioning approach. In addition, the authors do not provide any data on the maintenance costs associated with this partitioning approach.

In [23], the authors present an automated approach to partition data horizontally across multiple independent nodes in a shared-nothing parallel database system. The authors implement their approach on DB2. The cost estimates of the built-in relational query optimizer are used as the metric to determine optimal partitioning. The query optimizer is extended to operate in an RECOMMEND mode and EVALUATE mode. In the RECOMMEND mode, the optimizer recommends good candidate partitions. Each query is evaluated and a list of partitions that could improve the performance of the query is generated. Certain heuristics are used to prune the size of this list. Next, execution plans based on these partitions are generated. The query optimizer then proceeds to evaluate these plans as it would normally. The candidate partitions are selected based on the plan considered optimal by the query optimizer. In the EVALUATE mode, the non-partitioned tables in the queries are replaced with the as yet un-materialized partitions. Next, a query plan based on the modified query is generated and the cost of the plan is measured. A rank-based search is conducted through the candidate partition space and lowest rank candidate

CHAPTER 2. RELATED WORK

evaluated within a user-specified time is taken as the final partition configuration.

This work exploits the sophisticated cost model of the built-in query optimizer. It extends the cost model to incorporate cost calculation based on candidate partitions. The authors demonstrate that this approach is significantly better than prior heuristics-based approaches that do not utilize the functionality offered by the query optimizer. However, the focus of their work is on shared-nothing databases. Also, no performance results quantifying the gains in query execution time or the effort required to maintain the partitions are shown.

In [39], the authors describe AutoPart - an algorithm that generates both vertical and horizontal partitions based on a representative workload. The notion of a Query Attribute Set (QAS) is introduced. QAS for a query with respect to a relation is the subset of the relation's attributes accessed by the query. AutoPart then proceeds as follows. First, the algorithm generates *atomic vertical partitions*. Atomic partitions are mutually disjoint and no query accesses a subset of any atomic vertical partition. Next, the atomic fragments are combined to form *composite fragments* which reduce the joining overhead for queries but increase the I/O cost. This is followed by horizontal partitioning based on the query workload predicates to reduce the size of the composite partitions and generate *categorical partitions*. Several iterations of combining atomic fragments are done and the resulting partitions evaluated using the relational query optimizer or a cost model developed by the authors. The partition that provides the lowest query execution cost for all queries is selected. The authors present performance results that show the benefit of partitioning - both in terms of space and query execution.

This work, unlike the systems discussed till now, combines vertical and horizontal partitioning to generate more optimal partitions. In addition it considers intra-query dependencies to generate horizontal partitions and utilizes the built in query optimizer to evaluate the cost of query execution using a given partitioning choice. However, it assigns equal weight to all queries. Also, no data indicating the cost of maintaining the partitions is presented.

CHAPTER 2. RELATED WORK

In [8], the authors present an approach that combines horizontal and vertical partitioning taking both performance and manageability into account. First, vertical partitions - called column groups - are identified. The authors present an approach to restrict the number of column groups generated for a given workload. This reduces the number of combinations that need to be considered for the next step. Candidate partitions are selected based on their suitability for queries in the workload as estimated by the built-in query optimizer using a Greedy Search algorithm [7]. In the next step, candidate partitions are merged to prevent a design that is over-specific to the given workload. The authors present several merging strategies that minimize the cost of this operation. Then, an enumeration strategy is used to generate the final partition from the set of candidate partitions given the constraint that partitions should be aligned. Alignment implies that partitions of a given table are identical. This eases partition maintenance. The authors present two enumeration approaches - Eager and Lazy - that are used to generate aligned partitions from the candidate partitions.

This is the first work that combines performance and manageability. Techniques that prevent over-specialized partitions are also presented. Performance results demonstrate the effectiveness of an integrated approach to generating aligned partitions that incorporates column-group pruning and merging. However, the authors do not present results that demonstrate the costs associated with partition maintenance.

Prefetching and the associated technique of caching have been employed in several scenarios. Modern microprocessors prefetch and/or cache memory blocks based on location similarity. Operating systems retrieve file blocks adjoining those being currently accessed. These two techniques have also been employed extensively in modern database systems.

To the best of our knowledge, caching - though investigated extensively in relational database systems - is a relatively new area of research for XML data. In [15], the authors describe a semantic cache system for XML data. Cost of executing new queries is minimized by using cached queries whenever possible. The authors first normalize the given XQuery query to separate its pattern

CHAPTER 2. RELATED WORK

matching and restructuring components. Then, the containment mappings between the new and cached queries are analyzed. A new query based on the cached query is written using the containment mappings.

In [55], the authors present an approach to discover frequent XML query patterns and cache their results. An algorithm, FastXMiner, to mine XML query patterns is presented. The authors prove that only a small subset of the generated candidate patterns need to undergo expensive tree containment tests. Results are presented to demonstrate that caching frequent query patterns provides significant performance improvements.

Caching techniques mentioned above operate on the final query and do not take into account the individual steps in query formulation. In our approach, partial queries are materialized at each formulation step and the latency offered by GUI-driven query formulation is utilized to generate the materializations.

2.7 GUI-Latency Driven Query Optimization

There are two aspects to work in this area - a GUI-based query system and query optimization that utilizes the latency offered by such a system. Both are discussed below.

We have come across only one other visual XQuery query formulation system, XQBE (XQuery By Example) [12]. XQBE is inspired by earlier work on QBE (Query By Example) [59] for SQL. XQBE represents XQuery constructs with geometrical shapes and connectors. Both the input and the output are shown as diagrams that consist of these shapes and connectors. In addition, the input diagram also shows the query constraints. XQBE allows for the representation of various types of FLWR queries as opposed to our implementation that only allows for FWR queries. In addition, XQBE supports element renaming and projection which our implementation lacks. However, the interface expects the user to be familiar with the tree structure of the document being queried. In our implementation, the user is already presented with this information and only needs to know the name(s) of the element(s) he wants to query. Also, the various geometrical shapes and connectors used to represent XQuery constructs can be confusing as they appear to be quite similar to each other

CHAPTER 2. RELATED WORK

and have only minor visual differences. The authors have implemented a Java version of XQBE which is available at [12].

In [40], the authors present an approach that combines speculation with visual query formulation. Their approach is based on a visual query interface that monitors the construction of a query and takes advantage of the user 'think time'. In particular, based on the features of the partial query specified at any point, the interface prepares the database by issuing asynchronous manipulations to it that are likely to make the final query (or even queries further into the future) more efficient. Furthermore, the interface applies machine learning techniques on past user actions and builds a user-behavior model that guides speculation and deals with future uncertainty. The authors formalize speculative query processing as an optimization problem and derive algebraic properties of the corresponding cost model that are sufficient to address the complexities of the particular optimization. The authors have implemented the framework on top of an existing commercial database system and have evaluated its effectiveness experimentally, with actual user traces. The results show that speculation outperforms normal query processing, reducing query execution time by an average of 35% and achieving performance improvements of more than 90% on certain queries.

The method described above is for relational data and incorporates speculation where the final query (or sub-queries that will be present in the final query) is predicted based on the user's usage profile. Several machine learning techniques are employed to perform the prediction. Our work, on the other hand, focuses on XML data. Though, our underlying storage system is relational, the data is semistructured in nature and the queries are presented in XQuery. We employ deterministic prefetching and do not speculate on the final form of the query. This could result in a less than maximum gain in certain cases but there are no penalties. Speculation can lead to execution time penalties when the prediction is incorrect.

This chapter has provided a brief overview of XML storage and query processing techniques. The next chapter presents our work on XML storage and provides a detailed juxtaposition with existing work including a performance comparison.

Chapter 3

Relational Schema Design

3.1 Introduction

As mentioned in the previous chapter, the use of relational databases to store XML data has gained in popularity due to the simplicity, stability, and expressiveness of relational databases. Several recent projects have dealt with the issue of storing XML data in an RDBMS. However, there is still substantial room for improvement, most importantly in terms of query processing speed. This is especially true for schema-oblivious approaches which have been shown to perform much worse than schema-conscious approaches[52].

In this chapter, two schema-oblivious approaches are presented. The first of these two approaches, SUCXENT (Schema Unconscious XML Enabled System - pronounced "succinct"), provides significant improvement in performance over existing schema-oblivious approaches. However, it still has some drawbacks that hinder query execution performance which will be discussed in the next chapter. The second approach, SUCXENT++, provides tremendous improvement over the performance of SUCXENT in terms of insertion/extraction times, storage size and query execution. As the results in Section 3.4 will show, SUCXENT++ outperforms SUCXENT by up to 20 times and XParent by up to 40 times for document insertion and takes up to 2.5 times less storage space.

This chapter is organized as follows. Section 3.2 and Section 3.3 present SUCXENT and SUCXENT++ respectively. Next, the differences between SUCXENT and SUCXENT++ as well as existing relational storage approaches are presented. The chapter concludes with a performance evaluation. Specifically,


```

<?xml version="1.0" encoding="UTF-8"?>
<enzyme>
  <db_entry>
    <enzyme_id>1.14.17.3</enzyme_id>
    <alternate_name_list>
      <alternate_name>
        Peptidyl alpha-amidating enzyme
      </alternate_name>
      <alternate_name>
        Peptidylglycine 2-hydroxylase
      </alternate_name>
    </alternate_name_list>
    ...<!--Other parts not shown due to lack of space-->
    <enzyme_description>
      Peptidylglycine monooxygenase
    </enzyme_description>
    <swissprot_reference_list>
      ...<!--Other parts not shown due to lack of space-->
      <reference swissprot_accession_number="P10731"
        name="AMD_BOVIN" />
      <reference swissprot_accession_number="P19021"
        name="AMD_HUMAN" />
      ...
    </swissprot_reference_list>
  </db_entry>
</enzyme>

```

Figure 3.1: Sample XML data.

document insertion/extraction times and storage space requirements are compared. Query processing will be discussed in the next chapter.

3.2 The SUCXENT Approach

In this section we describe the SUCXENT approach. First, a brief overview of the relational schema is presented. This is followed by an example to illustrate the storage of XML documents in SUCXENT. Next, an algorithm to reconstruct shredded XML documents is discussed. We also present a brief comparison with existing relational storage approaches. A brief description of the work in this section can be found in [44]. Note that query processing aspects of the schema are discussed in greater detail in the next chapter.

3.2.1 Schema Description

Consider the document in Figure 3.1 and its tree representation in Figure 3.2. These two are used as examples to illustrate the SUCXENT schema.

We first define some symbols to facilitate exposition. Let n_k be a node in the XML tree X . Then the level of n_k is denoted as ℓ_k . We denote the

CHAPTER 3. RELATIONAL SCHEMA DESIGN

Figure 3.2: Tree representation.

Document (DocId, Name)
Path (PathId, Path)
PathValue (DocId, PathId, LeafOrder, SiblingOrder, BranchOrder, LeafValue)
LargeText (DocId, PathId, LeafOrder, SiblingOrder, BranchOrder, LeafValue)
AncestorInfo (DocId, SiblingOrder, AncestorOrder, AncestorLevel)

Attribute	Summary
DocId	Unique identifier for each document stored.
PathId	Unique identifier for each distinct root-to-leaf path.
Path	Root-to-leaf path
LeafOrder	Order of the leaf node in preorder traversal.
SiblingOrder	Leaf nodes with same parent have the same SiblingOrder
BranchOrder	Maintains intersection information.
LeafValue	Text content
AncestorOrder	Preorder traversal value for ancestor node.
AncestorLevel	Depth of the ancestor node in XML tree.

Figure 3.3: SUCXENT schema.

maximum level of the X as L_{max} . Also, A_k denotes the set of ancestor nodes of n_k . The relational schema for SUCXENT++ is shown in Figure 3.3. There are five relations in the schema. The semantics of these relations are as follows.

- (i) The **Document** table. The table **Document** is used for storing the names of the documents in the database. This name could be the file name of the XML document or its URL. Whenever a new document is inserted in SUCXENT its file name or URL is stored in the *Name* attribute and a unique identifier is stored in the *DocId* attribute. This identifier is used as a reference to this document in the rest of the schema. Figure 3.4

CHAPTER 3. RELATIONAL SCHEMA DESIGN

shows the shredded version of the example document. The name of the document is "Enzyme 1.4.17.3" and its unique identifier is " 1 " .

- (ii) The *Path* table. This table records every unique root-to-leaf path encountered in the XML documents stored in the database. The path expression corresponding to the path is stored in the Path attribute. A unique identifier to reference this path is stored in the PathId attribute. The rest of the schema uses this identifier which is an integer to refer to this path. This is to avoid storing the root-to-leaf path which is a string and requires more space than storing an integer. For example, consider the node enzyme-id in Figure 3.2. Its root-to-leaf path is `/enzyme/db_entry/enzyme_id`. This is stored as a path with PathId=1 in the Path table of Figure 3.4. Other root-to-leaf paths are stored similarly.
- (iii) The *PathValue* table. This table stores the leaf nodes of the XML documents stored in the database. Each tuple stores one leaf node. The DocId attribute in a tuple refers to the document the leaf node in this tuple belongs to. This refers to the DocId unique identifier of the Document table. The shredded document in Figure 3.4 has DocId=1 and the leaf nodes corresponding to this document are stored in the PathValue table with this DocId value.

The PathId attribute refers to the root-to-leaf path corresponding to this leaf node as stored in the Path table. The Leaf Value attribute stores the textual content of the leaf node. Consider the first tuple in the PathValue table. This stores the first leaf node, enzyme-id, of the document. As shown in Figure 3.1 this leaf node has the root-to-leaf path `/enzyme/db_entry/enzyme_id` corresponding to PathId=1 in the Path table. The text value of this node is "1.14.17.3" and this is stored in the Leaf Value attribute. The other attributes of this table require further explanation and are discussed below.

- (a) The LeafOrder attribute (denoted as leaforder(n)) records the order of the leaf node n. This corresponds to the order in which leaf

CHAPTER 3. RELATIONAL SCHEMA DESIGN

nodes are encountered when the document is parsed in preorder traversal. For example, the first leaf node encountered in the document of Figure 4.28 is `enzyme_id`. The first tuple in the PathValue table of Figure 3.4 represent this node and has a LeafOrder value equal to 1. Similarly, `enzyme_description` is the next leaf node encountered and, therefore, its LeafOrder is 2. The text value of this node is "**Peptidylglycine monooxygenase**". This node is shown in the second tuple in the PathValue table. Formally,

Definition 3.1 [LeafOrder] Let $P = \{n_1, n_2, \dots, n_k\}$ be the set of leaf nodes in X such that $preorder(n_i) < preorder(n_{i+1}) \forall 0 < i < k$. Then $leaf_order(n_1) = 1$ and $leaf_order(n_j) = leaf_order(n_{j-1}) + 1 \forall 1 < j \leq k$. ■

- (b) The SiblingOrder attribute (denoted as `siblingorder(n)`). Two leaf nodes have the same SiblingOrder if they share the same parent. For example, in Figure 3.2, leaf nodes with LeafOrder values of 3 and 4 have the same SiblingOrder equal to 2 since they share the same parent node (node `alternate_name_list`). The assignment of SiblingOrder is done in the same manner as LeafOrder, i.e., the order in which nodes with the same parent (siblings) are encountered. The dotted boxes in Figure 3.2 indicate the leaf nodes which have the same SiblingOrder. Formally,

Definition 3.2 [SiblingOrder] Let n_i and n_k be two leaf nodes in the XML tree X . Let n_a be an internal node of X at level ℓ_a such that $n_a \in A_i, n_a \in A_j$. Then, $siblingorder(n_i) = siblingorder(n_j)$ if and only if $\ell_a = \ell_i - 1 = \ell_k - 1$. ■

- (c) The BranchOrder attribute. Before this attribute is explained consider the problem of reconstructing an XML document based only on the information present in the PathValue table. The PathValue provides a list of the leaf nodes together with their paths (as referenced by PathId). The tree structure shown in Figure 3.2 can be obtained by "stitching" together these leaf nodes. However,

CHAPTER 3. RELATIONAL SCHEMA DESIGN

in order to do this, the level at which a leaf node path intersects the adjacent leaf node paths must be known. This is not a problem for nodes with the same *SiblingOrder* as they have the same parent. However, for nodes with different *SiblingOrder* values this value must be recorded. Consider the nodes with *SiblingOrder*=2 in Figure 3.2. The paths of these leaf nodes will intersect with paths of the leaf nodes with *SiblingOrder*=1 at the node *db_entry* which is at level 2. By recording this information the portion of the document corresponding to these four (two with *SiblingOrder*=1 and two with *SiblingOrder*=2) leaf nodes can be reconstructed. Observe that only the intersection level with the path of either of the two adjacent leaf nodes needs to be recorded. Here, the intersection level with the nodes on the left hand side is recorded. This is because standard XML parsers follow preorder traversal and hence recording the intersection level with the nodes on the right hand side would require backtracking.

The *BranchOrder* attribute records the intersection level of the leaf node with leaf nodes that immediately precede it in *SiblingOrder*. So, the *BranchOrder* value for the nodes with *SiblingOrder*=2 in Figure 3.2 is 2. Similarly, the node *name* with value "AMD_HUMAN" has a *BranchOrder* value of 3 (intersecting at *swissprot_reference_list*). It is useful for reconstructing the XML documents from their shredded relational format as discussed in the next section. Formally,

Definition 3.3 [BranchOrder] Let n_i and n_j be two leaf nodes in the XML tree X such that $leaforder(n_i) = leaforder(n_j) - 1$. Let n_a be an internal node of X at level ℓ_a such that $n_a \in A_i$, $n_a \in A_j$ and $\nexists n_k$ s.t. $\ell_k > \ell_a$, $n_k \in A_i$ and $n_k \in A_j$ for $k \neq a$. Then (1) $branchorder(n_i) = 0$ if $leaforder(n_j) = 1$ and (2) $branchorder(n_j) = \ell_a$ if $leaforder(n_j) > 1$. ■

- (iv) The **LargeText** table. Large text data (e.g., DNA sequences) can cause problems while indexing the corresponding column. So, they are not

CHAPTER 3. RELATIONAL SCHEMA DESIGN

stored in the PathValue table. Instead, a separate table, LargeText that has the same structure as PathValue table, is maintained for large text data. In SUCXENT, all textual data larger than 255 bytes in size is stored in this table.

- (v) The **AncestorInfo** table. The table AncestorInfo stores the ancestor information for each leaf node. As we know that nodes with the same parent have the same SiblingOrder we only need to maintain this information for distinct SiblingOrder values. For example, in Figure 3.2 we do not need to store the ancestor information for both leaf nodes enzyme-id (LeafOrder = 1) and enzyme-description (LeafOrder = 2). We only need to maintain the ancestor information for SiblingOrder = 1. Thus, the Siblingorder attribute reduces the number of entries in the AncestorInfo table. The DocId attribute indicates which XML document a particular ancestor node belongs to. The AncestorOrder attribute stores a unique identifier, which is the preorder traversal value, for the ancestor node in question.

As we shall see in Chapter 4, the AncestorInfo table is used to answer queries. For example, consider the XPath expression `//db_entry [enzyme_id="1.14.17.3"]`. The leaf node enzyme-id with value "1.14.17.3" can be obtained from PathValue. However, in order to retrieve the db-entry node we need to obtain all its other descendants (e.g. enzyme-description, alternate-name-list, alternate-name, *swissprot_reference_list*, reference etc.) as well. In order to do this we need to retrieve those leaf nodes whose path starts with db_entry and this db_entry node should be the same as that for the enzyme_id with value 1.14.17.3. So, we need to retrieve those leaf nodes that have the same ancestor as the enzyme-id node at level 2. This information is provided by the AncestorInfo table. The AncestorInfo table will be elaborated on further in the next chapter.

A semantic summary of the attributes in the schema is shown in Figure 3.3.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

Document		Path	
DocId	Name	PathId	PathExp
1	Enzyme1.14.17.3	1	enzyme.db_entry.enzyme_id
2	..	2	enzyme.db_entry.enzyme_description
		3	enzyme.db_entry.alternate_name_list.alternate_name
		4	enzyme.db_entry.catalytic_activity
		5	enzyme.db_entry.cofactor_list.cofactor
		6	enzyme.db_entry.comment_list.comment
		7	enzyme.db_entry.prosite_reference.prosite_accession_number
		8	enzyme.db_entry.swissprot_reference_list.reference.@name
		9	enzyme.db_entry.swissprot_reference_list.reference.swissprot_accession_number

PathValue						AncestorInfo			
DocId	PathId	LeafOrder	SiblingOrder	BranchOrder	LeafValue	DocId	SiblingOrder	AncestorOrder	AncestorLevel
1	1	1	1	-1	1.14.17.3	1	1	1	1
1	2	2	1	2	Peptidylglycine monooxygenase	1	1	2	2
1	3	3	2	2	Peptidyl alpha-amidating enzyme	1	2	1	1
1	..	...	...	...	...	1	2	5	3
1	8	11	7	2	AMD BOVIN	1	2	2	2
1	9	12	7	2	P10731	..	..	..	..
1	...	...	...	...	...	1	7	1	1
						1	..	..	...

LargeText					
DocId	PathId	LeafOrder	SiblingOrder	BranchOrder	LeafValue
1	..	...	...	...	Large text....

Figure 3.4: SUCXENT: XML data in RDBMS.

3.2.2 Document Extraction

The algorithm for reconstruction is presented in Figure 3.5. The input to the algorithm is a list of leaf nodes arranged in ascending Leaf Order. This list could be obtained as a result of a query or could simply be the whole document. The reconstruction proceeds as follows.

- (i) Each leaf node path is first split into its constituent nodes (lines 5 to 7). This process can be optimized by already storing the constituent nodes of each root-to-leaf path of the *Path* table.
- (ii) If the document construction has not yet started (line 10) then the first node obtained by splitting the first leaf node path is made the root (lines 11 to 15).
- (iii) When the next leaf node is analyzed we only need to process the nodes starting after the *BranchOrder* of that node as the nodes till this level have already been added to the document (lines 20 to 22). The nodes starting after this level are now added to the document (lines 27 to 32).
- (iv) Document extraction is completed once all the leaf nodes have been processed.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

```

Input:  $\mathcal{L} = \{n_1, \dots, n_k\}$ , a list of leaf nodes arranged in order of LeafOrder values
Output:  $\mathcal{D}$  is the document to be returned.
1:  $c$  is an XML node.
2:  $c \leftarrow \phi$ 
3:  $\mathcal{C} \leftarrow$  list of XML nodes.
4: for all  $n_i$  in  $\mathcal{L}$  do
5:   /* /book/authors/author would give  $p = [\text{book}, \text{authors}, \text{author}]$  */
6:    $p$  is the array of nodes in a path.
7:    $p = n_i.\text{Path.GetNodes}()$ 
8:   /*  $s$  is a counter */
9:    $s \leftarrow 0$ 
10:  if  $c = \phi$  then
11:     $c \leftarrow \text{new XmlDocumentNode}(p[0])$ 
12:    /* Make  $c$  the root. This happens only once. */
13:     $\mathcal{D}.\text{AddNode}(c)$ 
14:     $\mathcal{C}.\text{Add}(c)$ 
15:     $s \leftarrow 1$ 
16:  else if
17:    then
18:     $s \leftarrow n_i.\text{BranchOrder}()$ 
19:  end if
20:  /* Keep only those nodes in  $\mathcal{C}$  that are common between  $n_{i-1}$  and  $n_i$ . */
21:   $\mathcal{C}.\text{ClearFromIndex}(s)$ 
22:   $q$  is an XML node
23:  /* Need to keep it as the starting node for processing  $n_{i+1}$  */
24:   $q \leftarrow c$ 
25:  while  $s < p.\text{Length}()$  do
26:     $m \leftarrow \text{new XmlDocumentNode}(p[s])$ 
27:     $q.\text{AppendChild}(m)$ 
28:     $\mathcal{C}.\text{Add}(m)$ 
29:     $q \leftarrow m$ 
30:     $s++$ 
31:  end while
32: end for

```

Figure 3.5: Extraction algorithm.

This algorithm can be used to construct the whole document if all leaf nodes in the document are presented as input. A fragment of the document can be generated if a partial list of consecutive leaf nodes is provided. In fact, query results are returned as a list of leaf nodes and the result XML fragment(s) is constructed using this algorithm. As an example, consider reconstructing the document fragment corresponding to the first two SiblingOrders in Figure 3.2. Figure 3.6 shows the reconstructed document fragment. Essentially, we start with the first four tuples (in terms of LeafOrder) in the PathValue table in Figure 3.4. The reconstruction proceeds as follows.

- (i) The first leaf node has the path **enzyme.db_entry.enzyme_id**. It is split up into the nodes **enzyme**, **db_entry** and **enzyme_id** (line 7). Since this is the first leaf node (line 10), all the nodes in the path are added to the document and the value of the **enzyme_id** node is set to 1.14.17.3 (lines 23 to 30).

Figure 3.6: Partial tree.

- (ii) The next leaf node has the path `enzyme.db_entry.enzyme_description`. Now, only those nodes that start after the *BranchOrder* of this leaf node need to be considered (line 17). Since, its *BranchOrder* is 2, nodes are processed starting from `enzyme_description` (lines 23 to 30). So, this node is added to the document and its value is set to **Peptidylglycine monooxygenase**.
- (iii) The other nodes are processed similarly to produce the document in Figure 3.6.

3.2.3 Comparison with Existing Approaches

We discussed some of the existing relational storage approaches in Chapter 2. We present a brief comparison with some of these approaches. A detailed performance comparison is presented later in this chapter and the next.

- (i) Comparison with schema-conscious approaches. We take the Shared-Inlining approach as the representative approach for schema-conscious techniques. The main advantage SUCXENT provides is the ability to store XML documents even in the absence of defining DTDs or XML Schema. However, our performance evaluation in the next chapter indicates a major gap in performance both in terms of storage and querying.

- (ii) Comparison with XRel. XRel resolves queries using θ -joins. SUCXENT avoids θ -joins by storing the ancestor list for each leaf node. Though this results in an increase in storage space requirements, there is a significant gain in terms of query processing as we shall see in the next chapter.
- (iii) Comparison with XParent. Figure 3.7 shows the decomposition of the sample document using the XParent schema. XParent stores the ancestor information for every node in the document. SUCXENT, on the other hand, stores the ancestor information of only the leaf nodes. This results in a significant reduction in storage space requirements. Since, query processing in SUCXENT is still done without θ -joins, query performance also improves due to the reduced storage space.

3.2.4 Limitations of SUCXENT

Traditional XML query processing has taken the evaluation of the ancestor-descendant relationship as the basic query processing operation. Candidate nodes are first identified and then the final result is constructed by identifying all occurrences of ancestor-descendant relationships. This was discussed in detail in Chapter 2.

The SUCXENT approach relies on the observation that a query can be answered by evaluating the *candidate leaf nodes* and then identifying the *intersection relationships* between these leaf nodes. As illustrated in the previous section, the *AncestorInfo* table is utilized for this purpose. The advantage of following a leaf node-based approach is that far fewer candidate nodes have to be analyzed.

The *AncestorInfo* table represents the major limitation of SUCXENT. Since, the ancestor information of each leaf node is stored, the storage size requirements are quite large (though still better than XParent). A large storage size also effects query performance. Therefore, another approach to identify *intersection relationships* between leaf nodes is required. This approach should have lower storage space requirements than SUCXENT. The next section present SUCXENT++, an approach that resolves this issue.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

Document		LabelPath		
DocId	Name	Id	Len	Path
1	Enzyme1.14.17.3	1	3	enzyme.db_entry.enzyme_id
2	..	2	3	enzyme.db_entry.enzyme_description
		3	4	enzyme.db_entry.alternate_name_list.alternate_name
		4	3	enzyme.db_entry.catalytic_activity
		5	4	enzyme.db_entry.cofactor_list.cofactor
		6	4	enzyme.db_entry.comment_list.comment
		7	4	enzyme.db_entry.prosite_reference.prosite_accession_number
		8	4	enzyme.db_entry.swissprot_reference_list.reference.@name
		9	4	enzyme.db_entry.swissprot_reference_list.reference.swissprot_accession_number
		10	2	enzyme.db_entry
		11	3	enzyme.db_entry.prosite_reference
		12	3	enzyme.db_entry.swissprot_reference_list
		13	3	enzyme.db_entry.alternate_name_list
		..	..	..

Data				
DocId	PathId	Did	Ordinal	Value
1	1	&3	1	1.14.17.3
1	2	&4	1	Peptidylglycine monooxygenase
1	3	&6	1	Peptidyl alpha-amidating enzyme
1	..	..	..	..
1	8	&10	1	AMD_BOVIN
1	9	&11	1	P10731
1	..	..	..	..

Element			
DocId	PathId	Ordinal	Did
1	10	1	&2
1	1	1	&3
1	2	1	&4
1	13	1	&5
..	..	..	..
1	12	1	&8
1	..	..	..

Ancestor			
DocId	Did	Ancestor	Level
1	&2	&1	1
1	&3	&2	2
1	&4	&2	2
1	&6	&5	3
..	..	..	..
1	&10	&8	2
1	..	..	..

Figure 3.7: XML data in XParent.

3.3 The SUCXENT++ Approach

In this section we describe the SUCXENT++ approach for storing XML data in a relational database system. As the name suggests, it improves on SUCXENT. First, the relational schema and its constituents are described. Next, we present the similarities and differences between SUCXENT and SUCXENT++. A brief description of the work in this section can be found in [43]. Note that, query processing aspects of the schema are discussed in greater detail in the next chapter.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

Document (<i>DocId</i> , <i>Name</i>)	
Path (<i>PathId</i> , <i>Path</i>)	
PathValue (<i>DocId</i> , <i>PathId</i> , <i>LeafOrder</i> , <i>BranchOrder</i> , <i>BranchOrderSum</i> , <i>LeafValue</i>)	
LargeText (<i>DocId</i> , <i>PathId</i> , <i>LeafOrder</i> , <i>BranchOrder</i> , <i>BranchOrderSum</i> , <i>LeafValue</i>)	
DocumentRValue (<i>DocId</i> , <i>Level</i> , <i>RValue</i>)	

Attribute	Summary
<i>BranchOrderSum</i> , <i>RValue</i>	Useful for evaluating intersection level between two leaf nodes.
<i>Level</i>	Depth in the XML document.

Figure 3.8: SUCXENT++ schema.

3.3.1 Schema Description

Consider the document in Figure 3.1 and its tree representation in Figure 3.2. These two are used as examples to illustrate the SUCXENT++ schema.

The relational schema for SUCXENT++ is shown in Figure 3.8. There are five relations in the schema. The semantics of the **Document** and **Path** tables are the same as that of the **Document** and **Path** tables in SUCXENT. The rest of the tables have the following characteristics.

- (i) *The **PathValue** table.* This is similar to the **PathValue** table in SUCXENT except for two differences.
 - (a) There is no *SiblingOrder* attribute. In SUCXENT the *SiblingOrder* attribute was introduced to reduce the number of entries in the **AncestorInfo** table. That is, ancestor information was grouped by siblings to reduce storage requirements. In SUCXENT++, ancestor information is not maintained. Therefore, there is no need to store *SiblingOrder*.
 - (b) The attribute *BranchOrderSum* is introduced. This attribute enables efficient query processing and is discussed in the next section.
- (ii) *The **DocumentRValue** table.* This table, together with the attribute *BranchOrderSum* in the **PathValue** table is required for efficient query processing. It is discussed in the next section.
- (iii) *The **LargeText** table.* Like SUCXENT, SUCXENT++ stores large text values in a separate table. It has the same structure as the **PathValue** table.

A semantic summary of the attributes in the schema is shown in Figure 3.8. Note that only those attributes not present in SUCXENT are summarized.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

Document		Path	
DocId	Name	PathId	PathExp
1	enzyme.xml	1	enzyme.db_entry.alternate_name_list.alternate_name
2	..	2	enzyme.db_entry.catalytic_activity
DocumentRValue		3	enzyme.db_entry.cofactor_list.cofactor
		4	enzyme.db_entry.comment_list.comment
		5	enzyme.db_entry.enzyme_description
		6	enzyme.db_entry.enzyme_id
		7	enzyme.db_entry.prosite_reference.prosite_accession_number
		8	enzyme.db_entry.swissprot_reference_list.reference.@name
		9	enzyme.db_entry.swissprot_reference_list.reference.swissprot_accession_number

DocId	Level	RValue	DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue
1	1	71	1	1	1	0	0	1.14.17.3
1	2	10	1	1	2	2	10	Peptidylglycine monooxygenase
1	3	3	1	1	3	2	17	Peptidyl alpha-amidating enzyme
1	4	2	1	1	4	3	20	Peptidylglycine 2-hydroxylase
1	5	1	1	1	5	2	27	AMD_BOVIN
1	6	1	1	1	6	4	29	P19021
1	7	1	1	1	7	2	36	AMD_HUMAN
1	8	1	1	1	8	4	38	P10731
1	9	1	1	1	9	...	...	...

Figure 3.9: SUCXENT++: XML data in RDBMS.

3.3.2 Extending the SUCXENT++ Schema

Consider the query in Figure 3.10.a. It is evident that XML fragments that satisfy the query must satisfy the following *structural* constraints: the XML document must have paths that satisfy the path expressions `/enzyme/db_entry/swissprot_reference_list/reference/@name` and `/enzyme/db_entry/swissprot_reference_list/reference/swissprot_accession_number` and these paths must *intersect* (nearest common ancestor) at a level 4. In general, given an XQuery, often we need to determine if a pair of nodes n_1 and n_2 intersect at a specific level ℓ of the XML tree. For instance, consider the XML tree in Figure 3.10.b. Suppose that the root-to-leaf paths to nodes n and m satisfy the constraints of a query. Then we need to identify the intersection level or level of the nearest common ancestor of these two nodes (level 3) efficiently in order to identify related query results.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

Figure 3.10: Query tree and query processing in sc Succent++.

The attributes discussed till now in SUCXENT++ are insufficient for identifying such intersection level efficiently. Recall that the BranchOrder attribute records the intersection level of a leaf node n with leaf node that immediately precedes it. However, the nodes n and m may not be adjacent to one another. Hence, we need to extend the SUCXENT++ schema so that we can determine the intersection level by inspecting the nodes n and m only without processing all the leaf nodes and root-to-leaf paths between n and m (Black leaf nodes in Figure 3.10.b). We achieve this by extended the schema in the following ways. First, we store an attribute RValue (denoted as R_ℓ) in the DocumentRValue table for each level ℓ of the XML tree. Second, an attribute BranchOrderSum, denoted as S_n , is assigned to a leaf node with LeafOrder n . As we shall later, these attributes allow us to determine the intersection levels of any two leaf nodes efficiently. This results in a substantial reduction in storage size and query processing time. We now elaborate on these attributes. We begin by first defining the notion of maximum k -consecutive leaf node set which shall be used to define RValue.

Definition 3.4 [Consecutive Leaf Node Set] Let $C = \{n_1, n_2, \dots, n_r\}$ be a set of leaf nodes in X such that $leaforder(n_j) = leaforder(n_{j+1}) - 1 \forall 1 \leq j < r$. Then C is called the **consecutive leaf node set**. ■

CHAPTER 3. RELATIONAL SCHEMA DESIGN

Let us illustrate the above definition with an example. Consider Figure 3.2. The nodes labeled "swissprot_accession_number", "name" and "swissprot_accession_number" consist of a consecutive leaf node set as they have leaf order values 6, 7 and 8 respectively. That is, $C_1 = \{ \text{"swissprot_accession_number"}, \text{"name"}, \text{"swissprot_accession_number"} \}$. Similarly, $C_2 = \{ \text{"alternate_name"}, \text{"name"}, \text{"swissprot_accession_number"} \}$

Definition 3.5 [k-Consecutive Leaf Node Set] Let $C = \{n_1, n_2, \dots, n_r\}$ be a consecutive leaf node set. Then C is called k -consecutive leaf node set, denoted as C_k , if the following conditions are true.

- $\text{branchorder}(n_i) \geq k \ \forall \ 1 \leq i \leq r$ and $k \in [1, L_{\max}]$.
- If $\text{leaforder}(n_1) > 1$ then $\text{branchorder}(n_\ell) < k$ and $\text{leaforder}(n_\ell) = \text{leaforder}(n_1) - 1$.
- If n_r is not the rightmost leaf node in X then $\text{branchorder}(n_{r+1}) < k$ and $\text{leaforder}(n_{r+1}) = \text{leaforder}(n_r) + 1$.

The number of nodes in C_k is denoted as $|C_k|$. ■

For instance, consider C_1 in the above example. Let $k = 3$. Then, C_1 is a k -consecutive leaf node set (also denoted as C_3) as the *BranchOrder* values of the nodes "swissprot_accession_number", "name", and "swissprot_accession_number" are 4, 3 and 4 respectively, and the *BranchOrder* value of "name" (*LeafOrder* is 5) is 2. However, C_2 is not a k -consecutive leaf node set as it has nodes having *BranchOrder* value less than 3. Observe that $|C_3| = 2$.

Definition 3.6 [Maximum k-Consecutive Leaf Node Set] Let $C_k^1, C_k^2, C_k^3 \dots C_k^m$ be the set of k -consecutive leaf node sets for a given k . Then C_k^j is **maximum k-consecutive leaf node set**, denoted as M_k , if $|C_k^j| \geq |C_k^i| \ \forall \ 0 < i \leq m$ and $i \neq j$. The number of nodes in M_k is denoted as $|M_k|$. ■

Definition 3.7 [RValue] Let $L_{\max}$ be the maximum level of an XML tree. Then **RValue** of level ℓ_k for $0 < k \leq L_{\max}$, denoted as R_k , is defined as follows.

- (i) If $\ell_k = L_{\max}$ then $R_k = 1$ and $|M_k| = 1$. ■

CHAPTER 3. RELATIONAL SCHEMA DESIGN

(ii) If $\ell_k < L_{max}$ then $R_k = R_{k+1} \times |M_b| + 1$ where $b = k + 1$. ■

Consider the document in Figure 3.2. Here $L_{max} = 5$. So, $r_5 = 1$ and $c_5 = 1$. Then, $R_4 = 1 \times 1 + 1 = 2$ and $|M_4| = 1$. This means that $R_3 = 2 \times 1 + 1 = 3$. The maximum number of *consecutive* leaf nodes with $BranchOrder \geq 3$ is 3 (maximum 3-consecutive leaf node set under the *swissprot_reference_list* element). So, $R_2 = 3 \times 3 + 1 = 10$. Similarly, $|M_2| = 7$ and $r_1 = 10 \times 7 + 1 = 71$. Observe that as the level of the XML tree decreases the value of *RValue* increases. Formally,

Property 1 Let ℓ_i and ℓ_j be two levels in the XML tree X where $0 < (i, j) \leq L_{max}$. If $\ell_i > \ell_j$ then $R_i < R_j$. ■

Next, we define the notion of *BranchOrderSum*.

Definition 3.8 [BranchOrderSum] Let n be a leaf node of X . Let $b_n = branchorder(n)$. Then the **BranchOrderSum** of n , denoted as S_n , is defined as follows:

- If $leaforder(n) = 1$ then $S_1 = 0$.
- Otherwise, $S_n = \sum_{i=1}^{b_n} R_{b_i}$

For example, the *BranchOrderSum* of the first leaf node in Figure 3.2 is 0. Since the *BranchOrder* of the second leaf node is 2 and $R_2 = 10$, *BranchOrderSum* of the second leaf node is 10. The values for the complete document are shown in the *DocumentRValue* and *PathValue* tables of Figure 3.9.

We now discuss how the *RValue* and *BranchOrderSum* enables efficient query processing. We first introduce the following theorem which is key to efficient query processing in SUCXENT++.

Theorem 3.1 Let n_i and n_j be two leaf nodes in the XML tree X . Let $i = leaforder(n_i)$ and $j = leaforder(n_j)$. If $|S_i - S_j| < R_\ell$ then the nearest common ancestor of n_i and n_j is at a level greater than ℓ .

CHAPTER 3. RELATIONAL SCHEMA DESIGN

Proof: Let n_i and n_j be two leaf nodes in the XML tree X and $\text{leaforder}(n_i) < \text{leaforder}(n_j)$. Let the nearest common ancestor of n_i and n_j be at level ℓ' in X . We would like to prove that if $|S_i - S_j| < R_\ell$ then $\ell' > \ell$.

The BranchOrderSums of n_i and n_j are as follows:

$$\begin{aligned} S_i &= R_{b_1} + R_{b_2} + \dots + R_{b_i} \\ S_j &= R_{b_1} + R_{b_2} + \dots + R_{b_j} \end{aligned}$$

Then, $|S_i - S_j| = |R_{b_{i+1}} + R_{b_{i+2}} + \dots + R_{b_{j-1}} + R_{b_j}|$ where $b_{i+1}, b_{i+2}, \dots, b_{j-1}$ are the BranchOrder values of nodes between n_i and n_j , i.e., $n_{i+1}, n_{i+2}, \dots, n_{j-1}$. Also, $\text{leaforder}(n_i) < \text{leaforder}(n_{i+1}) < \text{leaforder}(n_{i+2}) < \dots < \text{leaforder}(n_{j-1}) < \text{leaforder}(n_j)$. The maximum value of $|S_i - S_j|$ depends on the BranchOrder values of the nodes between n_i and n_j (inclusive). Based on Property 1, the value of R_{b_k} increases as b_k decreases. Hence, we need to find the minimum possible BranchOrder (level) of the nodes between n_i and n_j .

As $\text{leaforder}(n_{i+1}) > \text{leaforder}(n_i)$, the nearest common ancestor of n_i and n_{i+1} (BranchOrder) cannot be at a level less than ℓ' . Therefore, BranchOrder of n_{i+1} is $\ell_+ \geq \ell'$. Similarly, it can be shown that BranchOrder values of $n_{i+2}, n_{i+3}, \dots, n_{j-1}, n_j$ are greater than or equal to ℓ' . Therefore, minimum value of BranchOrder for a node n_r is ℓ' where $i < r \leq j$. Hence,

$$\begin{aligned} |S_i - S_j|_{\max} &= (|R_{b_{i+1}} + R_{b_{i+2}} + \dots + R_{b_{j-1}} + R_{b_j}|)_{\max} \\ &= |R_{\ell'} + R_{\ell'} + \dots + R_{\ell'} + R_{\ell'}| \\ &= |kR_{\ell'}| \text{ where } k = j - i \end{aligned}$$

Therefore, we can say $|kR_{\ell'}| < R_\ell$. Now, if $\ell' = \ell$ then this statement cannot be true. Also, if $\ell' < \ell$ then it is also not true as $R_{\ell'} > R_\ell$ (Based on Property 1). Therefore, $\ell' > \ell$. I

The attributes *RValue* and *BranchOrderSum* allow the determination of the intersection level between any two leaf nodes in a more or less *constant time*, whereas in SUCXENT, it depends on the size of the **AncestorInfo** and **Path-Value** tables as a join between these tables is required to determine the ancestor node at a particular level. The same holds true for XParent as discussed in Chapter 2. This reduces the query processing time drastically. Since this

CHAPTER 3. RELATIONAL SCHEMA DESIGN

	SD (Single-Document)	MD (Multiple-Document)
TC (Text-centric)	Online dictionaries	Digital libraries, news corpus
DC (Data-centric)	E-commerce Catalogs	Transactional data

Figure 3.11: Data set.

is achieved without storing separate ancestor information (unlike XParent or SUCXENT) the storage requirements are also reduced significantly.

The attributes *RValue* and *BranchOrderSum* allow the determination of the intersection level between any two leaf nodes efficiently. Let us elaborate on this further. Suppose a query Q is evaluated on the XML tree in Figure 3.10.b. We wish to determine if the level of the nearest common ancestor (node A) of the nodes n and m in Figure 3.10.b satisfies the structural constraints of Q . Note that the level of the intersecting nodes (denoted as ℓ_a) can be computed in SUCXENT++ in two ways. If there is no descendant axis preceding the intersecting node in the path expressions of Q then the exact value of ℓ_a can be computed from Q . Otherwise, the minimum value of ℓ_a can be computed from the path expressions in the *Path* table. Next, based on the above theorem, we compute the difference between the *BranchOrderSums* of n and m ($|S_n - S_m|$) and compare it with the *RValue* of the level $\ell_a - 1$. For instance, in Figure 3.10.b the nearest common ancestor of n and m is in level 3 and consequently $|S_n - S_m| < R_2$. Therefore, the subtree rooted at A satisfies the query constraints if ℓ_a is computed to be at least 3. Let us illustrate this with an example.

We will now illustrate the use of these attributes in query processing with an example. Consider the query shown in Figure 3.10.a and the target document in Figure 3.1. The b_n value for the first constraint satisfying *@name* node is 36. The S_n value for the second *swissprot_accession_number* node is 38 and $R_3 = 3$. Using the property proven above, we can conclude that these two nodes have common ancestors till a level l such that $l \geq 3$, since $|24 - 26| < 3$. Since, *reference* is at level 4 in both cases it is clear that they have a common *reference* node and, therefore, satisfy the query. Similarly, it can be concluded that the second *@name* node and the first *swissprot_accession_number* node have common ancestors only till a level l such that $l < 3$. (since, $R_3 = 3$ and $|36 - 29| > 3$). Therefore, these nodes do not satisfy the query.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

Data Set	No. of nodes		
	10 MB	100 MB	1 GB
DC/MD	219,382	2,183,331	23,821,115
DC/SD	238,260	2,394,886	24,810,315
TC/MD	229,258	2,335,180	23,704,294
TC/SD	279,004	2,765,209	28,419,013

Figure 3.12: Data characteristics

3.13.a: DCMD

3.13.b: DCSD

Figure 3.13: Insertion times (Data Centric).

3.3.3 Extraction

Recall the extraction algorithm for SUCXENT as shown in Figure 3.5. The attributes required for reconstruction were Path from the Path table and Leaf-Value, BranchOrder from the PathValue table. Note that these attributes are still present in SUCXENT++. Therefore, the same algorithm can be used for reconstructing XML documents or fragments of XML documents.

3.4 Performance Evaluation

This section presents the results of our performance evaluation. In this chapter, we present the results for insertion times and storage space requirements. The next chapter presents the results of query processing comparisons. We provide a comprehensive comparison by comparing SUCXENT and SUCXENT++ with one schema-conscious approach (Shared-Inlining) and one schema-oblivious approach (XParent). Results for X-Hive [3], a native XML database system are provided to compare the performance of relational approaches with existing commercial native XML database systems.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

Figure 3.14: Insertion times (Text Centric).

3.4.1 Experimental Setup

Prototypes for SUCXENT, SUCXENT++XParent and Shared-Inlining were implemented using JDK1.5. The relational database used was SQL Server 2000. All experiments were carried out on a P4 1.4GHz machine with 256 MB of RAM.

Data Set

The XBench [56] data set was used for comparison as it provides a comprehensive range of XML document types. Both text-centric and data-centric documents are provided with data sizes ranging from 10 MB to 1 GB. Figures 3.11 and 3.12 summarize the characteristics of the data sets used.

3.4.2 Insertion/Extraction of XML Documents

Figures 3.13 to 3.16 show the insertion/extraction times for the four approaches. Note that a logarithmic scale is used to display the results due to the wide disparity in the performance of the four approaches. Exact insertion times can be found in Appendix B. The following observations can be made based on the results for insertion times.

- (i) XParent performs the worst for all four document types and across all data set sizes (up to 45 times worse than SUCXENT++). This is as expected because XParent stores the greatest amount of data for a given XML document in the form of every node and its ancestor information.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

3.15.a: DCMD

3.15.b: DCSD

Figure 3.15: Extraction times (Data Centric).

3.16.a: TCMD

3.16.b: TCSD

Figure 3.16: Extraction times (Text Centric).

- (ii) SUCXENT++ performs the best for all experiments. This is because it stores the least amount of data among the schema-oblivious approaches. It performs marginally better than Shared-Inlining. This can be due to the fact that the shredding process in Shared-Inlining has to enter data into several tables whereas SUCXENT++ only needs to enter data into one table.
- (iii) The performance of XPARENT and SUCXENT is significantly worse than Shared-Inlining and SUCXENT++ - up to 20 and 45 times slower respectively. This is expected as these two approaches store the most amount of data.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

- (iv) The gap in performance increases with data size. The maximum gap in insertion performance for the 10 MB data set was 6 times for the TCSD data set between SUCXENT++ and XParent. This gap increases to 45 times for the 1 GB TCSD data set. This is because the number of non-leaf nodes (which are stored by SUCXENT and XParent in addition to the leaf nodes) increases at a faster rate than the number of leaf nodes alone.
- (v) The time taken to insert the MD (multiple document) data sets is higher for both data-centric (DC) and test-centric (TC) data sets. This is because, the insertion process has to first enumerate the list of documents and then insert them one by one. This adds an additional overhead to the insertion process.
- (vi) X-Hive performs the best among all the approaches compared. We cannot elaborate on the reasons for this as its internal storage mechanism is proprietary information and is unavailable.

The extraction time depends on the time taken to extract the relevant tuples and main-memory processing time to reconstruct the document. Exact extraction times can be found in Appendix B. Based on the results in Figures 3.15 and 3.16 the following observations can be made.

- (i) The performance of SUCXENT and SUCXENT++ is quite similar. This is because the query to extract a particular document and the reconstruction algorithm are similar in both cases.
- (ii) The extraction performance of SUCXENT is only slightly better than XParent. Even though the time taken to extract the relevant tuples (only leaf nodes) is smaller than the corresponding operation in XParent (that involves retrieving all the nodes of the document), we still have to perform *substring* operations to determine the nodes in a path in order to create the document tree. In Step 7 of Figure 3.5 the process of obtaining the node array from the path is accomplished by the *substring* operation. This means that though retrieval time from the database is better in SUCXENT and SUCXENT++ the time taken for reconstruction is more.

CHAPTER 3. RELATIONAL SCHEMA DESIGN

- (iii) Even though, Shared-Inlining returns the least number of tuples while extracting a document, its performance is still marginally worse than SUCX-ENT and SUCXENT++. This is because a join query needs to be executed to extract the document. In addition, the main memory processing time required is also higher due to the fragmented nature of the retrieved data. This difference is most significant for the TCSD data set. The TCSD data set generates the most number of relations among the four data sets for the Shared-Inlining approach.

3.4.3 Storage Size

Figures 3.17.a to 3.18.b show the storage sizes for the four approaches. The following observations can be made based on these results.

- (i) XParent has the largest storage size (up to 2.5 times more than Shared-Inlining). In fact, SUCXENT and XParent have a significantly larger storage size than SUCXENT++ and Shared-Inlining. This is expected as they store substantially more data in the form of ancestor information.
- (ii) Shared-Inlining requires the least amount of storage. In SUCXENT++ an index is created on the PathId, BranchOrderSum and Value attributes in the **PathValue** table. Effectively, the entire data set is indexed. This is not the case in Shared-Inlining where individual columns can be indexed based on the queries being executed. This results in a significantly smaller storage space requirement of up to 1.5 times.

3.5 Summary

This chapter presented two schema-oblivious approaches - SUCXENT and SUCX-ENT++. The relational schema of each approach together with an algorithm to reconstruct XML documents was also presented. In addition, SUCXENT and SUCXENT++ were compared with existing relational storage approaches in terms of document insertion time and storage size. It was found that both perform better than existing schema-oblivious approaches such as XParent. The

CHAPTER 3. RELATIONAL SCHEMA DESIGN

3.17.a: DCMD

3.17.b: DCSD

Figure 3.17: Storage size (Data Centric).

3.18.a: TCMD

3.18.b: TCSD

Figure 3.18: Storage size (Text Centric).

performance of SUCXENT++ is far better than that of SUCXENT and is comparable to shared-inlining.

The next chapter focuses on the query processing aspects. Specifically, the mechanism for translating XQuery queries to SQL for each approach will be presented. Detailed comparison - in terms of query execution - with existing approaches will also be presented.

Chapter 4

Query Processing

In order to utilize the flexibility offered by XML, XQuery queries need to be executed transparently in both SUCXENT and SUCXENT++. This means that XQuery queries need to be translated to the corresponding SQL statements automatically.

In this chapter, we present the query translation and processing aspects of SUCXENT and SUCXENT++. First, we present a brief overview of the XQuery syntax as it is required to describe the translation mechanism. Then we highlight those aspects of the translation process that are common to both SUCXENT and SUCXENT++. This is followed by a formal algorithm to translate queries in SUCXENT. The translation algorithm is illustrated with a wide range of examples. Next, we present a brief comparison of the translated SQL statements in SUCXENT with those in existing approaches such as XParent. This is to highlight the benefits offered by SUCXENT. The same organization is followed for the discussion of query processing in SUCXENT++. This chapter concludes with a detailed performance comparison of the SUCXENT and SUCXENT++ with existing relational storage approaches - Shared-Inlining and XParent.

4.1 XQuery Syntax Overview

Figure 4.1 shows a portion of the XQuery syntax. To facilitate the discussion of the syntax, we first illustrate the concept of a parse tree. Consider the query in Figure 4.2. Its *parse tree* based on the XQuery grammar is shown in Figure 4.3.

CHAPTER 4. QUERY PROCESSING

```

XPath      ::= ExprSequence?
ExprSequence ::= Expr ("," Expr)*
Expr       ::= OrExpr
OrExpr     ::= AndExpr ( "or" AndExpr )*
AndExpr    ::= ForExpr ( "and" ForExpr )*
ForExpr    ::= (SimpleForClause "return")* QuantifiedExpr
. . . . .
PathExpr   ::= ("/" RelativePathExpr?) | ("//" RelativePathExpr) | RelativePathExpr
RelativePathExpr ::= StepExpr ("/" | "//") StepExpr*
StepExpr   ::= (ForwardStep | ReverseStep | PrimaryExpr) Predicates
. . . . .
NodeTest   ::= KindTest | NameTest
NameTest   ::= QName | Wildcard

```

Figure 4.1: XQuery Syntax.

```

for $b in /sptr/entry
where $b/feature[@type = 'transmembrane region'
and $b/organism/name = 'human'
return $b/accession

```

Figure 4.2: Sample query.

A *parse tree* represents the constituent pieces of an expression. Given a grammar (or syntax) and an expression confirming to that syntax, a parser generates a parse tree whose nodes are:

- (i) Leaf nodes that represents grammar components that correspond to string or numerical literals, operators, parentheses etc. called *atoms*.
- (ii) Internal nodes that correspond to grammar components that are further defined by the *rules* of the grammar.

Figure 4.3 shows the parse tree for the query in Figure 4.2. The node FLWR represents a FLWR expression and is, therefore, an internal node. It binds a PathExpr type (discussed below) to a variable. The bound variable now provides a means for referring to the context path as specified in the *for* clause.

Leaf nodes in this parse tree correspond to either names of the nodes in the path expressions (e.g., (*sptr*, *entry*) or string literals (e.g., "transmembrane region" and "human").

In Figure 4.3, the type PathExpr defines a path expression. For example, the first PathExpr type denotes */sptr/entry* as *sptr* and *entry* are the nodes that come under this expression. The FLWR expressions above this PathExpr

CHAPTER 4. QUERY PROCESSING

Figure 4.3: Parse Tree.

type binds the path `/sptr/entry` to the variable `b`. As a result the second `PathExpr` type actually denotes the path `/sptr/entry/feature`.

The type `WhereClause` defines a combination of one or more predicates in a query. In this example, it defines an `AndExpr` as the query involves a conjunction of two predicates. Each predicate is a `ComparisonExpr` type which defines the predicate by specifying the comparison operation (`'='`, `'>'` etc.) and the comparison value. In this example, two `ComparisonExpr` types are shown. The first one specifies the condition `/sptr/entry/feature[@type='transmembrane region']` and the second specifies the condition `/sptr/entry/organism/name='human'`.

To summarize, an XQuery consists of the following components:

CHAPTER 4. QUERY PROCESSING

- (i) A **for** or **let** clause. These two bind a variable to a context path. The difference between the two is that a **for** clause creates "n" different bindings when the context path evaluates to a sequence of "n" nodes whereas the **let** clause creates one single binding. The **let** clause is useful when specifying aggregate queries.

In the given example we have only shown a single variable binding using the **for** clause. More than one variable bindings can be specified. This is especially useful when formulating join queries. Examples will be shown later to illustrate this.

- (ii) A **where** clause. This specifies the filter condition for the query. It is defined as a combination of one or more **ComparisonExpr** types. Each **ComparisonExpr** type defines a single predicate in terms of a **PathExpr** type, a comparison operator and a comparison value. The **PathExpr** type indicates the path of the XML node(s) being compared. It is usually defined in terms of the context path(s) as specified in the **for** clause. The comparison value is either a literal or - in the case of a join condition - another **PathExpr** type. In the given example, a string literal is shown. Later examples will elaborate on the formulation of join conditions.
- (iii) A **return** clause. This specifies the XML document node(s) that should be returned as results. It is defined in terms of **PathExpr** types.

4.1.1 Types of XQuery Supported

In the next section, we discuss how XQuery queries are translated to corresponding SQL queries in SUCXENT++. A full implementation of the XML query processing system would require a fully-functional XQuery support. However, building a system like this would take a significant number of person-years to implement. Instead, we implemented an interface that supports basic types of recursive XQuery queries which do not include aggregate functions and ordering of result elements. These queries are sufficient to justify the positive contributions made by our approach.

The main structure of an XQuery query can be formulated by an FLWOR expression with the help of XPath expressions. An FLWOR expression is constructed

CHAPTER 4. QUERY PROCESSING

from FOR, LET, WHERE, ORDER BY, and RETURN clauses. FOR and LET clauses serve to bind values to one or more variables using path expressions. The FOR clause is used for iteration, resulting in a single binding for each variable. As the LET clause is usually used to process grouping and aggregate functions, the processing of the LET clause is not discussed here. The optional WHERE clause specifies one or more conditions to restrict the tuples generated by FOR and LET clauses. The RETURN clause is used to specify an element structure and to construct the result elements in the specified structure. The optional ORDER BY clause determines the order of the result elements. We ignore the ORDER BY clause in this paper.

A basic recursive XQuery query can be formulated with a simplified **FWCR** expression:

```

FOR       $x_1$  in  $p_1, \dots, x_n$  in  $p_n$ 
WHERE     $c$ 
RETURN    $s$ 

```

In the **FOR** clause, iteration variables $x_1, x_2, \dots, x_n$ are defined over the path expressions $p_1, p_2, \dots, p_n$. In the **WHERE** clause, the expression c specifies conditions for qualified binding-tuples generated by the iteration variables. Some conditions may be included in p_i to select tuples iterated by the variables x_i . In the **RETURN** clause, the return structure is specified by the expression s .

4.2 Overview

As highlighted in the previous chapter, both SUCXENT and SUCXENT++ rely on the observation that a query can be answered by evaluating the candidate leaf nodes and then identifying the intersection relationships between these leaf nodes. Both approaches represent a path-oriented approach to query processing. As a result the query translation process in both approaches is quite similar.

The component of XQuery that corresponds to these paths is the PathExpr type. The PathExpr type is further composed of StepExpr types (that define '/' or '//') and NodeTest (that define the names of the nodes) types. As

CHAPTER 4. QUERY PROCESSING

Query	::= QueryExpr ((and or)QueryExpr)*
QueryExpr	::= ComparisonExpr ((and or)ComparisonExpr)*
ComparisonExpr	::= Expr (ValueComp) Expr
ValueComp	::= "=" "!=" "<" "<=" ">" ">="
Expr	::= PathExpr Literal
PathExpr	::= (PathExpr ("/" "//" "*")PathExpr) Name

Figure 4.4: Modified XQuery syntax.

Figure 4.5: Modified parse tree (Abstract Query Tree).

both SUCXENT and SUCXENT++ are based on a path-oriented approach, only PathExpr types are of use in query translation. Therefore, the XQuery parse tree is first transformed to another structure that has PathExpr as an atomic type rather than a complex type. We refer to this structure as the *Abstract Query Tree*. Figure 4.4 shows the simplified grammar used in the translation process and Figure 4.5 shows the modified parse tree - i.e., the *Abstract Query Tree* - for the query in Figure 4.2. Notice how the PathExpr type is now used as an atomic type. Also notice that only the WhereClause type in the XQuery grammar is modified. The rest of the components (e.g., return clause) remain the same and are therefore not shown in Figure 4.4.

The translation algorithms in both approaches proceed as follows.

- (i) First, the ComparisonExpr types are resolved i.e., converted to SQL statement fragments.
- (ii) Next, the where clause is replaced with combinations of these SQL fragments in the SQL where clause.
- (iii) Additional SQL conditions that enforce the *intersection relationships* are added to the SQL where clause.

CHAPTER 4. QUERY PROCESSING

```

Input:
  XQuery query  $X_q$ 
Output:
  Translated SQL query  $S_q$ 
1:  $Q_T :=$  Abstract Query Tree
2:  $Q_T = \text{ParseAndGenerateAQT}( X_q )$ 
3: /* Rule Collection refers to Query Rule Collection. Here
4: Rule and Query Rule are used interchangeably */
5:  $C_R :=$  Rule Collection
6:  $C_R = \text{TraverseAQTAndGenerateRules}( Q_T )$ 
7: String where_clause_predicate
8: for all Rule  $r_i$  in  $C_R$  do
9:   /* $r_i$ .whereClausePredicate() specified
10:   in Figure 11*/
11:   where_clause_predicate +=  $r_i$ .whereClausePredicate()
12: end for
13: String where_clause_ixn
14: for all Rule  $r_i$  in  $C_R - \{r_{last}\}$  do
15:   /* $r_i$ .whereClauseIxn( $r_{i+1}$ ) specified
16:   in Figure 12*/
17:   where_clause_ixn +=  $r_i$ .whereClauseIxn( $r_{i+1}$ )
18: end for
19: String from_clause
20: for all Rule  $r_i$  in  $C_R$  do
21:   /*Specified in Figure 11*/
22:   from_clause +=  $r_i$ .fromClause()
23: end for
24:  $S_q = \text{from\_clause} + \text{where\_clause\_predicate} + \text{where\_clause\_ixn}$ 

```

Figure 4.6: Translation algorithm.

- (iv) Finally, the return clause is resolved by adding appropriate relations to the SQL select clause.

Both algorithms are approximately the same except for step 3. The resolution of the *intersection relationships* is what separates the two approaches. We will elaborate on this by discussing the translation procedure for each approach in greater detail.

4.3 Translation in SUCXENT

Figures 4.6, 4.7 and 4.8 show the translation algorithm for SUCXENT. We illustrate this algorithm with two examples. One example shows the translation of a simple query with two predicates.

Recursive XML queries are XML queries that contain the descendant axis (*//*). Recursive XML queries are considered to be quite significant in the context

CHAPTER 4. QUERY PROCESSING

```

Input:
  Rule  $r_i$ 
Output:
   $w := r_i.\text{whereClausePredicate}()$ . The where clause portion
  due to Rule  $r_i$ .
1: String predicate_stmt
2: /*Join PathValue (alias  $P_i$ ) and Path (alias  $E_i$ ) tables*/
3: predicate_stmt = '(  $P_i.\text{PathId} = E_i.\text{PathId}$  )'
4: if  $r_i.\text{GetPath}()$  contains '%' then
5:   /*GetPath() returns the path associated with the rule*/
6:   predicate_stmt += '( $E_i.\text{PathExp}$  LIKE  $r_i.\text{GetPath}()$ )'
7: else if
8:   then
9:   predicate_stmt += '( $E_i.\text{PathExp} = r_i.\text{GetPath}()$ )'
10: end if
11: /*Add tables and alias to from clause*/
12:  $r_i.\text{fromClause}() += \text{'PathValue as } P_i, \text{ Path as } E_i\text{'}$ 
13: /*GetValue() returns the value associated with the rule*/
14: /*Some rules may not have a value e.g. return paths as
15: no predicate has been placed on them*/
16: if  $r_i.\text{GetValue}() \neq \text{null}$  then
17:   predicate_stmt += '(  $P_i.\text{PathValue} = r_i.\text{GetValue}()$  )'
18: end if
19:  $w = \text{predicate\_stmt}$ 

```

Figure 4.7: Functions to generate **where-clause-predicate** of Figure 4.6.

of XML query processing [30] as they allow greater query flexibility. Therefore, we also illustrate the translation of these types of queries. The second example illustrates a join query with recursive path expressions.

4.3.1 Simple Query Translation

Consider the query in Figure 4.2. Figure 4.9 shows its SQL translation as obtained by applying the translation algorithm shown in Figures 4.6, 4.7 and 4.8. The translation proceeds as follows.

- (i) Lines 4 and 5 in Figure 4.9 translate the part of the query that seeks an entry with **"transmembrane region"**. Note that we store only the leaf nodes, their textual content and *PathId* in the PathValue table. The actual path expression corresponding to the leaf node is stored in the Path table. Therefore, we need to join the two to obtain leaf nodes that correspond to the path **/sptr/entry/feature/@type** and have the value **"transmembrane region"**. This corresponds to the lines 3, 15 and 16 in Figure 4.7.

CHAPTER 4. QUERY PROCESSING

Input:

Rule r_i and next rule r_{i+1} in the rule Collection

Output:

$w_{ixn} := r_i.\text{whereClausePredicateIxn}(r_{i+1})$. The where clause portion due to a tree relation between r_i and r_{i+1} . For example, the rules `embl_data.entry.keywords.keyword = "Kinase"` and `embl_data.entry.db_xref = "P9030"` share the same entry element.

```

1: String ixn_stmt
2: /*Check if the rules belong to the same DTD.
   Otherwise return an empty string*/
3: if  $r_i.\text{getDTD}() == r_{i+1}.\text{getDTD}()$  then
4:   /*If yes then join LeafAncestorInfo
     (alias  $A_i$  and PathValue (alias  $P_i$ ) tables*/
5:   ixn_stmt += ' $P_i.\text{SiblingOrder} = A_i.\text{SiblingOrder}$ '
6:   ixn_stmt += ' $P_{i+1}.\text{SiblingOrder} = A_{i+1}.\text{SiblingOrder}$ '
7:   /*Ensure that the ancestor orders are the same
     (e.g. same entry element)*/
8:   ixn_stmt += ' $A_i.\text{AncestorOrder} = A_{i+1}.\text{AncestorOrder}$ '
9:   /*Find the common level for the two rules
     e.g. in the above example
     the common element entry is at level 2*/
10:  ixn_stmt += ' $A_i.\text{AncestorLevel} = \text{level}$ '
11:  ixn_stmt += ' $A_{i+1}.\text{AncestorLevel} = \text{level}$ '
12:  /*Add tables and alias to from clause*/
13:   $r_i.\text{fromClause}() += \text{'LeafAncestorInfo as } A_i \text{'}$ 
14: end if
15:  $w_{ixn} = \text{ixn\_stmt}$ 

```

Figure 4.8: Functions to generate **where-clause-ixn** of Figure 4.6.

- (ii) Lines 6 and 7 do the same for the extraction of leaf nodes that correspond to the path `/sptr/entry/organism/name` and have the value "human"
- (iii) Line 8 extracts the leaf nodes that correspond to the path `/sptr/entry/accession`.
- (iv) Lines 3 to 13 in Figure 4.8 generate the SQL fragment that ensures that the extracted leaf nodes have the correct intersection relationships. In this example, the nodes have to intersect at the entry node which is at level 2. Therefore, they share the same level 2 ancestor. Lines 9 to 21 in Figure 4.9 ensure this together with the condition that these nodes should belong to the same document.
- (v) Line 1 returns the properties of the leaf nodes corresponding to the accession element. These properties are needed to construct the corresponding XML fragment. The reconstruction algorithm discussed in the previous

CHAPTER 4. QUERY PROCESSING

```

1: select P3.LeafOrder, P3.BranchOrder, P3.LeafValue, E3.PathExp
2: from PathValue P1, Path E1, PathValue P2, Path E2, PathValue P3, Path E3
3:   AncestorInfo as A1, AncestorInfo as A2, AncestorInfo A3
4: where   E1.PathExp = '/sptr/entry/feature/@type'
5:   and P1.PathId = E1.PathId and P1.LeafValue = 'transmembrane region'
6:   and E2.PathExp = '/sptr/entry/organism/name'
7:   and P2.PathId = E2.PathId and P2.LeafValue = 'human'
8:   and E3.PathExp = '/sptr/entry/accession' and P3.PathId = E3.PathId
9:   and A1.SiblingOrder = P1.SiblingOrder
10:  and A1.DocId = P1.DocId
11:  and A1.AncestorLevel = 2
12:  and A2.SiblingOrder = P2.SiblingOrder
13:  and A2.DocId = P2.DocId
14:  and A2.AncestorLevel = 2
15:  and A3.SiblingOrder = P3.SiblingOrder
16:  and A3.DocId = P3.DocId
17:  and A3.AncestorLevel = 2
18:  and A1.DocId = A2.DocId
19:  and A1.AncestorOrder = A2.AncestorOrder
20:  and A2.DocId = A3.DocId
21:  and A2.AncestorOrder = A3.AncestorOrder

```

Figure 4.9: Translated SQL query.

```

for $b in /sptr/entry, $c in /enzyme/entry
where
  ($b/feature/@type='transmembrane region')
  and $c/@swissprot_accesion_number = $b/accession
return $b/sequence

```

Figure 4.10: Join query.

chapter is used to construct the XML document from this relational result set.

The above example illustrated the translation of a simple query. Next, we will present an example to illustrate the translation of join query. This example will also demonstrate the translation of recursive queries.

4.3.2 Join Query Translation

Consider the query in Figure 4.10. Figure 4.11 shows its SQL translation as obtained by applying the translation algorithm shown in Figures 4.6, 4.7 and 4.8. The translation is similar to the previous example except for the following.

CHAPTER 4. QUERY PROCESSING

```

1: select P3.LeafOrder, P3.BranchOrder, P3.LeafValue, E3.PathExp
2: from PathValue P1, Path E1, PathValue P2, Path E2, PathValue P3, Path E3,
3: PathValue P4, Path E4, AncestorInfo A1, AncestorInfo as A2, AncestorInfo A3
4: where   E1.PathExp = '/sptr/entry/feature/@type'
5:         and P1.PathId = E1.PathId and P1.LeafValue = 'transmembrane region'
6:         and E2.PathExp = '/sptr/entry/accession'
7:         and P2.PathId = E2.PathId
8:         and E3.PathExp LIKE '/sptr/entry/sequence%' and P3.PathId = E3.PathId
9:         and A1.SiblingOrder = P1.SiblingOrder
10:        and A1.DocId = P1.DocId
11:        and A1.AncestorLevel = 2
12:        and A2.SiblingOrder = P2.SiblingOrder
13:        and A2.DocId = P2.DocId
14:        and A2.AncestorLevel = 2
15:        and A3.SiblingOrder = P3.SiblingOrder
16:        and A3.DocId = P3.DocId
17:        and A3.AncestorLevel = 2
18:        and A1.DocId = A2.DocId
19:        and A1.AncestorOrder = A2.AncestorOrder
20:        and A2.DocId = A3.DocId
21:        and A2.AncestorOrder = A3.AncestorOrder
22:        and E4.PathExp LIKE '/enzyme/entry/%/swissprot_accession_number'
23:        and E4.PathId = P4.PathId
24:        and P4.LeafValue = P2.LeafValue

```

Figure 4.11: Join query translation.

- (i) Line 8 evaluates the sequence element. The LIKE clause is used to extract leaf nodes that begin with the path `/sptr/entry/sequence`. This returns the sequence elements and its child nodes.
- (ii) Line 22 evaluates the path expression `/enzyme/entry//swissprot_accession_number`. The LIKE clause is used to evaluate the descendant (`'/'`) axis. This is how recursive queries (with descendant axis) are handled in SUCXENT.
- (iii) Line 24 evaluates the join condition. Note that the nodes returned by P4 and P2 are not constrained to be in the same document. This is because they are bound to different variables.

The above example illustrates the translation of join queries as well as recursive path expressions. It also demonstrates the extraction of an element and its children.

4.3.3 Comparison with Existing Approaches

We will compare the translations generated by XParent and SUCXENT for the query in Figure 4.2 to highlight the advantages of SUCXENT. Figure 4.12 shows the XParent translation. The following observations can be made based on this translation.

- (i) Both SUCXENT and XParent avoid θ -joins.
- (ii) XParent uses the **LabelPath**, **Ancestor** and **Data** tables to evaluate the query.
- (iii) The size of the **PathValue** and **Path** tables in SUCXENT is the same as that of the **Data** and **LabelPath** tables in XParent.
- (iv) XParent stores the ancestor information for every node in the document. SUCXENT stores this information only for the leaf nodes. This results in a substantial reduction in the processing time for joins between **PathValue** and **AncestorInfo** tables in SUCXENT compared to **Data** and **Ancestor** tables in XParent.

We will present a detailed performance comparison in Section 4.5.

4.4 Translation in SUCXENT++,

Figure 4.14 shows the translation algorithm for SUCXENT++. We illustrate this algorithm with two examples. One example shows the translation of a simple query with two predicates. The second example illustrates a join query.

4.4.1 Query Translation

Consider the query in Figure 4.2. Figure 4.13 shows its SQL translation as obtained by applying the translation algorithm shown in Figure 4.14. The translation proceeds as follows.

CHAPTER 4. QUERY PROCESSING

```

1: select P3.*
2: from Data P1, LabelPath E1, Data P2, LabelPath E2, Data P3, LabelPath E3
3:   Ancestor A1, Ancestor A2, Ancestor A3
4: where   E1.Path = '/sptr/entry/feature/@type'
5:   and P1.PathId = E1.PathId and P1.Value = 'transmembrane region'
6:   and E2.Path = '/sptr/entry/organism/name'
7:   and P2.PathId = E2.PathId and P2.Value = 'human'
8:   and E3.Path = '/sptr/entry/accession' and P3.PathId = E3.PathId
9:   and A1.Did = P1.Did
10:  and A1.DocId = P1.DocId
11:  and A1.Level = 2
12:  and A2.Did = P2.Did
13:  and A2.DocId = P2.DocId
14:  and A2.Level = 2
15:  and A3.Did = P3.Did
16:  and A3.DocId = P3.DocId
17:  and A3.Level = 2
18:  and A1.DocId = A2.DocId
19:  and A1.Ancestor = A2.Ancestor
20:  and A2.DocId = A3.DocId
21:  and A2.Ancestor = A3.Ancestor

```

Figure 4.12: Translated SQL query in XParent.

```

01: Select v3.* from Path p1, Pathvalue v1, Path p2, PathValue
02: v2, Path p3, PathValue v3, DocumentRValue d
03: Where
04: p1.Path='/sptr.entry.feature.@type' and p1.[id]=v1.pathid
05: And v1.[value]='transmembrane region'
06: And p2.Path='/sptr.entry.organism.name' and p2.[id]=v2.PathId
07: And v2.[value]='human'
08: And p3.path='/sptr.entry.accession' and p3.[id]=v3.pathid
09: And v1.docid=d.docid and d.[level]=2
10: And v1.docid=v3.docid
11: And abs(v1.branchordersum-v3.branchordersum)<d.RValue
12: And v1.docid=v2.docid
13: And abs(v1.branchordersum-v2.branchordersum)<d.RValue

```

Figure 4.13: Translated SQL query in SUCXENT++.

- (i) Lines 4 and 5 in Figure 4.13 translate the part of the query that seeks an entry with "transmembrane region". Note that we store only the leaf nodes, their textual content and *PathId* in the PathValue table. The actual path expression corresponding to the leaf node is stored in the Path table. Therefore, we need to join the two to obtain leaf nodes that correspond to the path `/sptr/entry/feature/@type` and have the value "transmembrane region". This corresponds to the lines 7 to 9 in Figure 4.7.
- (ii) Lines 6 and 7 do the same for the extraction of leaf nodes that correspond to the path `/sptr/entry/organism/name` and have the value "human".

CHAPTER 4. QUERY PROCESSING

```

Input:
  XQuery query X
Output:
  Translated SQL query Q
1: parse( X ) /* generates the parse tree */
2: WhereClause w = Q.getWhereClause() /* this is determined by the for and where clauses */
3: SelectClause s = Q.getSelectClause() /* the return clause determines this */
4: for all ComparisonExpr ei in X do
5:   if typeof(ei.lhs) is PathExpr and typeof(ei.rhs) is Literal then
6:     replace ei with
7:       "Pathi.Path " + GetPathExprCondition( ei.lhs );
8:       + "and Pathi.Id = PathValuei.Id"
9:       + "and PathValuei.value <ValueComp> ei.Literal"
10:   else if
11:     then
12:       replace ei with /* this is a join expression */
13:       "Pathi.Path " + GetPathExprCondition( ei.lhs );
14:       + "and Pathi.Id = PathValuei.Id"
15:       + "and Pathi+1.Path " + GetPathExprCondition(ei.rhs)
16:       + "and Pathi+1.Id = PathValuei+1.Id"
17:       + "and PathValuei.value <ValueComp> PathValuei+1.value"
18:   end if
19:   w.add(ei)
20: end for
21: for all PathExpr pi in X do
22:   for all PathExpr pj ≠ pi in X do
23:     if pi and pj bind to the same variable then
24:       w.add("PathValuei.DocId=PathValuej.DocId")
25:       l = I(pi, pj)
26:       w.add("abs(PathValuei.BranchOrderSum"
27:         + "-PathValuej.BranchOrderSum) < l")
28:     end if
29:   end for
30:   if pi in X.returnClause() then
31:     s.add("PathValuei.*")
32:   end if
33: end for
34: GetPathExprCondition( pathExpr ) {
35:   if pathExpr contains '/' or pathExpr contains '*' then
36:     replace '/' with
37:     return "LIKE " + pathExpr
38:   end if
39:   replace '/' with '.' /* Sucxent++ uses '.' as the path separator */
40:   return "=" + pathExpr; /* modified in the lines above to suite Sucxent++'s storage m
41: }
42: I( path1, path2 ) {
43:   x = highest common node between path1 and path2
44:   /* this represents a recursive query where the intersection level
45:   cannot be determined beforehand - has to be done while the query is executing */
46:   if x is preceded by '/' in both path1 and path2 then
47:     return indexOf( x, path1 );
48:   end if
49:   /* even if the path involves '/' the intersection level is
50:   not determinable beforehand only if both paths have '/' before the
51:   intersection node. Else, return the followin */
52:   return level of x in the path where it is not preceded by '/'
53: }

```

Figure 4.14: Translation algorithm for SUCXENT++.

- (iii) Line 8 extracts the leaf nodes that correspond to the path **/sptr/entry/accession**.
- (iv) Lines 9 to 13 in Figure 4.8 generate the SQL fragment that ensures that the extracted leaf nodes have the correct intersection relationships. In this example, the nodes have to intersect at the entry node which is at level 2. It calculates the absolute value of the difference between the *BranchOrderSum* values and ensures that it is below the *RValue* for level 2. This corresponds to lines 24 to 26 in Figure 4.14.
- (v) Line 1 returns the properties of the leaf nodes corresponding to the *accession* element. These properties are needed to construct the corresponding


```

01: Select v2.* from Path p1, Pathvalue v1, Path p2, PathValue 02:
v2, Path p3, PathValue v3, PathValue v4, Path p4, DocumentRValue d
03: Where
04: p1.Path='.sptr.entry.feature.@type' and p1.[id]=v1.pathid
05: And v1.[value]='transmembrane region'
06: And p2.Path LIKE '.sptr.entry.sequence%' and p2.[id]=v2.PathId
07: And p3.Path='.sptr.entry.accession' and p3.[id]=v3.pathid
08: And p4.Path LIKE '/enzyme/entry/%/swissprot_accession_number'
09: And p4.[id] = v4.PathId and v4.[value]=v3.[value]
10: And v1.docid=d.docid and d.[level]=2
11: And v1.subtreeid=v3.subtreeid and v1.docid=v3.docid
12: And abs(v1.branchordersum-v3.branchordersum)<d.RValue
13: And v1.subtreeid=v2.subtreeid and v1.docid=v2.docid
14: And abs(v1.branchordersum-v2.branchordersum)<d.RValue

```

Figure 4.15: Join query translation in SUCXENT++.

XML fragment. The reconstruction algorithm discussed in the previous chapter is used to construct the XML document from this relational result set.

Notice that the translation procedure in SUCXENT++ is quite similar to SUCXENT except for the determination of the intersection relationships between leaf nodes.

Figure 4.15 shows the SQL translation for the join query in Figure 4.10. The procedure is quite similar to the previous example. Notice the use of the LIKE clause to extract non-leaf elements and evaluate recursive path expressions. The join condition is also evaluated in a manner similar to SUCXENT. Lines 11 to 16 in Figure 4.14 generate the SQL fragments that correspond to the join condition. Lines 8 and 9 show the join condition in Figure 4.15.

One limitation of the current translation procedure is that it does not allow for the translation of queries with a **let** clause. This has been earmarked for future work.

4.4.2 Comparison with SUCXENT

We have already compared the translation of SUCXENT with XParent. Therefore, we will only present a comparison of SUCXENT++ with SUCXENT to illustrate why SUCXENT++ performs better (as shown in Section 4.5). The following observations can be made based on the SQL queries generated by SUCXENT and SUCXENT++.

CHAPTER 4. QUERY PROCESSING

- (i) Both approaches are based on the determination of intersection levels between leaf nodes.
- (ii) SUCXENT uses the `AncestorInfo` table to determine the intersection levels. A total of three joins are needed to determine whether two leaf nodes intersect at a given level - two joins between `PathValue` and `AncestorInfo` tables and one between two `AncestorInfo` tables. All these joins are equi-joins. SUCXENT++ on the other hand requires two joins - one between `PathValue` and `DocumentRValue` and one between two `PathValue` tables. The join between the `PathValue` tables is a 0-join.
- (iii) The number of rows in the `DocumentRValue` table is the same as the depth of the document. This means that SUCXENT++ not only has fewer joins but evaluates these joins over much smaller relations. This results in substantially better query performance as will be highlighted in Section 4.5.

4.4.3 Scope of Query Translation

The current translation algorithm and its implementation do not conform to the complete XQuery specification. That is, not all XQuery types can be translated automatically. Currently, the query types that can be translated must adhere to the simplified syntax shown in Figure 4.4. As a result quite a few query types such as those involving the `let` clause cannot be translated by the translation algorithms.

However, we must point out that the absence of automatic translation does not imply an inability to express such queries in either SUCXENT or SUCXENT++. In fact, as we shall see in the next section quite a few queries in our performance evaluation do not conform to the grammar in Figure 4.4. We translate such queries manually. The task of making the translation process XQuery complete has been earmarked for future work.

4.5 Performance Evaluation

As in the previous chapter, we present a comparison of SUCXENT and SUCXENT++ with existing relational storage approaches. The experimental setup

CHAPTER 4. QUERY PROCESSING

#	Query	Characteristics	#	Query	Characteristics
1	for \$order in input()/order[@id="1"] return \$order/customer_id	- DCMD - Exact match	7	for \$item in input()/catalog/ item[@id="I1"] return \$item/@id	- DCSD - Exact match
2	for \$a in input()/order[@id="2"] return \$a/order_lines/order_line[1]	- DCMD - Exact match, ordered access	8	for \$item in input()/catalog::item where every \$add in \$item/authors/author/ contact_information/mailling_address satisfies \$add/name_of_country = "Canada" return \$item/@id	- DCSD - Quantification
3	for \$a in input()/order[@id="4"] return \$a//item_id	- DCMD - Exact match - One '/' axis	9	for \$a in input()/catalog/item[@id="I6"] return <Output> { \$a/authors/author[i]/ contact_information/mailling_address } </Output>	- DCSD - Document construction
4	for \$a in input()/order where \$a/total gt 11000.0 order by \$a/ship_type return <Output> { \$a/@id } { \$a/order_date } { \$a/ship_type } </Output>	- DCMD - Sort - Return multiple elements	10	for \$a in input()/catalog::item where \$a/date_of_release gt "1990-01-01" and \$a/date_of_release lt "1991-01-01" and empty(\$a/publisher/ contact_information/FAX_number) return <Output> { \$a/publisher/name } </Output>	- DCSD - Data type casting - Irregular data
5	for \$a in input()/order[@id="6"] return \$a/total	- DCMD - Exact match - Document reconstruction			
6	for \$order in input()/order, \$cust in input()/customers/customer where \$order/customer_id = \$cust/@id and \$order/@id = "7" return <Output> { \$order/@id } { \$order/ order_status } { \$cust/first_name } { \$cust/ last_name } { \$cust/phone_number } </Output>	- DCMD - Join - Multiple return elements			

Figure 4.16: X Bench queries.

is the same as in the previous chapter. In order to provide a comprehensive evaluation both schema-oblivious and schema-conscious techniques are used for comparison. XParent is used as the representative schema-oblivious approach as it has been shown to outperform other schema-oblivious approaches in terms of query processing [52]. Shared-Inlining is used as the representative schema-conscious approach.

4.5.1 Data Set and Queries

In order to compare as many aspects of query processing as possible, seven data sets were used. Four of these are from the X Bench benchmark [56]. The characteristics of these data sets have been pointed out in Chapter 3. A total of 22 queries were tested on this data set. The list of queries and their characteristics is shown in Figures 4.16 and 4.17. The translated SQL queries under SUCXENT and SUCXENT++ are listed in Appendix A.

Notice that the X Bench data set does not have many recursive queries. Recursive queries are an important aspect of XML database systems. In order to evaluate the performance of SUCXENT and SUCXENT++ for recursive queries, three more data sets and 10 queries for these data sets were used. The list

CHAPTER 4. QUERY PROCESSING

#	Query	Characteristics	#	Query	Characteristics
11	for \$size in input()/catalog/:item/ attributes/size_of_book where \$size/length*\$size/width*\$size/ height > 500000 return <Output> {\$size/../../title} </Output>	- DCSD - Data type casting	16	for \$a in input()/article/prolog/ authors/author where empty(\$a/contact/text()) return <NoContact> {\$a/name} </NoContact>	- TCMD - Irregular data
12	for \$prolog in input()/article/prolog where \$prolog/authors/author/name="Ben Yang" return \$prolog/title	- TCMD - Exact match	17	for \$a in input()/article where contains (\$a/p, "the hockey") return <Output> {\$a/prolog/title} {\$a/body/abstract} </Output>	- TCMD - Text search - One '/' axis
13	for \$a in input()/article[@id="8"]/body/ section[heading="introduction"] return <HeadingOfSection> {\$a/heading} </HeadingOfSection>	- TCMD - Two conditions	18	for \$ent in input()/dictionary/e where \$ent/hwg/hw="the" return \$ent/@id	- TCSD - Exact match - Reconstruction
14	for \$a in input()/article where some \$b in \$a/body/abstract/p satisfies (contains(\$b, "the") and contains(\$b, "hockey")) return \$a/prolog/title	- TCMD - Quantification	19	for \$ent in input()/dictionary/e where \$ent/*/hw = "and" return \$ent/ss/s/qp/*/*qt	- TCSD - Path expressions
15	for \$a in input()/article[@id="5"] return <Output> {\$a/prolog/title} {\$a/prolog/authors/author/name} {\$a/prolog/dateline/date} {\$a/body/abstract} </Output>	- TCMD - Multiple return paths	20	for \$a in input()/dictionary/e [hwg/hw="the"]/ss/s/qp/q order by \$a/qd return <Output> {\$a/a}{a/qd} </Output>	- TCSD - Sorting
			21	for \$a in input()/dictionary/e where contains (\$a, "hockey") Return \$a/hwg/hw	- TCSD - Text search
			22	for \$ent in input()/dictionary/e where \$ent/ss/s/qp/q/qd="1900" return \$ent/hwg/hw	- TCSD - Exact match - Deep path

Figure 4.17: XBench queries (contd.).

Data Set	Size MB	Node Count	SUCXENT++ Size	Inlining Size	XParent Size	SUCXENT Size
ODP	142	2,884,074	174	178	402	294
XMark	150	2,668,227	202	209	477	382
Swiss-Prot	150	6,508,774	211	208	453	371

Figure 4.18: Data set.

of data sets is shown in Figure 4.18. The list of queries together with the characteristics of each query is shown in Figure 4.19.

4.5.2 Non-recursive Query Performance - XBench

We will first compare the four approaches based on the XBench benchmark. The performance characteristics for each of the 22 queries will be discussed. Note that this benchmark does not account for recursion - either in the data or in the queries.

The results are shown in Figures 4.20 to 4.25. Note that the results for the 100MB and 1GB data sets are shown on the log scale due to the wide disparity in performance between the four approaches. Actual query execution times are

CHAPTER 4. QUERY PROCESSING

Query	Database	Query Features
Q1: FOR \$b in document("odp.xml")//topic WHERE \$b/Title = "Photography" RETURN \$b/Description	ODP ¹	- Recursive schema - One // axis in query
Q2: FOR \$b in document("odp.xml")//topic WHERE \$b/Title = "Photography" RETURN \$b/Description	ODP	- Recursive schema - Two // axis in query
Q3: FOR \$b in document("odp.xml")//topic WHERE month(\$b/lastUpdate) >= 10 RETURN \$b/Description	ODP	- Recursive schema - One // axis in query - typecast
Q4: FOR \$b in document("odp.xml")//topic WHERE month(\$b/lastUpdate) >= 10 RETURN \$b/Description	ODP	- Recursive schema - Two // axis in query - typecast
Q5: FOR \$b in document("auction.xml")/site/regions RETURN count(\$b/item)	XMark[10]	- One // axis - Aggregate function
Q6: FOR \$b in document("auction.xml")/site RETURN count(\$b/description)+count(\$b/annotation)+count(\$b//email)	XMark	- One // axis with respect to root - Aggregate function
Q7: FOR \$b in document("auction.xml")/site/regions/africa/item WHERE contains(\$b/description,"gold") RETURN \$b/name	XMark	- One // axis on Recursive portion of schema - Text search
Q8: FOR \$b in document("sprot.xml")/sptr/entry WHERE \$b/reference//authorList/person[@name="Mueller P."] RETURN \$b/accession	Swiss-Prot ²	- One // axis - Distant return and where clause
Q9: FOR \$b in document("sprot.xml")/sptr/entry WHERE \$b/reference//person[@name="Hermann R."] RETURN \$b/reference	Swiss-Prot	- One // axis - Distant return and where clause - Shallow return clause
Q10: FOR \$b in document("sprot.xml")/sptr/entry WHERE \$b/reference//etyp="journal article" RETURN \$b/accession	Swiss-Prot	- One // axis - Large result size

¹The Open Directory Project. <http://dmoz.org>.²The Swiss-Prot Database. <http://us.expasy.org>

Figure 4.19: Recursive queries and their features.

4.20.a: DCMD - 10 MB

4.20.b: DCSD - 10 MB

Figure 4.20: Query execution time (Data-Centric).

provided in Appendix B.

Q1, Q2, Q3 and Q4 - Exact match and ordered access (DCMD)

For all three data sizes Shared-Inlining performs the best by a large margin (up to 30 times). This difference increases with an increase in data set size. This can be explained based on the relational schema generated for Shared-Inlining and the SQL query corresponding to query Q1. The Shared-Inlining version involves a single predicate on a single table. All other approaches involve several joins

CHAPTER 4. QUERY PROCESSING

and predicates. As a result the performance of Shared-Inlining is much better.

SUCXENT and XParent perform better than SUCXENT++ for Q1 and Q2 (up to 2 times) with SUCXENT performing better than XParent. This is due to two reasons. First, SUCXENT++ involves θ -joins whereas SUCXENT and XParent use equi-joins. Second, the data set has several small documents instead of one large document. Recall that the translations of all three schema-oblivious approaches involve a join of the *DocId* attribute to filter nodes in the same document. This generates much smaller join sizes and the results are particularly obvious for equi-join based queries.

However, the gap decreases for larger data sets. In fact for the 1GB data set SUCXENT++ performs better than XParent for Q1 and Q3. This is because both SUCXENT and XParent have a greater number of joins on a much larger data set. This reduces the advantage of equi-joins to a great extent as the data size increases. However, SUCXENT still performs better due to its more optimal storage strategy (as compared to XParent).

Q5 - Sort and Return multiple elements (DCMD)

Again, Shared-Inlining performs much better than the other three approaches (up to 100 times). The main observation here is that SUCXENT++ performs better than both SUCXENT and XParent (up to 2 times).

Recall that the number of joins in the translated SQL statement increases with the number of predicates or return clause elements in the XQuery query. Also, both SUCXENT and XParent require a greater number of joins (and on larger data) than SUCXENT++. Q4 has quite a few return elements (in addition to a sort operation). As a result, SUCXENT++ performs much better for this query.

Q6 - Join and multiple return elements (DCMD)

As expected Shared-Inlining performs the best (up to 400 times better than XParent) and SUCXENT++ performs better than both SUCXENT and XParent (up to 143 times better than XParent).

One interesting observation is that the gap between SUCXENT++ and Shared-Inlining is less for this query than for the previous queries as the data set size

CHAPTER 4. QUERY PROCESSING

4.21.a: TCMD - 10 MB

4.21.b: TCSD - 10 MB

Figure 4.21: Query execution time (Text-Centric).

4.22.a: DCMD - 100 MB

4.22.b: DCSD - 100 MB

Figure 4.22: Query execution time (Data-Centric).

increases to 1GB. This can be attributed to the fact that even Shared-Inlining has to execute more joins for this query as it involves an XQuery join.

Q7 - Exact match (DCSD)

As expected Shared-Inlining performs significantly better than the other three approaches. The main observation here is that, unlike the DCMD version of this query, SUCXENT++ performs better than SUCXENT and XParent for the 100 MB and 1GB data sets. This is because the join on the *DocId* attribute no longer generates small join sizes and the advantages of equi-joins are negated by the large data size in SUCXENT and XParent.

CHAPTER 4. QUERY PROCESSING

Figure 4.23: Query execution time (Text-Centric).

Q8 - Quantification (DCSD)

This query is quite complex as it involves quantification in the form of the every clause. The Shared-Inlining approach performs the best even though it has to execute quite a few joins as well. This is because the execution of the quantification clause is much better in the Shared-Inlining approach.

SUCXENT++ performs significantly better than the other two schema-oblivious approaches. In fact, SUCXENT and XParent take more than 60 minutes for the 1GB data set (and are, therefore, not included in the results). As before, this can be attributed to the greater number of joins on a larger data size.

Q9 - Document Construction (DCSD)

Shared-Inlining outperforms the other approaches again. With its inherent greater data fragmentation one would expect Shared-Inlining to perform the worst. However, the query involves a predicate and the construction is only for a small part of the document - with the entire part available in a single table. As a result shared-inlining performs better. However, the document reconstruction from relational tuples is less efficient in Shared-Inlining. As a result, the overall performance difference is not as significant as for previous queries.

Q10 - Irregular Data (DCSD)

Actually, the data for this query is not all that "irregular". For Shared-Inlining, all the data required for this query could be found in just one table. There-

CHAPTER 4. QUERY PROCESSING

4.24.a: DCMD - 1000 MB

4.24.b: DCSD - 1000 MB

Figure 4.24: Query execution time (Data-Centric).

fore, shared-inlining performs the best. Also, the empty clause is quite easily implemented by using `= null` in the corresponding SQL statement.

The query involves three path expressions with the empty clause. The translation for schema-oblivious approaches is quite complicated as there is no notion of "null" in these three approaches. All three implement this query using the SQL `not in` clause. As a result the performance is much worse than that of Shared-Inlining. SUCXENT++ as expected - outperforms the other two. In fact, SUCXENT and XParent fail to complete execution for the 100 MB and 1 GB data sets.

Q11 - Datatype Casting (DCSD)

Shared-Inlining can implement datatype casting in a much cleaner fashion than schema-oblivious approaches. Schema-oblivious approaches (in this case at least) store all data as strings in a single column. Shared-Inlining uses several different columns and each can be assigned a specific datatype. As a result, the query performance in Shared-Inlining is much better. In the schema-oblivious approach, data has to be first filtered based on the path expression and only then can it be typecast.

This query has five path expressions. As a result the number of joins in SUCXENT and XParent is significantly more than SUCXENT++. Therefore, SUCXENT++ performs much better.

CHAPTER 4. QUERY PROCESSING

Q12 - Exact match (TCMD)

The performance characteristics for this query are quite similar to those of Q1 except that the difference between Shared-Inlining and the schema-oblivious approaches is not as significant as in the data-centric case (Q1 to Q3). This is because the query in the Shared-Inlining approach requires two tables (compared to one in Q1).

Both SUCXENT and XParent perform better than SUCXENT++ for all three data sizes. The reasons are similar to those for Q1 to Q3 for the DCMD data set.

Q13 to Q15 (TCMD)

As expected, Shared-Inlining performs better than the other approaches. SUCXENT++ outperforms SUCXENT and XParent. The reasons are similar to those discussed earlier.

Q16 - Irregular Data (TCMD)

Unlike Q10, this query has only two path expressions. As a result, even though the schema-oblivious approaches perform worse than Shared-Inlining, the difference is not as significant as it was for Q10. Also, as expected, SUCXENT++ performed significantly better than SUCXENT and XParent.

Q17 - Text search (TCMD)

This is the first query in the data set for which Shared-Inlining performs worse than the schema-oblivious approaches. This is due to the recursive nature of the query. In Shared-Inlining, the resolution of the path expression `/article//p` requires the use of the UNION operator on three sub-queries. In addition, one of the sub-queries requires a join across seven tables.

SUCXENT++ performs better than both SUCXENT and XParent. This can be explained based on the fact that there are three path expressions in the query resulting in significantly greater number of joins (over larger data) in both SUCXENT and XParent.

One other observation is that the difference between Shared-Inlining and the schema-oblivious approaches reduces as the data size increases. This is

CHAPTER 4. QUERY PROCESSING

4.25.a: TCMD - 1000 MB

4.25.b: TCSD - 1000 MB

Figure 4.25: Query execution time (Text-Centric).

unlike the cases where Shared-Inlining performs better and the difference only increases with data set size.

Q18 - Exact match/Reconstruction (TCSD)

SUCXENT++ again performs the best. Unlike in Q9, Shared-Inlining has to join four tables to reconstruct the element e.

Q19, Q20 - Path expressions (TCSD)

SUCXENT++ performs better than all the other approaches. SUCXENT and XParent perform the worse due to the multiple path expressions in the query. XParent takes more than 60 minutes for the 1GB data set (and is, therefore, not included in the results).

Shared-Inlining performs slightly worse due to multiple joins in the resulting SQL query - although the performance is still comparable to SUCXENT++. Shared-Inlining performs worse for this query as it involves both joins and the UNION operators due to the wild card path expressions `/dictionary/e/*/hw` and `/dictionary/e/ss/s/qp/*/qt`.

Q21 - Text search (TCSD)

SUCXENT++ performs the best for this query. The query for the Shared-Inlining approach has several joins in small sub-queries that are finally combined with a UNION clause. This is due to the contains clause in the XQuery

CHAPTER 4. QUERY PROCESSING

query. As a result, the performance of Shared-Inlining is worse than that of SUCXENT++. In addition, Shared-Inlining must execute multiple joins to re-construct the *hw* element.

However, unlike Q19, SUCXENT and XParent do not perform better than Shared-Inlining. This is because the data set is a single document and the join on the *DocId* attribute no longer produces small joins.

Q22 - Exact match (TCSD)

This query presents a deep path expression resulting in several joins for the Shared-Inlining approach. As a result SUCXENT++ performs better.

Summary

Figure 4.26 presents a summary of the query performance results for the compared approaches with respect to SUCXENT++. This figure shows the ratio of time taken for a given approach to the time taken in SUCXENT++. There are two main observations. The first is that Shared-Inlining performs much better for most queries and even for the queries where its performance is worse than SUCXENT++ the gap decreases with increasing data size.

The second observation is that SUCXENT++ performs significantly better than both SUCXENT and XParent with the gap in performance increasing with data size. This is true for all the tested queries.

The above observations indicate that Shared-Inlining is the most scalable approach in addition to performing better at most queries. Among the schema-oblivious approaches SUCXENT++ delivers the best performance both in terms of query execution time and scalability. In fact SUCXENT++ outperforms Shared-Inlining for 6 of the 22 queries.

4.5.3 Query Performance - Recursive queries

In the previous set of experiments Shared-Inlining significantly outperformed other approaches for most queries. However, it did perform worse for queries with recursion and wild card path expressions. In addition, query performance is partially influenced by the flexible manner in which the XQuery return clause can be specified. In particular, two factors effect performance.

CHAPTER 4. QUERY PROCESSING

Query	Shared-Inlining			Sucxent			XParent		
Query	10MB	100MB	1GB	10MB	100MB	1GB	10MB	100MB	1GB
Q1	0.53	0.02	0.02	0.54	0.27	0.73	0.58	0.31	1.05
Q2	0.64	0.02	0.04	1.03	0.72	0.63	1.03	0.71	0.63
Q3	0.48	0.06	0.04	0.91	0.75	0.64	1.10	1.10	1.33
Q4	0.09	0.04	0.03	3.48	10.46	10.93	4.75	19.50	37.1
Q5	0.13	0.01	0.01	1.71	0.90	1.46	1.87	1.09	2.12
Q6	0.13	0.14	0.19	2.17	3.71	27.76	3.26	8.43	143.1
Q7	0.21	0.26	0.60	0.50	0.84	3.48	0.53	0.96	4.58
Q8	0.33	0.18	0.05	2.13	33.5	NA	3.91	112.2	NA
Q9	0.47	0.58	0.67	4.10	3.82	4.33	5.12	5.92	10.42
Q10	0.01	0.02	0.01	3.35	NA	NA	4.78	NA	NA
Q11	0.11	0.03	0.01	6.80	15.35	26.57	10.5	36.96	39.1

4.26.a: Summary

Query	Shared-Inlining			Sucxent			XParent		
Query	10MB	100MB	1GB	10MB	100MB	1GB	10MB	100MB	1GB
Q12	0.56	0.36	0.27	0.88	0.82	0.81	0.88	0.98	0.96
Q13	1.04	0.61	0.41	1.41	1.30	1.50	1.91	2.39	5.07
Q14	0.67	0.11	0.09	1.42	1.31	1.31	1.75	1.97	2.98
Q15	0.62	0.11	0.02	1.25	1.64	2.71	1.84	3.56	12.87
Q16	0.06	0.19	0.57	1.66	1.71	1.85	2.17	2.92	5.39
Q17	20.9	5.41	6.83	2.10	1.64	2.48	3.43	2.50	6.91
Q18	1.58	0.71	1.14	4.48	5.47	33.46	5.87	9.37	NA
Q19	5.98	1.74	3.64	3.61	4.65	18.82	5.09	9.22	NA
Q20	0.68	1.21	7.22	2.07	2.69	4.62	2.02	2.26	3.29
Q21	6.45	4.85	3.13	2.44	1.86	2.05	2.36	2.56	3.2
Q22	24.8	31.70	31.2	2.29	3.48	3.87	2.39	4.46	6.74

4.26.b: Summary (contd.)

Figure 4.26: Query performance summary.

- (i) The *distance* between the **where** and **return** clause elements, defined as the number of edges with cardinality of 1 or more (*) between these elements in the DTD influences query execution time. For example, the *distance* between *price* and *name* under *item* in the DTD in Figure 4.27 is 0 as there is no edge between them with cardinality of 1 or more. Similarly, the distance between *europe* and *price* is 1. The distance corresponds to the number of joins in the SQL query as generated by the Shared-Inlining approach. As another example, consider the query in Figure 4.29 on the document in Figure 4.28. The distance between the **return** and **where**

CHAPTER 4. QUERY PROCESSING

Figure 4.27: Sample DTD.

Figure 4.28: Sample XML document.

elements is greater than 1. The exact distance is not known as the schema is recursive and there could be any number of recursive text elements. For the shared-inlining approach this distance is the number of tables that need to be joined, thus effecting performance.

- (ii) The other factor is the depth of the element specified in the return clause. Shallow elements would require a greater number of joins in the shared-inlining approach as the descendants are likely to be fragmented across several tables.

Figures 4.31 and 4.32 show the results for the three data sets and the queries shown in Figure 4.19. Figure 4.31 shows the results for query performance. Figure 4.32 shows the variation of query execution time with increasing distance between the where and return clauses in the XQuery query. Figure 4.33 shows the results as the depth of the return clause element is reduced (or, as (D –

CHAPTER 4. QUERY PROCESSING

XQuery 1	
For	\$b in document("auction")/site/regions/africa/item
Where	contains(\$b//text, "Gold Ignot")
Return	<price>\$b/price</price>

SQL 1	
1	SELECT
2	v2.* from PathValue v1, Path p1, PathValue v2, Path p2,
3	DocumentRValue r
4	Where
5	p1.PathExp LIKE '/site/regions/africa/item/%/text'
6	AND p2.PathExp = '/site/regions/africa/item/price'
7	AND v1.PathId = p1.PathId and v2.PathId = p2.PathId
8	AND v1.LeafValue LIKE '%Gold Ignot%'
9	AND v1.DocId = v2.DocId and r1.Level=4
10	AND abs(v1.BranchOrderSum - v2.BranchOrderSum) < r1.RValue

Figure 4.29: Sample recursive query.

Figure 4.30: Query Tree.

Figure 4.31: Query performance.

depth) is increased, where D is the maximum depth of the document). Notice that query performance results are shown on the log scale. Actual timings can be found in Appendix B. The return clause results are shown as a ratio with respect to the baseline result.

Q1 to Q4 - ODP Data set

Shared-Inlining performs better than SUCXENT++ for queries Q1 and Q3. This is because the corresponding SQL query only involves the topic table and there is no need for any recursive SQL query as it is known that all topic elements are in the topic table and only their immediate title values need to be queried. SUCXENT++ on the other hand, has to execute a join query between the Path and PathValue tables. In addition Q3 involves a typecast to date in SUCXENT++'s case as all data is stored as strings.

CHAPTER 4. QUERY PROCESSING

Figure 4.32: Variation with distance.

Figure 4.33: Variation with $D - depth$ ($D = maximum\ depth$).

Q_2 and Q_4 are quite similar to Q_1 and Q_3 except that the title element can be a descendant and not just a child. A recursive SQL query is generated for the Shared-Inlining approach using the technique mentioned in [30]. In SUCXENT++ we only need to ensure that the intersection level is equal to the Length of the path minus one as the Description element is a child of the topic element in question. SUCXENT and XParent also benefit from this approach and therefore performs better than Shared-Inlining.

CHAPTER 4. QUERY PROCESSING

Q5 to Q7 - XMark Data Set

For Q_5 , Shared-Inlining involves a UNION of the joins between **item** and each of **asia**, **namerica**, **samerica**, **europe**, **africa** and **australia**. In SUCXENT this query merely look for paths with the expression **/site/regions/*/item**. However, the Shared-Inlining approach will perform much better if we use the knowledge that **item** is not used anywhere else in the document. Then, it would reduce to a count of the **item** table. This is highlighted by Q_6 where the Shared-Inlining approach performs much better than SUCXENT++ (35times). Here, the result is merely a sum of the tuples in **description** and **annotation**. This can be done because the paths are evaluated with respect to the root and it is implied that all **description** and **annotation** elements will be counted.

SUCXENT++ performs better than Shared-inlining for Q_7 . This is because the result that needs to be returned is in a different subtree. This leads to a greater number of joins in Shared-Inlining whereas, the number of joins remains unaltered for SUCXENT++.

Q8 to Q10 - Swiss-Prot Data Set

SUCXENT++ performs better for Q_9 as the result is in a different subtree. The difference is greater for Q_9 (about 5 times) than the other queries as it involves recursion, significant distance between the **return** and **where** clause elements and a shallow return clause in the form of the **reference** element (whose descendants are spread across 4 tables). However, SUCXENT++ performs worse for Q_{10} . This is because of the poor query plan generated by the database and can be resolved by applying the optimizations discussed in the next chapter.

Summary

Figure 4.34 presents a summary of the query performance results for the compared approaches with respect to SUCXENT++. This figure shows the ratio of time taken for a given approach to the time taken in SUCXENT++. SUCXENT++ outperforms Shared-Inlining for 6 out of 10 queries by up to 5 times. Shared-Inlining outperforms SUCXENT++ for one particular query by 35 times.

CHAPTER 4. QUERY PROCESSING

Query	Database	Shared-Inlining	Suxcent	XParent
Q1	ODP	0.54	0.94	1.03
Q2	ODP	1.21	1.01	1.11
Q3	ODP	0.27	1.75	1.95
Q4	ODP	1.82	1.21	1.32
Q5	XMark	1.01	1.07	1.19
Q6	XMark	0.028	1.14	1.27
Q7	XMark	1.30	1.87	2.07
Q8	Swiss-Prot	1.05	2.6	2.91
Q9	Swiss-Prot	5.48	8.29	9.31
Q10	Swiss-Prot	0.31	2.08	2.35

Figure 4.34: Summary.

4.5.4 Query Performance Summary

In this section, we have seen that Shared-Inlining performs better than schema-oblivious approaches for most types of queries. However, we also demonstrate that execution of certain types of recursive queries is more efficient using SUCX-ENT++ instead of schema-conscious approaches. SUCXENT++ performs better for recursive queries that have 1) Recursion in the schema, 2) A large distance between the elements of the where and return clauses and, 3) Shallow return clause elements. Recursive schema result in recursive SQL queries with the number of joins depending on the recursion level. For the latter, the number of joins increases with distance and shallowness.

4.6 Summary

In this chapter we have presented algorithms to translate XQuery queries to SQL based on the SUCXENT and SUCXENT++ approaches. The similarities and differences between these two approaches in terms of query translation were also highlighted. We have also demonstrated with a performance evaluation that SUCXENT++ performs better than other schema-oblivious approaches. Our experiments have shown that Shared-Inlining performs better than SUCXENT++ for most queries. However, SUCXENT++ has an advantage when the queries are recursive and on text-oriented irregular data.

CHAPTER 4. QUERY PROCESSING

In the next chapter we will demonstrate certain optimization techniques that will improve the performance of SUCXENT₊₊.

Chapter 5

Query Optimization

The comparison between the performance of SUCXENT++ and Shared-Inlining indicates quite a gap in their performances. In order to understand the reasons behind the poor performance of SUCXENT++ for some of the queries we analyzed the query plans generated by the relational query optimizer. This analysis indicated that query evaluation in SUCXENT++ can be optimized to a great extent.

In this chapter we present three orthogonal query optimization techniques. The first technique is based on altering the query plans generated by the relational query engine. The second technique utilizes query workload-based partitioning and the third uses GUI-based prefetching. The first and the third techniques essentially apply to all kinds of queries and XML data. The second technique is applicable only when a pre-defined query workload is available.

This chapter is organized as follows. Section 5.1 presents details of the optimization based on query plan alteration. Next, we present two data partitioning techniques. This is followed by our work on GUI-driven prefetching. Finally, we present a performance evaluation of the optimization techniques.

5.1 Query Plan Alteration

We first motivate the need for query plan alteration by highlighting the inadequacies of the relational query optimizer. Next, optimization approaches based on the characteristics of the query are presented.

5.1.1 Issues with the Relational Query Optimizer

A preliminary study of the query plans generated by the relational query optimizer yielded some interesting results. We noticed that the join between the **Path** and **PathValue** tables took a significant portion of the query processing time. This was because for most of the queries this join was being performed last. For example, in the SQL query of Figure 5.1.a the joins in lines 8 to 10 were evaluated first and only then was the join between **Path** and **PathValue** tables performed. The initial query plan is shown in Figure 5.1.b. Here, NL stands for Nested-Loop join, MJ for Merge Join and HM for Hash-Match join. Execution with joins between the **Path** and **PathValue** tables not only has a large number of joins, but also an inappropriate join order. The join between the tables **v1** and **p1** and **v2** and **p2** is performed at the end. As a result, **PathValue** is not filtered based on the paths `/sptr/entry/feature/@type` and `/sptr/entry/organism/name` before joining with **v3** leading to large intermediate joins. In order to improve the generated query plans we propose two optimizations that are based on the nature of the path expressions in the query.

5.1.2 Optimizing Simple Path Expressions

The join expression `v1.PathId = p1.Id` and `p1.PathExp = path` is replaced with `v1.PathId = n` where `n` is the `PathId` value corresponding to `path` in the table **Path**. Similarly, `v1.PathId = p1.Id` and `p1.PathExp LIKE path%` is replaced with `v1.PathId >= n` and `v1.PathId <= m`. For the second case `PathIds` are assigned in lexicographic order and `(n, m)` correspond to the first and last occurrences of expressions that have the prefix `path`. As an example consider the query in Figure 5.1.a and the modified version of this query in Figure 5.2.a. Notice that the join in line 4 of Figure 5.1.a is replaced by a non-join operation in line 4 of Figure 5.2.a. The value of `n` is obtained from the **Path** table in Figure 3.9. The joins in lines 6 and 8 are replaced similarly.

The above modification changes the query plan to the one in Figure 5.2.b. Since there is no join between the **PathValue** and **Path** tables anymore, the joins in Lines 9 and 10 now get executed the last. The `PathId` and `LeafValue` predicates are evaluated earlier resulting in smaller inputs to the join operations.

CHAPTER 5. QUERY OPTIMIZATION

5.1.a: SQL query with path joins.

5.1.b: Query plan with path joins.

Figure 5.1: Query and query plan with path joins.

5.2.a: SQL query without path joins.

5.2.b: Query plan without path joins.

Figure 5.2: Query and query plan without path joins.

This optimization resulted in an improvement of up to 60% in query execution time as shown in Section 5.2.6.

5.1.3 Optimizing Recursive Path Expressions

A lexicographic numbering of paths is not sufficient for recursive expressions when the DTD structure is a graph. Figure 5.3 shows an example of such a DTD. It has a graph structure due to the recursion on the section element. If only lexicographic PathId is available, expressions such as `//title` cannot be optimized i.e., converted to a range expression instead of a join. We assign another pathId, called CPathId, to a Path based on the following rules.

- (i) Elements in the DTD graph are ordered by the number of incoming edges. Lexicographic ordering is followed within this ordering. Figure 5.3 shows the "reordered" graph. The element title is ordered first as it has the

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.3: DTD graph and path numbering.

Figure 5.4: Sample XML data.

highest number of incoming edges. 1.. ..n are the CPathId values for paths ending in title.

- (ii) Cycles in the DTD graph are handled by clustering paths with the same non-recursive element after the end of the cycle. Based on this rule, `/book/section/title`, `/book/section/section/title`, ..., `/book/section/..section/title` would all occur consecutively for the DTD in Figure 5.3. This allows the replacement of paths such `//section//title` with range expressions in the SQL translation.

CHAPTER 5. QUERY OPTIMIZATION

Document (<i>DocId</i> , <i>Name</i>)
Path (<i>PathId</i> , <i>Path</i> , <i>CPathId</i> , <i>Length</i>)
PathValue (<i>DocId</i> , <i>PathId</i> , <i>CPathId</i> , <i>LeafOrder</i> , <i>BranchOrder</i> , <i>BranchOrderSum</i> , <i>LeafValue</i>)
LargeText (<i>DocId</i> , <i>PathId</i> , <i>LeafOrder</i> , <i>BranchOrder</i> , <i>BranchOrderSum</i> , <i>LeafValue</i>)
DocumentRValue (<i>DocId</i> , <i>Level</i> , <i>RValue</i>)

Figure 5.5: Modified schema.

Document		Path			
DocId	Name	PathId	PathExp	CPathId	Length
1	Auction.xml	1	site.regions.africa.item.description.keyword	1	6
2	..	2	site.regions.africa.item.description.text	3	6
		3	site.regions.africa.item.name	5	5
		4	site.regions.africa.item.price	7	5
		5	site.regions.europe.item.description.keyword	2	6
		6	site.regions.europe.item.description.text	4	6
		7	site.regions.europe.item.name	6	5
		8	site.regions.europe.item.price	8	5

PathValue						
DocId	PathId	CPathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue
1	7	6	1	0	0	Gold Ignor
1	8	8	2	4	3	\$100
1	6	4	3	4	6	gold
1	5	2	4	5	8	kwd1
1	...	...	...	...	...	...
1	3	5	13	3	85	Item3
1	4	7	14	4	88	\$30
1	2	3	15	4	91	desc4
1	1	1	16	5	93	kwd4

DocumentRValue		
DocId	Level	RValue
1	1	329
1	2	41
1	3	10
1	4	3
1	5	2
1	6	1

Figure 5.6: Relational decomposition for modified schema.

The SUCXENT++ schema has to be extended to incorporate the CPathId attribute together with the existing PathId column in the PathValue table. The modified relational decomposition based on the modified schema in Figure 5.5 is shown in Figure 5.6. Note that this corresponds to the data shown in Figure 5.4. We have not taken the running example of the previous chapter as the DTD does not contain any cycles. Any recursive path expression can now be converted to a range query on the CPathId attribute. Consider the following examples based on the DTD in Figure 5.3.

- (i) A query containing the path expression `//title` will have the join $(v.\text{PathId} = p.\text{Id} \text{ and } p.\text{Path LIKE ' \%title\%' })$ where v and p correspond to the PathValue and Path tables respectively. This join is replaced by $(p.\text{CPathId}$

≥ 1 and $p.CPathId \leq n$) as all paths ending in title have *CPathId* values between 1 and n .

- (ii) Consider the path `//section/title`. To begin with, the first and last *CPathId* values of `%section/title` in the *Path* table are obtained. Say, these are n_f and n_l , respectively. Then, the *Path*, *PathValue* join expression is replaced by $(p.CPathId \geq n_f \text{ and } p.CPathId \leq n_l)$. Note that the *PathId* column is still needed to optimize the cases mentioned in the previous section.

We must point out here that the current numbering scheme for *CPathId* does not allow for any room when new XML data with new paths is added. New *CPathId* values will have to be assigned to each path all over again. This can be easily remedied by leaving gaps in the numbering of *CPathId* (and *PathId*). This is an approach similar to the one used in numbering nodes in [33].

Another observation is that the above optimization techniques can be applied to other schema-oblivious approaches such as *XParent*, *XRel* and *SUCXENT* as they too have similar joins to process. The applicability to schema-conscious approaches needs to be investigated further as their storage approach is significantly different and the above optimizations (such as optimizing the join between *Path* and *PathValue*) tables may not apply.

5.1.4 Performance Study

Figure 5.7 to Figure 5.12 compare the performances of the 22 queries on the *XBench* data set which were also evaluated in the previous chapter. A comparison with *XParent* and *SUCXENT* is not provided as they have already been compared in the previous chapter. Notice that the performance improvement is quite significant in most cases. In some cases the improvement is significant enough to outperform *Shared-Inlining*. Figures 5.13 and 5.14 show a summary of the performance comparison. These figures show the ratio of time taken for a given approach to the time taken in *SUCXENT++* after optimization. There are four main observations.

CHAPTER 5. QUERY OPTIMIZATION

5.7.a: DCMD - 10 MB

5.7.b: DCSD - 10 MB

Figure 5.7: Query execution time (Data-Centric).

5.8.a: TCMD - 10 MB

5.8.b: TCSD - 10 MB

Figure 5.8: Query execution time (Text-Centric).

- (i) The performance of every query improves. This is expected as the number of joins in the translated SQL query is reduced. This improvement can be up to 45 times (Q11 for 1 GB data set).
- (ii) The improvement in performance increases with the data set size. This is mainly because the relational query optimizer performs worse when larger data sets are involved.
- (iii) The improvement is greater for queries with multiple path expressions or path expressions that resolve to multiple root-to-leaf paths. Consider the 1 GB data set in Figures 5.11 and 5.12. The greatest improvement is observed for queries Q2, Q10, Q11, Q18 and Q20. Q2 and Q18 have path expressions that resolve to multiple root-to-leaf paths. Q10, Q11 and Q20 have several path expressions. This result is expected as the reduction in path joins is most significant for these types of queries.

CHAPTER 5. QUERY OPTIMIZATION

5.9.a: DCMD - 100 MB

5.9.b: DCSD - 100 MB

Figure 5.9: Query execution time (Data-Centric).

5.10.a: TCMD - 100 MB

5.10.b: TCSD - 100 MB

Figure 5.10: Query execution time (Text-Centric).

- (iv) The gap in performance between Shared-Inlining and SUCXENT++ is dramatically reduced. In fact for queries Q2, Q3, and Q14 Suxent++ now performs better. It performed worse for these queries when no optimizations were done as shown in Section 4.5.3. SUCXENT++ now performs better for 12 of the 22 queries. However, Shared-Inlining still has better scalability as demonstrated by the fact this gap reduces as the data set size increases.

Figure 5.15 compares the 10 queries for the remaining three data sets presented in the previous chapter to study recursive queries. Again,

7 times (for In the case of the improvement in performance is significant enough to outperform Shared-Inlining.

CHAPTER 5. QUERY OPTIMIZATION

5.11.a: DCMD - 1000 MB

5.11.b: DCSD - 1000 MB

Figure 5.11: Query execution time (Data-Centric).

5.12.a: TCMD - 1000 MB

5.12.b: TCSD - 1000 MB

Figure 5.12: Query execution time (Text-Centric).

5.1.5 Summary

In this section, we have presented optimization strategies to improve query execution performance in SUCXENT++. The relational query optimizer is unable to generate optimal query plans for the SQL queries generated based on the translation algorithm discussed in the previous chapter. Specifically, the join between the *Path* and *PathValue* tables is a major bottleneck. Replacing this with a predicate on the *PathId* (and *CPathId*) attribute in the *PathValue* table not only results in fewer joins in the query but also enables the relational query optimizer to generate more optimal query plans. This results in a performance improvement of up to 45 times over the non-optimized version of SUCXENT++. Now, SUCXENT++ performs comparably to the Shared-Inlining approach.

CHAPTER 5. QUERY OPTIMIZATION

Query	Shared-Inlining			Sucxent			XParent		
	10 MB	100 MB	1 GB	10 MB	100 MB	1 GB	10 MB	100 MB	1 GB
Q1	0.89	0.57	0.70	0.98	8.75	41.71	0.91	7.50	29.03
Q2	1.00	0.96	1.59	1.60	27.03	24.55	1.60	27.03	24.55
Q3	0.61	1.04	1.29	1.37	18.21	39.13	1.14	12.65	18.87
Q4	0.47	0.80	0.96	24.54	320.96	972.30	17.98	172.25	286.69
Q5	0.26	0.10	0.15	3.77	8.87	21.77	3.44	7.37	15.03
Q6	1.06	1.55	2.16	26.55	88.90	1629.09	17.62	39.15	315.91
Q7	0.23	0.33	0.92	0.57	1.26	7.07	0.53	1.10	5.39
Q8	0.91	0.77	0.64	10.62	472.97	NA	5.81	141.28	NA
Q9	0.76	0.93	1.07	8.30	9.48	16.67	6.66	6.11	6.93
Q10	0.52	0.16	0.12	153.33	NA	NA	107.66	NA	NA
Q11	0.95	0.33	0.22	84.84	475.41	1699.53	54.67	197.45	1155.55

Figure 5.13: Performance summary.

Query	Shared-Inlining			Sucxent			XParent		
	10 MB	100 MB	1 GB	10 MB	100 MB	1 GB	10 MB	100 MB	1 GB
Q12	1.07	0.67	0.92	1.67	1.51	2.74	1.67	1.81	3.26
Q13	1.96	1.08	0.91	2.67	2.32	3.40	3.62	4.27	11.44
Q14	2.54	2.64	3.85	5.42	30.30	59.15	6.67	45.77	134.97
Q15	1.04	0.17	0.10	2.1	2.55	13.14	3.1	5.56	62.41
Q16	0.30	0.51	1.89	7.67	4.47	6.20	10.02	7.63	18.06
Q17	26.12	9.14	13.9	2.62	2.61	5.05	4.2	4.33	14.06
Q18	10.56	20.63	33.01	29.88	158.11	967.1	39.11	270.9	NA
Q19	3.67	2.75	4.48	2.21	7.33	23.16	3.11	14.54	NA
Q20	2.59	1.88	1.04	6.07	3.96	4.14	5.93	7.34	5.07
Q21	16.38	9.89	5.58	6.2	3.79	3.70	6	5.20	5.62
Q22	69.46	87.84	68.71	6.4	9.66	8.55	6.7	12.38	14.81

Figure 5.14: Performance summary (contd).

5.2 Data Partitioning

Partitioning has been used effectively to improve the performance of relational data warehouses [18]. In the context of XML data management in relational databases, the improvement partitioning can bring is evident from the performance of the Shared-Inlining approach. One of the reasons Shared-Inlining performs better for certain queries than schema-oblivious approaches is that it can direct these queries to specific parts of the fragmented data. The queries operate on smaller data corresponding to only some of the tables in the entire schema. A similar approach of fragmenting data to optimize query execution can be used to improve the performance of SUCXENT++. To the best of our knowledge, no other schema-oblivious approach has been extended to incorporate partitioning.

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.15: Query performance

Query	Shared-Inlining	SUXENT	XParent
Q1	0.70	1.22	1.34
Q2	2.95	2.47	2.72
Q3	0.69	4.52	5.04
Q4	3.45	2.29	2.50
Q5	1.35	1.42	1.59
Q6	0.19	7.72	8.60
Q7	1.12	1.61	1.79
Q8	0.83	2.05	2.29
Q9	7.39	11.18	12.55
Q10	2.03	13.53	15.29

Figure 5.16: Summary.

5.2.1 Overview of Partitioning

The gains in performance that can be achieved by optimizing the join between the Path and PathValue tables are evident from Section 5.1.2. This operation can be further optimized by the judicious use of data partitioning. There are several ways in which partitioning can be done. We present an approach that takes into account the properties of the query workload to determine the partitioning strategy.

In our approach the query workload is first analyzed to determine how the PathValue table can be partitioned to optimize the execution of the queries in the workload. The partitions are then created and indexed. When data is added or deleted from the XML documents, these partitions are updated as well. Since the query workload determines the partitions, an update to the query workload also leads to an update of the partitions. The update process involves partition creation, deletion and modification of the partitions.

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.17: XQuery2 and SQL translation.

Figure 5.18: XQuery3 and SQL translation.

5.2.2 Problem Description

We first define the following terms to facilitate exposition.

Definition 5.9 [Partition] Let R_v be the **PathValue** relation. A relation V_i is a partition if $V_i \subset R_v$. ■

Consider the relations shown in Figure 5.19.a. The partitions PV1, PV2, PV3 and PV4 contain a subset of the rows in the **PathValue** relation of Figure 3.9.

Definition 5.10 [Partition-Set] Let $V_1, V_2, \dots, V_n$ be such that $V_1 \cup V_2 \cup \dots \cup V_n \subseteq R_v$. Then, the set $\mathbb{V} = \{V_1, V_2, \dots, V_n\}$ is called **Partition-Set**. I

The join between the **Path** and **PathValue** table essentially selects those rows in **PathValue** that match the given path expression. A reduction in the cost of this selection will improve query performance. Consider the query in

Figure 5.17 on the data in Figure 3.9. Notice that the query requires only three out of the nine paths (lines 4, 6 and 8 in Figure 5.17) of the **Path** table in Figure 3.9. Consider a Partition-Set $\mathbb{V} = \{V_1, V_2, \dots, V_i, \dots, V_n\}$ such that, each $V_i \in \mathbb{V}$ contains rows corresponding only to one (or more) of these paths. Then, these V_i can be used instead of R_v in the translated SQL query. A join operation between **Path** and $V_i \in \mathbb{V}$ will be less expensive as $|V_i| \ll |R_v|$.

We now present two approaches to partitioning that we have investigated - naive and query workload based. The naive approach is presented first and problems associated with this approach are discussed. This is followed by a discussion of query workload based partitioning and how it overcomes the deficiencies of the naive approach.

Naive Approach

Based on the discussion in the previous section, it may seem that minimizing $|V_i|$ will improve query performance. That is, query performance will improve the most where given any V_i, V_j such that $V_i \in \mathbb{V}$ and $V_j \in \mathbb{V}$, $V_i \cap V_j = \phi$. This implies that a separate partition is created for each unique value of *PathId* in the **PathValue** table. However, consider the queries shown in Figures 5.17 and 5.18. Figure 5.19.a shows the partitions obtained when this approach is applied and Figure 5.20.a shows the corresponding translated query for XQuery2. Here, PV1, PV2 and PV3 are the partitions for the three paths in the query in lines 4, 6 and 8 of Figure 5.17 respectively¹. This approach has several problems. First, the non-partitioned translation in Figure 5.17 has no **UNION** operator whereas the one in Figure 5.20.a has **UNION** operator. The **UNION** operator is required to consolidate the results returned from each individual sub-query that deals with a single path. This operation could have widely varying performance results depending on extent to which duplicates need to be eliminated in the resulting sub-query results. Although the **UNION** operator need not be introduced for all types of queries, it will have to be introduced for the following cases.

- (i) The query contains a disjunction. The query in Figure 5.17 is an example of this case.

¹PV4 is not used as it is a partition generated specifically for XQuery3 - as we will see later.

CHAPTER 5. QUERY OPTIMIZATION

PV1						
DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue	
1	1	3	2	14	Peptidyl alpha-amidating enzyme	
1	1	4	3	17	Peptidylglycine 2-hydroxylase	

PV2						
DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue	
1	8	5	2	24	AMD_BOVIN	
1	8	7	2	33	AMD_HUMAN	

PV3						
DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue	
1	5	2	2	7	Peptidylglycine monooxygenase	

PV4						
DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue	
1	6	1	0	0	1,14,17,3	

PV5						
DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue	
1	9	5	4	26	P19021	
1	9	8	4	35	P10731	

PV1'						
DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue	
1	1	3	2	14	Peptidyl alpha-amidating enzyme	
1	1	4	3	17	Peptidylglycine 2-hydroxylase	
1	8	5	2	24	AMD_BOVIN	
1	8	7	2	33	AMD_HUMAN	

PV2'						
DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue	
1	5	2	2	7	Peptidylglycine monooxygenase	

PV3'						
DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue	
1	6	1	0	0	1,14,17,3	

5.19.a: Partitions - Naive approach

5.19.b: Partitions - Query based approach

Figure 5.19: Partitions for data in Figure 3.9.

- (ii) The query contains path-expressions that resolve to more than one root-to-leaf path. Consider the query in Figure 5.22.a. The SQL queries for the non-partitioned and the naive partitioning approach are shown in Figure 5.23(a) and Figure 5.23(b). Notice that the root-to-leaf paths corresponding to the path expression in the return clause are fragmented across PV2 and PV5 requiring the UNION operator in line 7 (Figure 5.23.b) to combine the results.

Also, as the translated query is different from the non-partitioned version the query translation algorithm will have to be altered. Although any partitioning will lead to an alteration of the translation algorithm, it is possible to partition in a manner that minimizes this change (as we shall see in later sections). Finally, partitioning on *PathId* can take a significant amount of time if there are a large number of distinct *PathIds*.

Query Workload Based Approach

Now, consider the partitions in Figure 5.19.b and the corresponding translated query in Figure 5.21.a. The SQL query has no UNION operator. This change can be explained as follows. Lines 4 to 6 in Figure 5.17 indicate that two different path expressions are evaluated from the same relation P1. In Figure 5.20.a, the

CHAPTER 5. QUERY OPTIMIZATION

<pre> 01: Select v3.* from PV1 v1, PV3 v3 02: DocumentRValue r 03: Where 04: v1.LeafValue LIKE '%synthetase' AND v1.DocId = v3.DocId 05: AND v1.DocId = r.DocId AND r.Level = 1 06: AND abs(v1.BranchOrderSum-v3.BranchOrderSum)<r.RValue 07: UNION 08: Select v3.* from PV2 v1, PV3 v3 09: DocumentRValue r 10: Where 11: v1.LeafValue = 'AMD_HUMAN' AND v1.DocId = v3.DocId 12: AND v1.DocId = r.DocId AND r.Level = 1 13: AND abs(v1.BranchOrderSum-v3.BranchOrderSum)<r.RValue </pre>	<pre> 01: Select v2.* from PV4 v1, Path p1, 02: PV3 v2, Path p2, DocumentRValue r 03: Where 04: p1.Path = 'enzyme.db_entry.enzyme_id' 05: AND v1.LeafValue = '1.1.1.14' 06: AND p2.Path = 'enzyme.db_entry.enzyme_description' 07: AND p2.PathId = v2.PathId AND v1.DocId = v2.DocId 08: AND v2.DocId = r.DocId AND r.Level = 1 09: AND abs(v1.BranchOrderSum-v2.BranchOrderSum) < r.RValue </pre>
---	---

5.20.a: Translated query (XQuery2) for naive partitioning.

5.20.b: Translated query (XQuery3) for naive partitioning.

Figure 5.20: Translated queries for naive partitioning.

<pre> 01: Select v2.* from PV1' v1, Path p1, 02: PV2' v2, DocumentRValue r 03: Where 04: ((p1.Path LIKE 'enzyme.db_entry.%alternate_name 05: AND v1.LeafValue LIKE '%synthetase') OR 06: (p1.Path LIKE 'enzyme.db_entry. 07: swissprot_reference_list.reference.@name' 08: AND v1.LeafValue = 'AMD_HUMAN') AND p1.PathId=v1.PathId) 09: AND p2.Path = 'enzyme.db_entry.enzyme_description' 10: AND p2.PathId = v2.PathId AND v1.DocId = v2.DocId 11: AND v2.DocId = r.DocId AND r.Level = 1 12: AND abs(v1.BranchOrderSum-v2.BranchOrderSum) < r.RValue </pre>	<pre> 01: Select v2.* from PV3' v1, Path p1, 02: PV2' v2, Path p2, DocumentRValue r 03: Where 04: p1.Path = 'enzyme.db_entry.enzyme_id' 05: AND v1.LeafValue = '1.1.1.14' 06: AND p2.Path = 'enzyme.db_entry.enzyme_description' 07: AND p2.PathId = v2.PathId AND v1.DocId = v2.DocId 08: AND v2.DocId = r.DocId AND r.Level = 1 09: AND abs(v1.BranchOrderSum-v2.BranchOrderSum) < r.RValue </pre>
---	---

5.21.a: Translated query (XQuery2) for workload-based partitioning.

5.21.b: Translated query (XQuery3) for workload-based partitioning.

Figure 5.21: Translated queries for workload-based partitioning.

set of rows corresponding to P1 is fragmented across two partitions (PV1 and PV2) and as a result the UNION operator is required to combine the results. If these two path expressions are maintained in the same partition then the UNION operator will not be required. If the partition $PV1' = PV1 \cup PV2$ is generated instead of PV1 and PV2, the partitions will be as shown in Figure 5.19.b and the translated query will be as in Figure 5.21.a. This partitioning takes into account the fact that the two path expressions in the XQuery query in Figure 5.17 are referenced by the same relation in the SQL query. The partition PV1' contains rows that correspond to both path expressions. As a result the SQL query can use PV1' instead of PV1, PV2 of Figure 5.19.a without requiring an additional UNION operator.

The above example shows how a disjunction between path expressions can lead to the introduction of a UNION operator when naive partitioning is used.

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.22: Path expressions resolve to more than one root-to-leaf path.

Figure 5.22 also depicts a similar case. These examples illustrates that a more optimal $\mathbb{V}$ should be generated based on the properties of the query workload. A query workload is simply a set of $\mathcal{Q} = \{Q_1, Q_2, \dots, Q_i, \dots, Q_n\}$ where Q_i is an XQuery query. On one hand distinct partitions can be generated for each *PathId* leading to small partitions but requiring the introduction of additional **UNION** operators. On the other hand, *PathValue* need not be partitioned at all leading to one monolithic "partition" without the introduction of **UNION** operators in the SQL query. The aim of optimal partitioning is the generation of partitions that have minimal overlap without the introduction of additional **UNION** operators in the translated SQL queries. Formally,

Definition 5.11 [Optimal Partition-Set] Let $\mathcal{Q} = \{Q_1, Q_2, \dots, Q_i, \dots, Q_n\}$ be the query workload. Let t_i be the SQL translation of Q_i when no partitioning is employed and $S_i(\mathbb{V})$ be the SQL query for the Partition-Set $\mathbb{V}$. Let $U(s)$ be the number of **UNION** operators in an SQL query q . Then, the Optimal Partition-Set for $\mathcal{Q}$ is a Partition-Set $\mathbb{V}_{\mathcal{Q}}$ such that for any $V_i, V_j \in \mathbb{V}_{\mathcal{Q}}$, $i \neq j$, $|V_i \cap V_j|$ is minimum given the condition that, $\forall Q_i \in \mathcal{Q}$ if $U(t_i) = 0$ then $U(S_i(\mathbb{V}_{\mathcal{Q}})) = 0$. ■

Consider the query in Figure 5.20.a and the partitions in Figure 5.19.a. Based on the above definition the Partition-Set $\mathbb{V} = \{PV1, PV2, PV3, PV4\}$ is not an *Optimal Partition-Set*. This is because, the translation without any partitions (as shown in Figure 5.17) has no **UNION** operator whereas the query in Figure 5.20.a has a **UNION** operator. The partition-set $\mathbb{V} = \{PV1', PV2', PV3'\}$ (Figure 5.19.b) is the optimal partition-set in this case and the corresponding

CHAPTER 5. QUERY OPTIMIZATION

<pre> 01: Select v2.* from PathValue v1, Path p1, 02: PathValue v2, Path p2, DocumentRValue r 03: Where 04: p1.Path LIKE 'enzyme.db_entry.' 05: swissprot_reference_list.reference.@name' AND 06: p1.PathId=v1.PathId AND v1.LeafValue = 'AMD_HUMAN' 07: AND p2.Path LIKE 08: 'enzyme.db_entry.swissprot_reference_list.reference%' 09: AND p2.PathId = v2.PathId AND v1.DocId = v2.DocId 10: AND v2.DocId = r.DocId AND r.Level = 1 11: AND abs(v1.BranchOrderSum-v2.BranchOrderSum) < r.RValue </pre>	<pre> 01: Select v3.* from PV1 v1, PV2 v3 02: DocumentRValue r 03: Where 04: v1.LeafValue = 'AMD_HUMAN' AND v1.DocId = v3.DocId 05: AND v1.DocId = r.DocId AND r.Level = 1 06: AND abs(v1.BranchOrderSum-v3.BranchOrderSum)<r.RValue 07: UNION 08: Select v3.* from PV2 v1, PV5 v3 09: DocumentRValue r 10: Where 11: v1.LeafValue = 'AMD_HUMAN' AND v1.DocId = v3.DocId 12: AND v1.DocId = r.DocId AND r.Level = 1 13: AND abs(v1.BranchOrderSum-v3.BranchOrderSum)<r.RValue </pre>
--	--

5.23.a: SQL query - no partitioning

5.23.b: SQL query - naive partitioning

<pre> 01: Select v2.* from PV1' v1, Path p1, 02: PV1' v2, Path p2, DocumentRValue r 03: Where 04: p1.Path LIKE 'enzyme.db_entry.' 05: swissprot_reference_list.reference.@name' AND 06: p1.PathId=v1.PathId AND v1.LeafValue = 'AMD_HUMAN' 07: AND p2.Path LIKE 08: 'enzyme.db_entry.swissprot_reference_list.reference%' 09: AND p2.PathId = v2.PathId AND v1.DocId = v2.DocId 10: AND v2.DocId = r.DocId AND r.Level = 1 11: AND abs(v1.BranchOrderSum-v2.BranchOrderSum) < r.RValue </pre>
--

5.23.c: SQL query - Query based partitioning

Figure 5.23: Path expressions resolve to more than one root-to-leaf path.

query is shown in Figure 5.21.a. Notice that $PV3'$ is not used in the translation as it is required only for XQuery3 (Figures 5.18, 5.20.b and 5.21.b).

It is important to point out here that the above definition of an Optimal Partition-Set will not generate $\mathbb{V}$ that results in the lowest query evaluation cost for all cases - though it is still a significant improvement over the non-partitioned approach as we shall see in Section 5.2.6. There may be cases where the UNION operator would be less expensive than selecting from larger V_i . This approach is taken to simplify partition generation and avoid drastic changes to our query translation algorithm. Though, the translation algorithm is modified even with this approach, we will see in Section 5.2.4 that this modification is quite minimal. Future work will study truly optimal partitioning approaches which can possibly alter the translation algorithm more drastically.

5.2.3 Query Workload-based Partitioning

We have already elaborated on the deficiencies of naive partitioning. Query workload based partitioning is further explained here together with an algorithm

CHAPTER 5. QUERY OPTIMIZATION

<pre> for \$a, \$b, ..., \$z where ($k_a^1 \vee k_a^2 \vee \dots \vee k_a^n$) $\wedge \dots$ $\wedge (k_b^1 \vee k_b^2 \vee \dots \vee k_b^m)$ $\wedge \dots$ return $e_a^1, e_a^2, \dots, e_z^r$ </pre>	<pre> Select $V_{e_a^1}.*, V_{e_a^2}.*, \dots, V_{e_z^r}.*$ from PathValue $V_{k_a^1}$, Path $P_{k_a^1}$, ..., PathValue $V_{e_z^r}$, Path $P_{e_z^r}$ where (($P_{k_a^1}$.Path LIKE $k_a^1.lhs$ or ... or $P_{k_a^1}$.Path LIKE $k_a^n.lhs$) and $P_{k_a^1}$.Id = $V_{k_a^1}$.PathId and $V_{k_a^1}$.LeafValue = ...) and ... and $P_{k_b^1}$.Path LIKE $k_b^1.lhs$ and $V_{k_b^1}$.PathId = $P_{k_b^1}$.Id and ... and $P_{k_z^1}$.Path LIKE $k_z^1.lhs$ and $V_{k_z^1}$.PathId = $P_{k_z^1}$.Id and ... and $P_{e_a^1}$.Path LIKE e_a^1 and $V_{e_a^1}$.PathId = $P_{e_a^1}$.Id and ... and $P_{e_z^r}$.Path LIKE e_z^r and $V_{e_z^r}$.PathId = $P_{e_z^r}$.Id </pre>
5.24.a: XQuery	5.24.b: SQL translation

Figure 5.24: General XQuery and SQL translation.

<pre> {/enzyme/db_entry/alternate_name_list/ alternate_name, /enzyme/db_entry/ swissprot_reference_list/reference/@name}, {/enzyme/db_entry/enzyme_description} </pre>	<pre> {/enzyme/db_entry/enzyme_id}, {/enzyme/db_entry/enzyme_description} </pre>	<pre> {/enzyme/db_entry/ swissprot_reference_list/reference/ @name, /enzyme/db_entry/ swissprot_reference_list/reference/ swissprot_accession_number} </pre>
5.25.a: Q-Set for XQuery2	5.25.b: Q-Set for XQuery3	5.25.c: Q-Set for query in Figure 5.22.a

Figure 5.25: Q-Sets.

that generates the Optimal Partition-Set as specified in Definition 5.11.

Background

We have already highlighted how naive partitioning introduces additional UNION operators. This can be extended to a general XQuery² as shown in Figure 5.24.a. Notice that the XQuery where clause is represented in the *Conjunctive Normal Form*, e_a represents a path expression bound to \$a and k_a represents a predicate containing an expression bound to \$a. The SQL translation of the XQuery can be represented as shown in Figure 5.24.b based on the translation algorithm discussed in Chapter 4. There are two main observations here. First, path expressions in the same disjunction in the XQuery are referred to by the same alias in the SQL query if they are bound to the same variable. For example $P_{k_a^1}$ in the SQL query is used to refer to all the expressions in the first disjunction

²We have limited the scope of XQuery queries in this thesis as discussed in Chapter 4

CHAPTER 5. QUERY OPTIMIZATION

```

{/enzyme/db_entry/alternate_name_list/
alternate_name, /enzyme/db_entry/
swissprot_reference_list/reference/{name},
{/enzyme/db_entry/enzyme_description}, {/enzyme/
db_entry/enzyme_id}

```

Figure 5.26: Global set of Path-sets.

```

input:
1: XQuery query X
Output:
2: Q-Set  $\mathcal{P}_Q$ 
3: parse(X) /* Generates the parse tree. */
4: Q-Expression-Set  $\mathcal{E}_Q$ 
5: for all ComparisonExpr  $e_i$  in X do
6:   for all ComparisonExpr  $e_j$  in X do
7:     if  $e_i, e_j$  bind to the same variable then
8:       if  $e_i, e_j$  are in sam OR clause then
9:         place  $e_i, e_j$  in the same  $P_{e_k} \in \mathcal{E}_Q$ 
10:      end if
11:    end if
12:  end for
13: end for
14: Q-Set  $\mathcal{P}_Q$ 
15: for all  $P_{e_i} \in \mathcal{E}_Q$  do
16:   /* Find all root-to-leaf paths for each path express:
17:   /* in each Path-Expression-Set. This will require */
18:   /* querying the Path table. */
19:    $\mathcal{P}_Q = \mathcal{P}_Q \cup L(P_{e_i})$ 

```

Figure 5.27: Q-Set generation algorithm.

in the XQuery. Second, path expressions bound to different variables in the XQuery are referred to by different aliases in the SQL query.

From our discussion in the previous section and the general form of the translated SQL query we can infer that expressions referred to by the same alias should be maintained in the same partition. We will present some examples later on that will highlight this further. First, we present some definitions followed by the partition generation algorithm.

Definition

In order to ease exposition of the partition generation algorithm in Figure 5.28 the following terms are defined first.

Definition 5.12 [Path-set] Given an XML document X , let $P(X)$ be the set of all root-to-leaf paths in X . Let $X_1, X_2, \dots, X_k$ be the set of XML documents being queried by the query workload $\mathcal{Q}$. Then Path-set is a set $P = \{p_1, p_2, \dots, p_i, \dots, p_n\}$ such that $P \subset P(X_1) \cup P(X_2) \cup \dots \cup P(X_k)$. ■

CHAPTER 5. QUERY OPTIMIZATION

```

Input: Query load  $\mathcal{Q} = \{Q_1, Q_2, \dots, Q_n\}$  where  $Q_i$  is a query.
Output: Optimal Partition-Set  $\mathbb{V}_x$ 
1: Global set of Path-Sets  $\mathcal{P}_Q$ .
2: for all  $Q_i \in \mathcal{Q}$  do
3:   /*GetQSet() calls the Q-Set generation algorithm*/
4:    $\mathcal{P}_{Q_i} = \text{GetQSet}(Q_i)$ 
5:   for all  $P \in \mathcal{P}_Q$  do
6:     for all  $P_{Q_i} \in \mathcal{P}_{Q_i}$  do
7:       if  $P_{Q_i} \cap P \neq \emptyset$  then
8:         /*All these paths need to be in
9:         the same partition as they share Q-Sets.*/
10:         $P = P \cup P_{Q_i}$ 
11:      else if
12:        then
13:          /*New partition is added.*/
14:           $\mathcal{P}_Q = \mathcal{P}_Q \cup P_{Q_i}$ 
15:        end if
16:      end for
17:    end for
18:  /*Generate the set of partitioned relations*/
19:  for all  $P \in \mathcal{P}_Q$  do
20:    /* Generate the Path-Set-Partition and
21:    /* add to the Partition-Set. */
22:     $\mathbb{V}_x = \mathbb{V}_x \cup V(P)$ 
23:  end for

```

Figure 5.28: Partition generation algorithm.

The set $\{\text{/enzyme/db_entry/enzyme_id, /enzyme/db_entry/enzyme_description}\}$ is an example of a *Path-set*. It refers to the document in Figure 3.9 and the query in Figure 5.18.

Definition 5.13 [Path-Expression-set] Given an XML document X , let $P_e(X)$ be the set of all possible path-expressions in X . Let $X_1, X_2, \dots, X_k$ be the set of XML documents being queried by the query workload $\mathcal{Q}$. Then Path-Expression-set is a set $P_e = \{p_{e_1}, p_{e_2}, \dots, p_{e_i}, \dots, p_{e_n}\}$ such that $P_e \subset P_e(X_1) \cup P_e(X_2) \cup \dots \cup P_e(X_k)$. ■

The set $\{\text{/enzyme/db_entry/swissprot_reference_list/reference}\}$ is an example of a *Path-Expression-set*. It refers to the document in Figure 3.4 and the query in Figure 5.22.a.

Property 2 Given, a Path-expression-set $P_e = \{p_{e_1}, p_{e_2}, \dots, p_{e_i}, \dots, p_{e_n}\}$, the corresponding Path-Set is $L(P_e) = R(p_{e_1}) \cup R(p_{e_2}) \cup \dots \cup R(p_{e_i}) \dots R(p_{e_n})$ where $R(x)$ is the set of root-to-leaf paths that match the path expression x .

The set $L(P_e)$ for a Path-expression-set P_e can be obtained from the Path table. For example, if $P_e = \{\text{/enzyme/db_entry/swissprot_reference_list/ref}$

CHAPTER 5. QUERY OPTIMIZATION

erence) then, from Figure 3.9 $L(P_e) = \{ /enzyme/db_entry/swissprot_reference_list/reference/@name, /enzyme/db_entry/swissprot_reference_list/reference/swissprot_accession_number \}$.

The where clause of an XQuery Q can be expressed in the conjunctive normal form as $C_1 \wedge C_2 \wedge \dots \wedge C_k$ where C_i is a disjunction of predicates on path expressions. Let $E(C_i)$ be the set of path-expressions in C_i . Then, we define a Q-Expression-Set as follows.

Definition 5.14 [Q-Expression-Set] The Q-Expression-Set $\mathcal{E}_Q$ for a query Q is the set of Path-Expression-sets P_{e_i} such that if path expression $e_a, e_b \in P_{e_i}$, then $a = b$ and $\exists C_i$ such that $e_a, e_b \in C_i$. ■

Definition 5.15 [Q-Set] Given a query Q and its Q-Expression-Set $\mathcal{E}_Q = \{P_{e_1}, P_{e_2}, \dots, P_{e_n}\}$, the Q-Set is the set $\mathcal{P}_Q = \{L(P_{e_1}), L(P_{e_2}), \dots, L(P_{e_n})\}$. ■

Figure 5.27 depicts the algorithm for generating the Q-Set for a query. Note that two path expressions will be in the same disjunction (line 8 in the algorithm), if they are in the same disjunction when expressed in the Conjunctive Normal Form (DNF). This is the condition specified by Definition 5.14.

As an example Figure 5.25(a) shows the Q-Set for XQuery2. The two paths in the first Path-Set for XQuery2 are from the OR clause. The first path in this Path-Set is evaluated from the expression `/enzyme/db_entry//alternate_name` in XQuery2. Figure 5.25(c) shows the Q-Set for the query in Figure 5.22.a which contains a path expression that resolves to more than one root-to-leaf path.

We now explain why the Q-Set is chosen in this manner. Essentially, such paths when bound to the same variable in the XQuery query are selected from the same relation in the SQL query as shown in Figure and should therefore be in the same partition to avoid the UNION operator. In the SQL query shown in Figure 5.17 they are selected using the relation that corresponds to the alias P1. In the SQL query shown in Figure 5.23(a), the multiple paths resolved from the path expression in the return clause are selected from P2. The same holds true if p_l, p_m are resolved from the same regular path expression. Note that Q-Set determination does not require the SQL query to be generated. The algorithm in Figure 5.27 is used to identify the Q-Set.

CHAPTER 5. QUERY OPTIMIZATION

Definition 5.16 [Path-Set-Partition], $V(P)$ for a Path-Set $P = \{p_1, p_2, \dots, p_n\}$ is defined as follows.

$$V(P) = \Pi_{(DocId, PathId, LeafOrder, BranchOrder, BranchOrderSum, LeafValue)} R$$

$$where, R = \sigma_{(Path=p_1 \vee Path=p_2 \vee \dots \vee Path=p_n)} S$$

$$and, S = Path \bowtie_{(Path.PathId=PathValue.PathId)} R_v \blacksquare$$

Essentially, the Path-Set-Partition for a Path-Set is the result of partitioning the PathValue table on all the paths in the Path-Set. Consider the first Path-Set of the Q-Set shown in Figure 5.25. The relation PV1' in Figure 5.19.b is the Path-Set-Partition for this Path-Set. The above definitions facilitate the discussion of the partition generation algorithm.

Partition Generation Algorithm

The algorithm in Figure 5.28 executes as follows. First, the queries in the workload are parsed to generate their Q-Sets (line 4). The identification of Q-Sets can be done by parsing the query and then traversing the parse tree to identify paths that meet the criteria discussed in Definition 5.14. Then, the algorithm proceeds to identify Path-sets across Q-Sets that share path expressions (lines 5 to 7). These Path-sets are combined to produce a global set of Path-Sets, $\mathcal{P}_Q$ (line 10). This is done to avoid instances where the UNION operator - which would otherwise not be required - is used to combine the results from different partitions. $\mathcal{P}_Q$ is then used to generate the partitions (lines 19 to 23). Note that the partitions are represented as Path-Set-Partitions corresponding to the Path-Sets in the global set of Path-Sets.

As an example, consider the Q-Sets of XQuery2 and XQuery3 shown in Figure 5.25. Upon initialization, $\mathcal{P}_Q$ is the Q-Set of XQuery2. When the Q-Set of XQuery3 is considered it is found that a new Path-Set corresponding to `/enzyme/db_entry/enzyme_id` needs to be added to $\mathcal{P}_Q$. The second Path-Set, `{/enzyme/db_entry/enzyme_description}`, in the Q-Set of XQuery3 is already included in $\mathcal{P}_Q$. Therefore, the final $\mathcal{P}_Q$ is as shown in Figure 5.26. A total of three Path-sets are present in $\mathcal{P}_Q$ resulting in an equal number of partitions

CHAPTER 5. QUERY OPTIMIZATION

as shown in Figure 5.19.b. XQuery2 requires only the first two (PV1' and PV2') partitions whereas XQuery3 only requires the second and third (PV2' and PV3') partitions as shown in Figures 5.21.a and 5.21.b respectively.

The complexity of GetQSet () is $O(n^2)$ where n is the number of path expressions in the given query. The Q-Set of a query can be determined by the algorithm in Figure 5.27. Therefore, the complexity of the partition algorithm is $O(|Q|n^2mp)$ where m is the average number of Path-Sets for each Q-Set and p is the average number of Q-Sets across all queries.

However, this approach does have some limitations. The objective of partitioning is to reduce the time taken to select a subset of the the **PathValue** table. The combination of two or more PathId values in the same partition increases the size of the partition leading to an increase in the cost associated with this selection. A truly optimal approach would consider this trade-off.

I/O Cost - Partition Generation

The total I/O cost associated with partition generation can be calculated as follows.

$$C = E + X \quad (\text{Eq. 5.1})$$

where E is the cost associated with creating and inserting data into the partitioned tables and X is the cost of indexing the partitioned tables.

The I/O cost associated with extracting the partitioned data depends on the partition size. Therefore, if $V = \{V(P_1), V(P_2), \dots, V(P_n)\}$ is the Partition-Set then,

$$E = \sum_{i=1}^n (k_s \times |V(P_i)|)$$

where, k_s = cost of inserting a single row in a table. (Eq. 5.2)

Note that the manner in which C is calculated is independent of the partition-generation algorithm. However, C will be significantly different for the two algorithms discussed above. For the naive approach $|V(P_i)|$ will be a lower value whereas n will be smaller for the query-workload based approach.

CHAPTER 5. QUERY OPTIMIZATION

```

Input:
  XQuery query X
  Global set of Path-Sets  $\mathcal{P}_Q$ 
Output:
  Translated SQL query Q
1: parse( X ) /* generates the parse tree */
2: WhereClause w = Q.getWhereClause() /* this is determined by the t
3: SelectClause s = Q.getSelectClause() /* the return clause determ
4: for all ComparisonExpr  $e_i$  in X do
5:   if typeof( $e_i$ .lhs) is PathExpr and typeof( $e_i$ .rhs) is Literal then
6:     replace  $e_i$  with
7:     "Pathi.Path " + GetPathExprCondition(  $e_i$ .lhs );
8:     PathValuei = GetPartition( $e_i$ .lhs)
9:     +"and Pathi.Id=PathValuei.Id"
10:    +"and PathValuei.value <ValueComp>  $e_i$ .Literal"
11:   else if
12:   then
13:     replace  $e_i$  with /* this is a join expression */
14:     "Pathi.Path " + GetPathExprCondition(  $e_i$ .lhs )
15:     PathValuei = GetPartition( $e_i$ .lhs)
16:     +"and Pathi.Id = PathValuei.Id"
17:     +"and Pathi+1.Path " + GetPathExprCondition( $e_i$ .rhs)
18:     PathValuei+1 = GetPartition( $e_i$ .rhs)
19:     +"and Pathi+1.Id = PathValuei+1.Id"
20:     +"and PathValuei.value <ValueComp> PathValuei+1.value"
21:   end if
22:   w.add( $e_i$ )
23: end for
24: for all PathExpr  $p_i$  in X do
25:   for all PathExpr  $p_j \neq p_i$  in X do
26:     if  $p_i$  and  $p_j$  bind to the same variable then
27:       w.add("PathValuei.DocId=PathValuej.DocId")
28:        $l = l(p_i, p_j)$ 
29:       w.add("abs(PathValuei.BranchOrderSum"
30:       +"-PathValuej.BranchOrderSum) < l")
31:     end if
32:   end for
33:   if  $p_i$  in X.returnClause() then
34:     s.add("PathValuei.*")
35:   end if
36: end for
37: /* GetPartition() function. */
38: GetPartition( pathExpr ) {
39:   for all  $P \in \mathcal{P}_Q$  do
40:     for all  $P_{Q_i} \in \mathcal{P}_{Q_i}$  do
41:       if pathExpr  $\in P_{Q_i}$  then
42:         /* The partition has been found. Return the Path-Set-P.
43:         return  $V_{P_{Q_i}}$ 
44:       end if
45:     end for
46:   end for
47: }

```

Figure 5.29: Translation algorithm with partitions

5.2.4 Query Translation

In order to utilize the partitions in the translated query, the translation algorithm has to be modified. Figure 5.29 shows the translation algorithm when query workload based partitioning is employed. Notice that the only difference from Figure 4.14 is the addition of the function `GetPartition(pathExpr)` (lines 38 to 46). This returns the partition in which the given pathExpr occurs. This replaces the PathValue table (lines 8, 14 and 17).

Figure 5.21.a shows the translated SQL query for *XQuery2* (Figure 5.17) based on the modified translation algorithm. The only difference with respect to non-partitioned translation is the use of partitioned tables. The `GetPart it ion ()`

CHAPTER 5. QUERY OPTIMIZATION

```

Input: Current PathValue table  $VT$  and PathValue
         table after data modification  $VT_{new}$ . Optimal
         partition-set  $\mathbb{V}_x$ . Global Q-Set  $\mathcal{P}_Q = \{P_1, P_2, \dots, P_n\}$ 
Output: Updated partitions  $V(P_1), V(P_2), \dots, V(P_n) \in \mathbb{V}_x$ .
1:
2: for all  $V(P_i) \in \mathbb{V}_x$  do
3:   /* Check if partition needs to be updated */
4:   if  $V(P_i) \bowtie_{PathId} (VT_{new} - VT) \neq \phi$  then
5:     /*  $S(P_i)$  is the sub-set of  $(VT_{new} - VT)$ 
6:       that corresponds to Path-Set  $P_i \in \mathcal{P}_Q$ . */
7:      $V(P_i) = V(P_i) \cup S(P_i)$ 
8:   end if
9: end for

```

Figure 5.30: Partition maintenance on data/schema change.

function is used to evaluate these. The path expression `/enzyme/db_entry/alternate_name` can be found in the partition PV1' in Figure 5.19.b as it belongs to the first Path-Set in Figure 5.26. Similarly, `/enzyme/db_entry/swissprot_reference_list/reference/@name` is also in PV1'. The path `/enzyme/db_entry/enzyme_description` belongs to the second Path-set in Figure 5.26 and is therefore in the second partition PV2' in Figure 5.19.b. Therefore, the PathValue relation is replaced with PV1' and PV2' in the query.

PV1' (corresponding to the first Path-Set in Figure 5.26) and the first partition in Figure 5.19.b) and PV2' (corresponding to the second Path-Set). These partitions that replace the PathValue table in the query are determined by the `GetPartition(pathExpr)` function.

5.2.5 Partition Maintenance

An important aspect of data partitioning is the maintainability of the partitions. There are three main factors that influence partition management - change in the stored data, change in the structure of the stored data and change in the query workload. We discuss how these influence the partitioning algorithms and the associated I/O cost.

Data addition/removal

Data modification either results in the insertion of a new subtree, the removal of a subtree or the modification of the value of leaf node(s). In the first two cases leaf nodes will be added and deleted respectively. This will lead to the modification of Leaf Order, BranchOrder and BranchOrderSum attributes in

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.31: Partition maintenance on data addition.

SUCXENT++These attributes will have to updated irrespective of whether or not partitioning is used. Therefore, the cost of reevaluating these attributes will be incurred even without partitioning. The only additional cost partitioning will introduce is the update of existing partitions. Rows will be added to existing partitions when a new subtree is added and deleted when a subtree is removed.

As an example consider the addition of a new **reference** element to the **swissprot_reference_list** element in Figure 3.1. New rows are added to the PathValue table. The data inserted and the new rows added are shown in Figure 5.31(a) as shaded rows. Based on the algorithm in Figure 5.30 the effected partition is evaluated to be PV1'. A new row is added to this partition as shown in Figure 5.31(b).

Figure 5.32 shows an example when data is removed. Figure 5.32(a) shows the data that is removed and the resultant PathValue table. The updated partition is shown in Figure 5.32(b).

The complexity of the algorithm is $O(n)$ where n is the number of partitions. The I/O cost associated with this change is determined by the time required to insert/remove rows from the affected partitions. If R_v is the existing PathValue table and R'_v is the PathValue table after the update, then the I/O cost of maintaining $\mathbb{V}$ is

CHAPTER 5. QUERY OPTIMIZATION

<pre><reference siwssprot_accession_number="P10731" name="AMD_HUMAN"/></pre>					
--	--	--	--	--	--

DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue
1	6	1	0	0	1,14,17,3
1	5	2	2	7	Peptidylglycine monooxygenase
1	1	3	2	14	Peptidyl alpha-amidating enzyme
1	1	4	3	17	Peptidylglycine 2-hydroxylase
1	8	5	2	24	AMD_BOVIN
1	9	6	4	26	P19021

DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue
1	1	3	2	14	Peptidyl alpha-amidating enzyme
1	1	4	3	17	Peptidylglycine 2-hydroxylase
1	8	5	2	24	AMD_BOVIN

5.32.a: Updated **PathValue** after deletion.

5.32.b: Updated partition.

Figure 5.32: Partition maintenance on data removal.

$$C_D = k_s \times |I| + k_d \times |D|$$

where, $k_d = \text{cost of deleting a row.}$ (Eq. 5.3)

where $I = (R'_v - R_v)$ and $D = (R_v - R'_v)$. I is the set of rows added as a result of data modification and D is the set of rows deleted. C_D essentially shows the cost of adding/deleting rows.

Section 5.2.6 presents results that quantify the cost of this operation.

XML Schema/DTD Change

A change in the XML Schema/DTD can be modeled as the addition or deletion of PathIds. This will lead to the insertion/deletion of rows in the **PathValue** table. Therefore, the maintenance process will be the same as in the previously discussed instance - where data was modified.

Change in Query Workload

This modification applies only to workload-based partitioning. A change in the query workload essentially changes the Q-Set. This will alter the set of partitions. Existing partitions may have to be coalesced or separated. New partitions may need to be created. Figure 5.33 presents the algorithm. The complexity of this algorithm is $O(|\mathcal{P}_Q| |\mathcal{P}_Q|)$.

As an example, consider the query in Figure 5.34 being added to the query workload. The Q-Set of the query is shown in Figure 5.34 and the current

CHAPTER 5. QUERY OPTIMIZATION

```

Input: Query  $Q_i$ . Existing global Q-Set  $\mathcal{P}_Q$ 
Output: New optimal Partition-Set  $\mathbb{V}_x$ 
1:  $\mathcal{P}_{Q_i} = \text{GetQSet}(Q_i)$ 
2: for all  $P \in \mathcal{P}_Q$  do
3:   for all  $P_{Q_i} \in \mathcal{P}_{Q_i}$  do
4:     if  $P_{Q_i} \cap P \neq \emptyset$  then
5:       /*All these paths need to be in
6:       the same partition as they share Q-Sets.*/
7:        $P = P \cup P_{Q_i}$ 
8:       /* Update the partition
9:       to reflect new Path-Set.  $V(P) \in \mathbb{V}_x$  */
10:       $V(P) = V(P) \cup V(P_{Q_i} - P)$ 
11:     else if
12:       then
13:       /*New partition is added.*/
14:        $\mathcal{P}_Q = \mathcal{P}_Q \cup P_{Q_i}$ 
15:        $\mathbb{V}_x = \mathbb{V}_x \cup V(P_{Q_i})$ 
16:     end if
17:   end for
18: end for

```

Figure 5.33: Partition maintenance on query workload change.

```

for $b in document("enzyme")/enzyme/db_entry
where
$b//alternae_name LIKE '%oxygenase'
return $b/catalytic_activity

```

5.34.a: New query.

```

{/enzyme/db_entry/alternate_name_list/
alternate_name},
{/enzyme/db_entry/catalytic_activity}

```

5.34.b: Q-Set for new query.

Figure 5.34: Addition of new query to workload.

global Q-Set is shown in Figure 5.26. The first Path-Set in Figure 5.34 should be combined with the first Path-Set in Figure 5.26 as these Path-sets have common root-to-leaf paths (lines 4 to 7 in Figure 5.33). The second Path-Set generates a new partition as it cannot be combined with any existing Path-Set (lines 11 to 14). The final global Q-Set and the updated partitions are shown in Figure 5.35. PV4' is the new partition added and it corresponds to the last Path-Set in Figure 5.35(a).

Consider the query in Figure 5.22.a being added to the workload. Based on the Q-Set of this query (Figure 5.25(c)) the Path-Set of Figure 5.25(c) must be combined with the first Path-Set in Figure 5.35 to yield the final $\mathcal{P}_Q$ and partitions as shown in Figure 5.36. Here PV1' of Figure 5.19.b is extended to account for the new path in the first Path-Set of Figure 5.36(a).

A query workload change leads to the generation of a new $\mathcal{P}_{Q'}$ in the following ways.

CHAPTER 5. QUERY OPTIMIZATION

```

{/enzyme/db_entry/alternate_name_list/alternate_name,
/enzyme/db_entry/swissprot_reference_list/reference/
@name},{/enzyme/db_entry/enzyme_description}, {/enzyme/
db_entry/enzyme_id},{/enzyme/db_entry/catalytic_activity}

```

PV4'

DocId	PathId	LeafOrder	BranchOrder	BranchOrderSum	LeafValue
1	2	10	2	51	aminolevulinate

5.35.a: New $\mathcal{P}_Q$.

5.35.b: New partition added.

Figure 5.35: Partition modification due to query workload change (XQuery 4 added).

5.36.a: Final $\mathcal{P}_Q$.

5.36.b: Combined partitions.

Figure 5.36: Partition modification due to query workload change.

- Addition of new Path-Sets. The new Path-Sets are $\mathcal{P}_{Q'} - \mathcal{P}_Q$. $\{ /enzyme/db_entry/catalytic_activity \}$ is the new Path-Set for the example shown in Figures 5.34 and 5.35.
- Addition of paths to existing Path-Sets. Each Path-Set $P_i \in \mathcal{P}_Q$ is modified resulting in a new Path-Set P'_i such that $P'_i \in \mathcal{P}_{Q'}$. Essentially, an existing Path-Set in the global Path-Set is modified as a result of adding a new query to the workload. This is the example shown in Figure 5.36. Note that if a Path-Set does not change then $P'_i - P_i = \phi$. This is the case for the example shown in Figures 5.34 and 5.35.

Based on the above changes, the I/O-cost associated with partition-maintenance due to a query workload change can be characterized as follows.

$$C_Q = k_s \times |N| + k_s \times |M_I| + k_d \times |M_D| \quad (\text{Eq. 5.4})$$

CHAPTER 5. QUERY OPTIMIZATION

5.37.a: DCMD - 10 MB

5.37.b: DCSD - 10 MB

Figure 5.37: Query execution time (Data-Centric).

5.38.a: TCMD - 10 MB

5.38.b: TCSD - 10 MB

Figure 5.38: Query execution time (Text-Centric).

where $N = V(P_{k_1}) \cup V(P_{k_2}) \cup \dots \cup V(P_{k_m})$, $P_{k_1}, P_{k_2}, \dots, P_{k_m} \in (\mathcal{P}_{Q'} - \mathcal{P}_Q)$, $M_I = V(P_{1'} - P_1) \cup V(P_{2'} - P_2) \cup \dots \cup V(P_{n'} - P_n)$, $P_i \in \mathcal{P}_Q$, $P_{i'} \in \mathcal{P}_{Q'}$ and $M_D = V(P_1 - P_{1'}) \cup V(P_2 - P_{2'}) \cup \dots \cup V(P_n - P_{n'})$, $P_i \in \mathcal{P}_Q$, $P_{i'} \in \mathcal{P}_{Q'}$. N characterizes the cost of adding new partitions, M_I the cost of adding rows to existing partitions and M_D the cost of deleting rows from existing partitions. M_I and M_D together quantify the cost of modifying partitions.

Section 5.2.6 will present results that quantify the cost associated with partition maintenance when there is a change in query workload.

5.2.6 Performance Evaluation

Two sets of experiments were performed. The first experiment evaluates the benefit of partitioning by comparing the query execution time of the partitioned and non-partitioned approaches. In the second experiment the time required to

CHAPTER 5. QUERY OPTIMIZATION

5.39.a: DCMD - 100 MB

5.39.b: DCSD - 100 MB

Figure 5.39: Query execution time (Data-Centric)

5.40.a: TCMD - 100 MB

5.40.b: TCSD - 100 MB

Figure 5.40: Query execution time (Text-Centric).

update the partitions with changes in data and query workloads are measured. This quantifies the cost of partition maintenance.

Improvement in query execution time

Figure 5.37 to Figure 5.42 compare the performances of the 22 queries on the X Bench data set. In these figures SUCXENT++ represents the implementation without any optimizations. Figures 5.43 and 5.44 show a summary of the performance comparison. These figures show the ratio of time taken for a given approach to the time taken in SUCXENT++ after partitioning. There are two main observations based on these results.

- (i) There is a significant improvement in performance for every query. This is an expected result as queries are no longer executed on the entire Path-Value table. Instead, partitions that are much smaller in size are used.

5.41.a: DCMD - 1000 MB

5.41.b: DCSD - 1000 MB

Figure 5.41: Query execution time (Data-Centric).

5.42.a: TCMD - 1000 MB

5.42.b: TCSD - 1000 MB

Figure 5.42: Query execution time (Text-Centric).

The performance improvement will be greater for queries that execute on smaller partitions.

- (ii) The performance improvement due to data partitioning increases is more significant for larger data sets. This indicates that data partitioning allows in a more scalable storage mechanism. The performance improvement seen was as much as 450 times for the 1GB data set (Q18).

The improvement in performance is significant enough to outperform Shared-Inlining for 16 of the 22 queries. Recall that this number was 12 for the previous optimization based on query plan alteration as discussed in Section 5.1.4.

For the recursive data set, the greatest improvement in performance is seen for Q4 and Q6 as shown in Figures 5.45 and 5.46. This is because these queries execute on much smaller partitions as compared to the other queries.

CHAPTER 5. QUERY OPTIMIZATION

Query	Shared-Inlining			Sucxent			XParent		
	10 MB	100 MB	1 GB	10 MB	100 MB	1 GB	10 MB	100 MB	1 GB
Q1	1.40	1.31	1.12	1.43	17.15	46.45	1.54	20.01	66.74
Q2	9.38	8.59	10.25	15.00	242.84	158.54	15.00	242.84	158.54
Q3	1.08	1.57	1.46	2.03	19.11	21.38	2.43	27.52	44.34
Q4	0.79	0.74	0.76	30.08	158.67	227.79	41.06	295.64	772.54
Q5	0.70	0.53	0.53	9.18	37.49	54.29	10.07	45.11	78.63
Q6	1.78	3.50	2.91	29.53	88.25	425.59	44.50	200.40	2194.64
Q7	0.24	0.35	0.83	0.55	1.16	4.83	0.59	1.33	6.33
Q8	1.05	1.58	1.57	6.74	290.68	NA	12.34	973.08	NA
Q9	2.95	1.40	1.02	25.95	9.17	6.61	32.32	14.22	15.91
Q10	0.78	6.91	15.17	160.57	NA	NA	228.68	NA	NA
Q11	2.28	0.65	0.51	131.51	384.80	2642.01	204.07	926.49	3885.10

Figure 5.43: Performance summary.

Query	Shared-Inlining			Sucxent			XParent		
	10 MB	100 MB	1 GB	10 MB	100 MB	1 GB	10 MB	100 MB	1 GB
Q12	6.40	1.04	0.78	10.00	2.33	2.31	10.00	2.81	2.75
Q13	5.70	2.00	1.44	7.75	4.28	5.35	10.50	7.86	18.04
Q14	5.25	28.56	21.31	11.19	327.21	327.21	13.75	494.27	746.63
Q15	1.51	0.27	0.23	3.04	3.97	27.67	4.49	8.66	131.40
Q16	0.37	0.72	2.11	9.52	6.36	6.87	12.43	10.85	20.02
Q17	46.96	15.18	26.94	4.72	4.34	9.78	7.70	7.72	27.23
Q18	47.72	326.09	521.59	135.04	2498.24	15279.93	176.76	4280.33	NA
Q19	28.55	18.47	53.26	17.21	49.15	275.43	24.25	97.54	NA
Q20	4.24	4.26	1.89	9.94	8.95	7.49	9.70	16.6	9.17
Q21	54.60	12.18	11.92	20.67	4.67	7.82	20.00	6.41	12.22
Q22	119.76	135.76	85.21	11.03	14.93	10.55	11.55	19.14	19.83

Figure 5.44: Performance summary (contd.).

Partition maintenance cost

Three experiments were conducted to study the partition maintenance requirements. The four data sets were combined to produce data sets of 40MB, 400MB and 4GB. The query workload varied from a single query to all 22 queries.

- (i) *Effect of data size on partition maintenance.* The query workload used in this experiment had all 22 queries. The time required to generate the partition for the 40MB data set was measured. Next, data was added and the time required to update the partitions was measured. Figure 5.47 shows the results.
- (ii) *Effect of increasing query workload.* In this experiment the query workload is varied while keeping the data set constant at 4GB. The workload was varied from a single query to all 22 queries. Figure 5.48.a shows the results. The number of partitions created, deleted or modified as each query is added is shown in Figure 5.50.

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.45: Query performance

Query	Shared-Inlining	SUXCENT	XParent
Q1	1.50	2.62	2.87
Q2	9.18	7.67	8.45
Q3	3.33	21.91	24.43
Q4	48.03	31.95	34.80
Q5	4.72	4.98	5.56
Q6	0.54	22.31	24.83
Q7	9.38	13.50	14.98
Q8	3.28	8.125	9.09
Q9	21.64	32.75	36.74
Q10	4.71	31.38	21.79

Figure 5.46: Summary.

- (iii) *Effect of decreasing query workload.* The data set was kept constant at 4GB and the workload was reduced from 22 queries to 1 query. Figure 5.48.b shows the results. The number of partitions created, deleted or modified as each query is added is shown in Figure 5.51.

The first observation is that the increase in the partition maintenance cost is scalable - it is less than linear as the data set size increases from 40MB to 400MB to 4GB.

The second observation is that an increase in query workload has a greater impact than a decrease in query workload. This is expected as an increasing query workload leads to a rearrangement of existing partitions and creation of newer ones. Reduction in query workload, on the other hand, leads to either the complete deletion of existing partitions or deletion of a part of data from existing partitions. These are significantly less expensive operations than the creation of new partitions or addition of new data to existing ones. Only in

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.47: Partition maintenance cost - Data change.

5.48.a: Increasing workload

5.48.b: Decreasing workload

Figure 5.48: Increasing workload.

Figure 5.49: Partition maintenance cost - Query workload change.

one case (when Q21 is removed) does it lead to the creation of partitions. It is for this reason that removal of Q21 leads to the highest maintenance cost while reducing the query workload.

The third observation is that deletion of data from a partition is more expensive than the complete deletion of a partition. This is reflected in the costs as the workload reduces from Q15 to Q14 and Q9 to Q8. Here, data is removed from partitions (together with deletion of partitions) as opposed to just the removal of partitions in the other cases.

The I/O cost associated with creating a partition depends on the number of rows that need to be extracted from the `PathValue` table. When Q9 is added to the workload it leads to the creation of a new partition that has large

CHAPTER 5. QUERY OPTIMIZATION

Query added	Partitions created	Partitions deleted	Partitions modified
Q1	2	0	0
Q2	1	0	0
Q3	1	0	0
Q4	3	0	0
Q5	0	0	0
Q6	4	0	0
Q7	1	0	0
Q8	1	0	0
Q9	0	0	1
Q10	2	0	0
Q11	4	0	0
Q12	2	0	0
Q13	2	0	0
Q14	1	0	0
Q15	1	0	1
Q16	1	0	0
Q17	1	0	0
Q18	2	0	0
Q19	1	0	1
Q20	1	0	0
Q21	0	2	1
Q22	0	0	0

Figure 5.50: Partition change as workload increases.

number of rows extracted from the PathValue table. The cost of creating this partition is high resulting in a higher maintenance cost as reflected by the peak in Figure 5.48.a at Q9. The peaks at Q20 and Q21 can be explained similarly.

Based on the experimental results we can conclude partition maintenance cost is lower with a decreasing query workload than with an increasing workload. Also, it is more manageable when new partitions are created rather than when data is added/deleted from existing partitions.

5.2.7 Comparison with Existing Work

We have discussed some of the existing work in literature on partitioning in Section 2.6. Our work is different in the following ways.

- (i) These approaches focus on relational data with SQL as the query language whereas our work is specific to XML with XQuery as the query language - even though we use relational storage.
- (ii) The above approaches can be applied to any relational schema. Our approach is specific to the SUCXENT++ schema.
- (iii) Our approach may appear to be horizontal partitioning as the relations are partitioned that way. However, conceptually, it is more similar to

CHAPTER 5. QUERY OPTIMIZATION

Query added	Partitions created	Partitions deleted	Partitions modified
Q22	0	0	0
Q21	2	0	1
Q20	0	1	0
Q19	0	1	1
Q18	0	2	0
Q17	0	1	0
Q16	0	1	0
Q15	0	1	1
Q14	0	1	0
Q13	0	2	0
Q12	0	2	0
Q11	0	4	0
Q10	0	2	0
Q9	0	0	1
Q8	0	1	0
Q7	0	1	0
Q6	0	4	0
Q5	0	0	0
Q4	0	3	0
Q3	0	1	0
Q2	0	1	0
Q1	0	2	0

Figure 5.51: Partition change as workload decreases.

vertical partitioning. Keeping this in mind, our work considers intra-query dependencies whereas [38] only considers inter-query dependencies. The other approaches takes an integrated approach using both vertical and horizontal partitioning. Though the partitions in our approach are conceptually vertical we can combine horizontal partitioning to improve performance.

- (iv) We present performance results that quantify both the improvement in query execution time and the costs associated with partition maintenance.

To the best of our knowledge, there is little work in utilizing partitioning to improve the performance of XML storage approaches based on relational database systems. LegoDB [10], discussed earlier, comes closest to our work on data partitioning as it deals with relational storage of XML data. It is application-driven, i.e., the storage schema is generated based on the query workload and data statistics. Once generated, the schema is used throughout the lifecycle of the application - even if the workload or data change. Our work, on the other hand, maintains a fixed schema-oblivious relational schema. Partitions are generated based on this schema and the query workload. These partitions may change as the query workload and/or data change. The LegoDB approach uses

the recommendations of the relational query optimizer to provide an optimal schema for the given application. Our approach, on the other hand uses a cost model that relies only on the query workload and does not take into account data statistics. We have earmarked the development of a more sophisticated cost model that incorporates data statistics for future work. In [10], the authors do not elaborate on the approach taken when the query workload changes - although results are presented for queries that do not conform to the workload used to generate the schema. We present approaches to manage the partitions when data and/or the query workload changes and present results to highlight the cost of this maintenance operation.

5.3 GUI-driven Prefetching

XQuery is the current de facto standard XML query language. However, due to the nature of XML data, formulating an XML query using XQuery requires considerable effort. A user must be completely familiar with the syntax of the query language, and must be able to express his/her needs accurately in a syntactically correct form. In many real life applications it is not realistic to assume that users are proficient in expressing such textual queries. For example, warehousing and querying biological data is a key area of research in biological data integration. Recently, genomic data are increasingly being provided in XML format. New bioinformatics applications also increasingly take XML as input, making it essential that existing non-XML data be easily converted to XML. For instance, PIR, Entrez are already becoming XML compatible [6]. Several public domain and proprietary XML databanks such as the INTERPRO databank are already in existence. Hence, biologists need to be familiar with XML-like query languages to be able to formulate meaningful queries over these data sources. However, they cannot be expected to learn the complex syntax of query languages for integrating and querying biological data sources [53]. Hence, such data integration system needs to provide an easy and intuitive way of querying. This highlights the need for a user-friendly visual querying schemes to replace data retrieval aspects of XQuery.

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.52: Query interface.

We take a novel and non-traditional approach to improving query performance by *prefetching data during the formulation of a query*. The latency offered by the GUI-based query formulation is utilized to prefetch portions of the query results. In order to expedite XML query processing using such GUI-based prefetching three key tasks must be addressed. The first task is to design an intuitive GUI-based system. Second, GUI actions that can be used as indicators to perform prefetching need to be identified. Third, each GUI action can possibly lead to more than one prefetching operation. Therefore, an algorithm needs to be designed to select the "best" operation. In this section, we address these issues in detail.

A full implementation of the GUI-driven prefetching system would require a fully-functional XQuery support. However, building a system like this would take a significant number of person-years to implement. Instead, we implemented an interface that supports simpler types of XQuery. These queries are sufficient to justify the positive contributions made by the GUI-based prefetching technique. The type of queries supported are discussed in Section 5.3.1.

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.53: Query Editor.

5.3.1 Visual Query Interface

Anatomy of the GUI

The user interface (Figure 5.52) is presented as an adjustable multi-panel window comprising the following items. The *Repositories View* (labelled A) occupies the left pane. It serves as a data source browser in which the user can view the list of available data sources and their respective structures in terms of a tree display of their DTD/XML Schema. Showing multiple data sources allows the formulation of queries spanning more than one source. The data sources shown in Figure 5.52 are SWISSPROT and ENZYME.

The *Query Editors* are stacked in the middle pane (labelled C), with tabs for navigating between queries. The user drags the node to be queried from the Repository View and drops it in a Query Editor. A *Condition Dialog* (labelled E), appears and the user is expected to fill in the condition that should be satisfied by the selected node. In Figure 5.52, the selected node (labelled 1) is `/sptr/entry/feature/@type` and the condition is `"=transmembrane region"` thus forming the predicate `/sptr/entry/feature/@type="transmembrane region"` (labelled 2). Notice the similarity between this construct and **ComparisonExpr** of Figure 4.1. The visual representation of a **ComparisonExpr** type is referred to as **ExprBox**.

The user can combine two or more visual components that represent the **ComparisonExpr** by dragging a region around them and assigning an AND or OR condition. In Figure 5.53, the first two **ComparisonExpr** (labelled 2 and 3) are combined using the OR operator thus forming the **QueryExpr** (`/sptr/entry/feature/@type="transmembrane region" or /sptr/entry/organism/name="human"`). In order to specify a join condition two nodes, each representing one side of the join condition are selected and dragged on to the *Query Editor*. This is

CHAPTER 5. QUERY OPTIMIZATION

```

for $b in /sptr/entry, $c in /enzyme/entry
where
  ($b/feature/@type='transmembrane region'
or $/organism/name='human')
and $c/swissprot_reference/@swissprot_accession
  = $b/accession
return $b/sequence, $c/enzyme_id

```

Figure 5.54: Query.

shown by the labels 4 and 5 in Figure 5.52. The join condition represented by the visual query is `/sptr/entry/accession = /enzyme/entry//reference/@swissprot~accession`. The visual representation of a QueryExpr type is also referred to as ExprBox.

The *Selection View* (labelled *B*) is a drop target for nodes dragged from the *Repository View* and displays the nodes that will be visible in the result of the query. This enables the visual formulation and representation of the XQuery return clause. In order to facilitate prefetching, the GUI ensures that the user selects the nodes that should be present in the return clause before adding predicates to the query in the *Query Editor*. In Figure 5.52, the selection is `/sptr/sequence` - indicating that the user only wants to view the *sequence* element in the result. The final XQuery query can be interpreted from the *Query Editor* and *Selection View* components. Figure 5.54 shows the XQuery corresponding to Figure 5.52.

The user can execute the query by clicking on "Run" in the *Query Toolbar* (labelled *F*). The *Result View* (labelled *D*), displays the query results. Notice the similarity between the *Selection* and *Result* views.

Scope of Queries

Recall from Chapter 4 (Section 4.2) that the translation algorithms only allow for the automatic translation of queries that conform to the simplified grammar of Figure 4.4. As a result these restrictions also apply to the kind of queries that can be formulated using the GUI.

The GUI imposes restrictions in addition to those imposed by the modified syntax. Since, only leaf nodes are allowed to be selected for predicate assignment, all non-return clause path expressions in the resulting query will be

root-to-leaf path expressions. As a result queries with the **contains** or recursive path expressions cannot be formulated.

5.3.2 Query Formulation Time (QFT)

Our query processing approach utilizes the user's query formulation time to prefetch results of the intermediate queries. In order to determine the time available for prefetching (and to measure the improvement provided by prefetching) the time required to formulate a query visually needs to be measured. This is referred to as the query formulation time (QFT). It is the duration between the time the first predicate is added and the execution of the "Run" command as prefetching can start only when the first predicate is known. We have used the Keystroke-Level Model [13] to calculate QFT. Figure 5.57.b(a) lists average task times for the visual query formulation actions based on this model. Figure 5.57.b(b) further explains the estimation for the task of adding a non-join predicate. Timings for other tasks are calculated similarly. Notice that adding a join predicate takes less time than adding a non-join predicate as it merely involves selecting the two join nodes and dragging them on to the Query Editor. We first compute QFT in the absence of any query formulation error committed by the user. We call such QFT as error-oblivious query formulation time (EO_QFT). Note that the above model for calculating the QFT can as well be used for other types of visual XML query formulation systems (such as XQBE [12]). This is because similar actions would be required to formulate a query.

Error-oblivious QFT (EO_QFT)

Based on the timings discussed above the EO_QFT (denoted as T_f) for a query can be calculated as follows:

$$T_f = 7.4(x_{nj} - 1) + 3.2x_j + 1.3b + 0.7 \quad (\text{Eq. 5.5})$$

where x_{nj} is the number of non-join predicates, x_j is the number of join predicates, and b is the number of boolean operators in the query. Observe that $(x_{nj} - 1)$ is used as prefetching can start only when the first query formulation

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.55: Correcting mistakes.

step is complete. For the same reason, QFT does not include the time taken to add the return clause. If prefetching were to start as soon as the return clause were added, no useful intermediate results can be generated as no predicates have been added.

Error-conscious QFT (EC_QFT)

The above approach used to calculate error-oblivious QFT does not take into account errors committed by the user. But it is natural to assume that users may make mistakes while formulating queries. These errors are referred to as *query formulation errors* (QFE). Note that QFEs may impact our prefetching approach. Hence, it is necessary to quantify the effect of QFEs by extending QFT with the time lost due to QFEs. We first discuss how the GUI enables the user to correct queries by undoing certain actions. Then, we compute the *error-conscious* QFT (EC_QFT) that incorporates QFE.

The Visual Query Interface is extended to allow for corrections while formulating queries. Figure 5.52(b) shows the interface presented to the user. When

CHAPTER 5. QUERY OPTIMIZATION

the user discovers a mistake he/she clicks on the UNDO icon (labeled F in Figure 5.52(a)). The user is then presented with the list of actions he/she has performed (labeled G in Figure 5.52(b)). For example, in Figure 5.52(b) the list shows that the user has added two predicates and combined them using a conjunction. The user then selects the action(s) to be corrected. Suppose the user wanted the second predicate to be `.sptr.entry.comment.text="cardiac muscle"` instead of `.sptr.entry.comment.text="skeletal muscle"` in Figure 5.52(b). Consequently, the user has to modify the predicate by replacing `"skeletal muscle"` with `"cardiac muscle"`.

In general, a user will execute the following steps in our GUI to rectify a mistake.

- (i) *Click on UNDO icon.* This takes 0.7s based on the task times in Figure 5.57.c.
- (ii) *Select the action(s) to modify:* Each action selection will take 2.4s. As there can be k number of actions to be modified, the total time will be $2.4k$ seconds.
- (iii) One of the following actions need to be taken by the user depending on the mistake committed.
 - (a) *Modify the LHS (left hand side) of a non-join predicate.* This task will take 3.5s (calculated based on the first, second and fifth tasks in Figure 5.57.b(b)). If there are p_ℓ number of such corrections during query formulation, then the additional time taken is $3.5p_\ell$.
 - (b) *Modify the RHS (right hand side) of a non-join predicate.* This task will take 3.9s (based on the fourth and fifth tasks in Figure 5.57.b(b)). If the number of non-join predicates whose RHS needs to be corrected during query formulation is p_r , then the additional time taken is $3.9p_r$.
 - (c) *Modify the comparison operator of a non-join predicate.* This will take 2.2s (based on the third and fifth tasks in Figure 5.57.b(b)). Given p , comparison operator corrections during query formulation, the additional time taken will be $2.2p_c$.

CHAPTER 5. QUERY OPTIMIZATION

5.56.a: One formulation error.

5.56.b: Two formulation errors.

Figure 5.56: Examples of QFE.

- (d) *Modifying the LHS or RHS of a join predicate.* This will take $2.3s$ (based on second and the fifth tasks in Figure 5.57.b(b)). If there are p_j such modifications, then the additional time taken will be $2.3p_j$.
 - (e) *Changing a conjunction to a disjunction (or vice versa).* This will require $1.3s$ (based on the third task in Figure 5.57.b(a)). Given p_b such operations, the additional taken will be $1.3p_b$.
 - (f) *Deleting a predicate.* This will require $1.1s$ (based on the fifth task in Figure 5.57.b(b)). Given p_d such operations, the additional time taken will be $1.1p_d$.
 - (g) *Click on "OK" to accept the changes.* This will take $1.1s$ (last task in Figure 5.57.b(b)) and will have to be done for each modification. As a result, the total time taken for this operation is $1.1 \times (p_\ell + p_r + p_c + p_j + p_b + p_d)$.
- (iv) *Click on "OK" button in Figure 5.52(b).* This takes $1.1s$ (Figure 5.57.c).

Therefore, *each* time the *UNDO* icon is clicked and a set of mistakes is corrected by following the above steps, the *additional* time taken for formulating a query will be $(1.8 + 2.4k + T_u)$ where $0 < k \leq (p_\ell + p_r + p_c + p_j + p_b)$ and

$$\begin{aligned}
 T_u &= 3.5p_\ell + 3.9p_r + 2.2p_c + 2.9p_j + 1.3p_b \\
 &\quad + 1.1p_d + 1.1(p_\ell + p_r + p_c + p_j + p_b + p_d) \\
 &= 4.6p_\ell + 5p_r + 3.3p_c + 4p_j + 2.4p_b + 2.2p_d
 \end{aligned}
 \tag{Eq. 5.6}$$

The query formulation time, T_f can now be extended to incorporate query formulation error. If the user clicks on UNDO n times and corrects a set of

CHAPTER 5. QUERY OPTIMIZATION

mistakes each time then *error-conscious query formulation time* (denoted as T_{fe}) is given by the following equation.

$$T_{fe} = 7.4(m_{nj} - 1) + 3.2m_j + 1.3m_b + \sum_{i=1}^n (1.8 + 2.4k_i + T_{u_i}) + 0.7 \quad (\text{Eq. 5.7})$$

where k_i and T_{u_i} are the number of action selections and the total time taken to correct the mistakes for the i^{th} instance of the UNDO operation, m_{nj} , m_j , and m_b are the number of non-join predicates, number of join predicates, and the number of boolean operators *correctly* added during query formulation respectively. Note that m_{nj} , m_j , and m_b do not include those predicates and boolean operators that contain mistakes. These erroneous predicates and boolean operators are already incorporated in Equation Eq. 5.6.

5.3.3 Visual Query to XQuery Translation

In order to formulate a query, the user first selects the nodes that should be present in the RETURN clause. For instance, in Figure 5.52, the nodes selected are **sequence** and **enzyme_id** indicating that the user only wants to view these elements in the result. Next, the predicates in the **WHERE** clause are formulated in the *Query Editor*. The visual constructs in the *Query Editor* and *Selections View* need to be translated to formulate a complete XQuery. Recall that each visual component in the editor has a corresponding **ComparisonExpr** or **QueryExpr**. These can be combined to obtain a **Query** type. The translation to XQuery can be easily done by following the syntax presented in Section 4.2. Figure 5.54 shows the XQuery corresponding to Figure 5.52. Notice that the **ExprBox** with **.sptr.entry.feature. @type="transmembrane region"** is replaced by the **ComparisonExpr**\$b/feature/@type = "transmembrane region"****. Similarly, the composite **ExprBox** is replaced by the disjunction **\$b/feature/@type = "transmembrane region" or \$b/organism/name = "human"**. The join **ExprBox** is replaced by **\$b/accession = \$c/swissprot_reference/@swissprot_accession**. Also, the component in the *Selections View* is replaced by **\$b/sequence** and **\$c/enzyme_id** in the RETURN clause.

Figure 5.57: GUI task times.

5.3.4 Prefetching Model

We first define few terminology that will be used for our subsequent discussion on prefetching. Next, we describe our approach to improving query performance by utilizing the latency offered by GUI-based query formulation.

Definitions

Given an XML document and a path expression P the *Path Count*, C(P) is defined as the number of leaf nodes that satisfy P. The C(P) value for a root-to-leaf path P is the number of tuples in PathValue with path P. The Path table maintains this value for each Path in the PathCount attribute as shown in Figure 5.59. This figure is based on the decomposition shown in Figure 3.9. The C(P) value for a non-root-to-leaf path P is $\sum_{j=1}^n C(P_j)$ where $P_1, P_2, \dots, P_n$ are the root-to-leaf paths that satisfy P. Note that, as C(P) increases so does the I/O cost of a query that contains P as one of its path expressions. The *Total Path Count* for an XML document is defined as $T = \sum_{j=1}^N C(P_j)$ where N is the number of distinct root-to-leaf paths in the XML document.

Next, we define the notion of *value selectivity*. Given an XML document and a root-to-leaf path P, *value selectivity* V(P) is defined as the number of nodes in the XML document with Path P that have unique text values. This value is the number of tuples in PathValue for a given Path value that have unique LeafValue attributes. The Path table maintains this value for each Path in the ValueCount attribute as shown in Figure 5.59. Note that this corresponds to

CHAPTER 5. QUERY OPTIMIZATION

Document (<i>DocId</i> , <i>Name</i>)
Path (<i>PathId</i> , <i>Path</i> , <i>CPathId</i> , <i>Length</i> , <i>PathCount</i> , <i>ValueCount</i>)
PathValue (<i>DocId</i> , <i>PathId</i> , <i>CPathId</i> , <i>LeafOrder</i> , <i>BranchOrder</i> , <i>BranchOrderSum</i> , <i>LeafValue</i>)
LargeText (<i>DocId</i> , <i>PathId</i> , <i>LeafOrder</i> , <i>BranchOrder</i> , <i>BranchOrderSum</i> , <i>LeafValue</i>)
DocumentRValue (<i>DocId</i> , <i>Level</i> , <i>RValue</i>)

Figure 5.58: Modified schema.

Path				
PathId	PathExp	CPathId	PathCount	ValueCount
1	enzyme.db_entry.alternate_name_list.alternate_name	1	2	2
2	enzyme.db_entry.catalytic_activity	2	2	2
3	enzyme.db_entry.cofactor_list.cofactor	3	4	4
4	enzyme.db_entry.comment_list.comment	4	4	4
5	enzyme.db_entry.enzyme_description	5	1	1
6	enzyme.db_entry.enzyme_id	6	1	1
7	enzyme.db_entry.prosite_reference.prosite_accession_number	7	3	3
8	enzyme.db_entry.swissprot_reference_list.reference.@name	8	2	2
9	enzyme.db_entry.swissprot_reference_list.reference.swissprot_accession_number	9	2	2

Figure 5.59: Relational decomposition for modified schema to facilitate prefetching.

the Figure 3.9 of the previous chapter. Only the **Path** table is shown as it is the only table that is modified. The modified schema is shown in Figure 5.58.

Based on the above definitions, the cost of evaluating a **QueryExpr** κ , denoted as $cost(\kappa)$, can be calculated as follows.

- (i) If $\kappa ::= \text{PathExpr (ValueComp) Literal}$ then the usual procedure to estimate the I/O cost is followed. When **ValueComp** is "=" or ">" $cost(\kappa) = C(P) / V(P)$ or $cost(\kappa) = C(P) / 3$ respectively [19], where P is the parameter of type **PathExpr**. This can be extended to other types of **ValueComp**.
- (ii) If $\kappa ::= \text{PathExpr (ValueComp) PathExpr}$ and the two **PathExpr** types are denoted as P_1 and P_2 then $cost(\kappa) = \frac{C(P_1)}{V(P_1)} \times \frac{C(P_2)}{V(P_2)}$.
- (iii) If $\kappa ::= \text{ComparisonExpr (A) ComparisonExpr}$ and the two **ComparisonExpr** types are denoted as κ_1 and κ_2 then probability that κ_i ($i = 1, 2$) is satisfied is $\frac{cost(\kappa_i)}{T}$. Then, $cost(\kappa) = \frac{cost(\kappa_1) \times cost(\kappa_2)}{T}$.
- (iv) If $\kappa ::= \text{ComparisonExpr (V) ComparisonExpr}$ and the two **ComparisonExpr** types are denoted as κ_1 and κ_2 then $cost(\kappa) = cost(\kappa_1) + cost(\kappa_2)$.

CHAPTER 5. QUERY OPTIMIZATION

```

Input: Actions from the query interface.
Output: Intermediate materializations.
1: State S = getGUIState()
2: /*prefetch till user executes query*/
3: while S != "Execute Query" do
4:   /*Call materialization selection algorithm*/
5:   selectMaterialization()
6:   /*Call materialization replacement algorithm*/
7:   replaceMaterialization()
8:   S1 = getGUIState()
9:   while S1 == S do
10:    /*wait till GUI state changes*/
11:    S1 = getGUIState()
12:   end while
13: end while

```

Figure 5.60: Prefetching algorithm.

Note that the last two formulae can be extended for any number of conjunctions/disjunctions.

Prefetching Algorithm

As mentioned earlier, our approach utilizes the latency offered by GUI-based query formulation to expedite XML query processing. The objective of our approach is to perform prefetching operations at *certain* steps. In order to perform these operations, prefetching friendly GUI actions need to be identified first.

When a user formulates a query, constructs of types QueryExpr and CornparisonExpr are created. These types are parts of the final query and, therefore, are candidates for temporary materializations. Therefore, GUI actions that result in the addition of these types are also indicators for prefetching. These actions are: (1) the addition of an ExprBox and, (2) combining two or more ExprBox types to create another ExprBox type that corresponds to a QueryExpr type.

Next, given a GUI state, *the optimal prefetching operations need to be determined*. Finally, since each prefetching operation is useful for the next, *existing materializations need to be replaced with new materializations preferably using the previous materializations*. Figure 5.60 shows the overall prefetching algorithm. The process continues till the user clicks on "Run" to execute the query (line 3). The process waits for changes in the user interface (lines 8 to 12) before

CHAPTER 5. QUERY OPTIMIZATION

```

Input: Expressions  $\mathcal{K} = \{\kappa_1, \kappa_2, \dots, \kappa_n\}$  in descending order of
          $cost(\kappa_i)$ . Materialization limit  $L_M$ .
Output: Coefficient of each  $\kappa_i$  in the final materialization.
1:  $start = 0, end = 2^n - 1, middle$ 
2: while  $start < end$  do
3:    $middle = (start + end)/2$ 
4:   /*GetSelection(order, n) generates
5:   the coefficients for orderth combination out of  $2^{n-1}$ .*/
6:    $S = \text{GetSelection}(middle, n)$ 
7:    $l_S = \sum_{j=0}^n s_j \times cost(\kappa_j)$ 
8:   if  $l_m > L_M$  then
9:      $end = middle - 1$ 
10:  else if
11:    then
12:       $start = middle + 1$ 
13:  end if
14: end while
15: return  $\text{GetSelection}(middle, n)$ 

```

Figure 5.61: selectMaterialization().

selecting new materializations (line 5). Once new materializations are selected, existing ones are replaced (line 6). We elaborate on these two steps in turn.

Materialization Selection

At any given step during query formulation there can be more than one materialization option. Therefore, an algorithm that selects the "best" materialization is required. We present two heuristics that simplify the formulation of this algorithm.

*Heuristic 1. Consider only disjunctions of **ComparisonExpr** and **QueryExpr** as candidates for temporary materializations.* While formulating queries the GUI contains n **ComparisonExpr** and **QueryExpr** types, $\kappa_i, i=1 \dots m$. The possible materializations are $(\kappa_1 \vee \kappa_2 \vee \kappa_3 \vee \dots \vee \kappa_n), (\kappa_1 \wedge \kappa_2 \vee \kappa_3 \vee \dots \vee \kappa_n), (\kappa_1 \wedge \kappa_2 \wedge \kappa_3 \vee \dots \vee \kappa_n)$ etc. The number of possible combinations is 2^{n-1} . Obviously, evaluating all possible materializations, though guaranteed to generate a useful materialization, is not feasible. Therefore, only disjunctions are generated. This is because given $\kappa_1, \dots, \kappa_n, (\kappa_1 \wedge \kappa_2 \wedge \dots \wedge \kappa_n)$ can be evaluated from the materialization of $(\kappa_1 \vee \kappa_2 \vee \dots \vee \kappa_n)$.

Heuristic 2. Given a materialization space limit L_M , include the maximum possible number of expressions κ_i in the materialization. This is because, the

CHAPTER 5. QUERY OPTIMIZATION

greater the number of expressions included in the current materialization the greater the usefulness of the intermediate result towards evaluating the final result.

Based on the above heuristics we define a materialization selection and the optimality of a materialization selection. Given n expressions $\kappa_1, \kappa_2 \dots \kappa_n$ in the GUI, a materialization selection is defined as $S = \{\mu_1, \mu_2, \dots, \mu_n\}$ where $\mu_i \in \{0, 1\}$ and the cost associated with the selection (which is the same as the result size) is calculated as $l_S = \sum_{i=1}^n \text{cost}(\kappa_i) \times \mu_i$. Essentially, an expression κ_i is included in the materialization if $\mu_i = 1$. The cost l_S is a summation as only disjunctions are considered based on the first heuristic.

The optimality of a materialization selection, $\Theta(S)$, is defined such that given two materialization selections $S_a = \{\mu_{a_1}, \mu_{a_2}, \dots, \mu_{a_n}\}$ and $S_b = \{\mu_{b_1}, \mu_{b_2}, \dots, \mu_{b_n}\}$, $\Theta(S_a) > \Theta(S_b)$ if and only if $(\sum_{i=1}^n \mu_{a_i} > \sum_{i=1}^n \mu_{b_i}) \vee (\sum_{i=1}^n \mu_{a_i} = \sum_{i=1}^n \mu_{b_i} \wedge l_{S_a} > l_{S_b})$. This optimality condition satisfies the second heuristic.

Consider a GUI state with three expressions κ_1, κ_2 and κ_3 such that $\text{cost}(\kappa_1) > \text{cost}(\kappa_2) > \text{cost}(\kappa_3)$. The most desirable materialization selection would be $S = \{1, 1, 1\}$ as it will include all the expressions. However, if $l_S > L_M$ then selections with only two expressions will have to be considered. Then, the optimal materialization would be $S = \{1, 1, 0\}$ as it includes the expressions that will yield the largest result. This can be extended to generate the sequence $\Theta(\{1, 1, 1\}) > \Theta(\{1, 1, 0\}) > \Theta(\{1, 0, 1\}) > \dots > \Theta(\{0, 0, 1\}) > \Theta(\{0, 0, 0\})$. This sequence can be generated for any number of expressions n .

The input to the algorithm is the list of **ComparisonExpr** and **QueryExpr** types, κ_i , currently present in the GUI. They are listed in decreasing order of $\text{cost}(\kappa_i)$ as discussed above. The algorithm in Figure 5.61 essentially performs a binary search over this sequence to determine the best materialization given the limit L_M . Notice that the sequence need not be pre-generated. The `GetSelection` method returns a selection S given its order in the sequence and the number of expressions n . For example, in the case where $n = 3$, `GetSelection(3,3)` would return $\{1, 0, 1\}$ - the third selection for three rules. Similarly, `GetSelection(1,3)` would return $\{1, 1, 1\}$. The time-complexity of `GetSelection` is $O(n^2)$. A binary search through the 2^n possible materialization

CHAPTER 5. QUERY OPTIMIZATION

```

Input:
  GUI state, last GUI operation op and current set of
  materializations  $M$ .
Output: Updated  $M$ .
1: if op is add then
2:   /* refer to Figure 9 */
3:   selectMaterialization()
4: end if
5: if op is combine then
6:    $e_1$  and  $e_2$  are the combined ExprBox types
7:    $m_1$  and  $m_2$  are the corresponding materializations.
8:   if op is OR then
9:     if  $m_1$  and  $m_2$  then
10:       $m = m_1 \cup m_2$ 
11:       $M = M - (m_1) - m_2$ 
12:       $M = M \cup m$ 
13:    end if
14:    if ! $m_1$  or ! $m_2$  then
15:       $M = \text{selectMaterialization}()$ 
16:    end if
17:  end if
18: if op is AND then
19:   /*  $m_1$  and  $m_2$  have already been materialized */
20:   if  $m_1$  and  $m_2$  then
21:      $m = \text{GetCommonNodes}(m_1, m_2)$ 
22:      $M = M - (m_1) - m_2$ 
23:      $M = M \cup m$ 
24:   end if
25:   /*  $m_1$  or  $m_2$  or both are not materialized yet */
26:   if ! $m_1$  or ! $m_2$  then
27:      $M = \text{selectMaterialization}()$ 
28:     if  $(m_1 \wedge m_2) \in M$  then
29:       /* Use  $m_1$  or  $m_2$  to generate the new SQL query */
30:       SQL  $s = \text{SQL query using only } m_1 \text{ and } m_2$ .
31:        $M = M - (m_1 \text{ and } m_2)$ 
32:       materialize  $s$ .
33:        $M = M \cup s$ 
34:     end if
35:   end if
36: end if
37: end if
38: if op is UNDO then
39:   Cancel ongoing materialization.
40:    $M_D = \text{completed materializations dependant on}$ 
   step being corrected.
41:   for all  $m_d \in M_D$  do
42:     Delete  $m_d$ .
43:   end for
44: end if
45: /*materialize the new ComparisonExpr or QueryExpr types in  $M^*$ ,
46: materializeNew()

```

Figure 5.62: Algorithm replaceMaterialization.

selections has $O(\log(2^n))$ complexity. Therefore, the overall time-complexity is $O((n^2)(\log 2^n)) = O(n^3)$.

Once the list of κ_i s is selected by the algorithm a separate materialization, denoted as $\mathcal{M}_{\kappa_i}$ is maintained for each κ_i . A disjunction of the selected κ_i s could be maintained instead. The cost for both is approximately the same and is equal to $\sum \text{cost}(\kappa_i)$.

Materialization Replacement

Once the optimal materialization to replace the current state is selected it needs to be generated preferably using the results from the previous materializations. The materialization replacement algorithm is presented in Figure 5.62. The worst case complexity of the replacement algorithm without executing the new materialization is $O(n^3)$ - when **selectMaterialization0** is called. The overall time taken depends on the execution time the SQL query(s) correspond-

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.63: Prefetching steps.

Figure 5.64: Prefetching scenarios.

ing to the new materialization. We illustrate this algorithm with a simple example. Consider the query shown in Figure 5.52. The following prefetching operations take place during the formulation of the first two predicates:

- (1) Add the **ComparisonExpr** `/sptr/entry/featuer/@type="transmembrane region"`. The system materializes this query based on the materialization selection algorithm (lines 1 to 4 in Figure 5.62). Figure 5.63(a) shows the corresponding SQL query. Here **mat1** is the temporary table holding the materialization results.
- (2) Add the **ComparisonExpr** `/sptr/entry/organism/name="human"`. The algorithm selects both **ComparisonExpr** for materialization (again, lines 1 to 4 in Figure 5.62). However, only the second one needs to be materialized (line 40 in Figure 5.62). Figure 5.63(b) shows the corresponding SQL query. Now, the user can combine these two expressions using an AND or OR condition.

CHAPTER 5. QUERY OPTIMIZATION

- **AND Condition.** The result is the join of the two materializations `mat1` and `mat2` on their `DocId` and `LeafOrder` attributes to select only those tuples that satisfy both conditions (lines 19 to 22 in Figure 5.62). Figure 5.63(c) shows the corresponding SQL query.
- **OR Condition.** The result is the UNION of the two previous materializations to remove the duplicates (lines 9 to 13 in Figure 5.62). The corresponding SQL query is shown in Figure 5.63(d).

In Section 5.3.5, the cost associated with materialization replacement is presented in the form of the cost associated with generating all the intermediate materializations.

Benefit Estimation for Prefetching

In this section, we quantify the benefit of prefetching in the absence of QFEs. In the next section, we shall elaborate on the benefit of error-conscious prefetching. We begin by defining few terms to facilitate exposition.

The response time as perceived by the user when prefetching is not employed is called the normal execution time (NET)(denoted as T_n). The perceived response time (PRT) is the query response time when prefetching is employed. In the absence of QFEs, we refer to the PRT as error-oblivious perceived response time (EO_PRT). If QFEs are present then we refer to the PRT as error-conscious perceived response time (EC_PRT).

In order to quantify the benefit of prefetching, the best and worst case scenarios for PRT should be measured. We illustrate this with an example. Consider the query in Figure 5.52(a). The best case scenario for EO-PRT would be when the first materialization corresponding to `. sptr . entry. feature .@type="trans membrane region"` is completed before the user adds the second predicate `. sptr. entry. organism.name="human"`. In such a case, the user can view the results almost as soon as the execution command is given by clicking on "Run". The EO_PRT would simply be the network latency combined with the time required to construct the results and display them in Figure 5.52(a). In the worst-case scenario, none of the intermediate materializations finish before the next visual query formulation step.

CHAPTER 5. QUERY OPTIMIZATION

Theorem 5.2 Let T_f , T_p , T_δ be the EO_QFT, EO_PRT, network latency and result display time respectively. Then,

$$(T_\delta \leq T_p \leq ((\sum_{i=1}^{i \leq n} t_i) - T_f) + T_\delta)$$

where n is the total number of query formulation steps and t_i is the time for the i^{th} prefetching materialization.

Proof: Let $f_1, f_2, \dots, f_n$ be the formulation time taken for n steps in the GUI. Let t_i be the time for the i^{th} prefetching materialization where $0 < i \leq n$. In the best-case scenario every prefetching operation is completed before the user visually formulates the next step of the query. In this case $t_i \leq f_i \forall 0 < i \leq n$. Therefore, $\sum_{i=1}^n t_i \leq T_f$. Consequently, after clicking on "RUN" (n th step) the time taken to display query results is only T_δ . In the worst-case scenario none of the intermediate materializations finish before the next visual query formulation step. Hence, the worst-case value of T_p can be calculated as follows:

$$T_p = (T_{mat} - T_f) + T_\delta = ((\sum_{i=1}^{i \leq n} t_i) - T_f) + T_\delta$$

where T_{mat} is the total time for prefetching materialization. Hence, EO_PRT will have a value between these two cases. ■

Benefit Estimation for Prefetching with QFE

A user may commit any number of mistakes at different steps of query formulation. As a result he/she may invoke the UNDO operation multiple times to rectify the mistakes. For the sake of clarity, in the following discussion we assume that the UNDO operation has been invoked only once. Our model can easily be extended to handle the case for multiple invocation of UNDO operations. We present the best case and the worst case impact of the UNDO operation on the PRT. We begin by defining the notion of *error realization distance*.

Consider a query with n formulation steps where the user clicks on "Run" at n th step. Let the error is committed at p th step and the UNDO operation is invoked at q th step where $0 < p < q \leq n - 1$. Then, the *error realization distance*, denoted as ϵ , is defined as $\epsilon = q - p$. For example, in Figure 5.52(a) suppose the user commits a mistake at Step 1 (label 2) and realizes it before

CHAPTER 5. QUERY OPTIMIZATION

clicking on "Run" (the last step). Hence, the undo operation is invoked before the "Run" operation (Step 6). Therefore, $\epsilon = 5$. Observe that ϵ is maximum when the mistake is committed at the first step and UNDO is invoked only before the last step ((n-1)th step). Therefore, $\epsilon_{max} = n - 2$. Similarly, ϵ is minimum when the mistake is committed at the pth step and UNDO is invoked at (p+1)th step. Therefore, $\epsilon_{min} = 1$.

When the UNDO operation is invoked at step q the intermediate materializations till q that are dependant on step p (step where the mistake is made) can no longer be used. These materializations will have to be reevaluated once the mistake is corrected. Also, any materialization currently in progress that depends on step p will have to be canceled as it will unnecessarily take up database resources. Observe that the materialization reevaluation can follow an eager or a lazy approach. In the eager approach, the materialization reevaluation is triggered as soon as the user clicks on the UNDO icon in Figure 5.52 (label F). In the lazy approach, it is triggered only when the user clicks on the "OK" button in Figure 5.52(b). We follow the latter approach as the user may mistakenly click on the UNDO icon or he/she may cancel the UNDO operation. As a result, the eager approach will result in prematured reevaluation of materialized results.

Hence, the cost of materialization reevaluation (denoted as C_{me}) can be quantified as follows.

$$C_{me} = T_{mc} + T_{md} + T_{mr} \quad (\text{Eq. 5.8})$$

where T_{mc} is the time taken to cancel an ongoing materialization, T_{md} is the time taken to delete the intermediate materializations that are affected by the error and T_{mr} is the time taken to regenerate the deleted materializations based on the corrected step. The above equation can be expressed as follows:

$$C_{me} = T_{mc} + \sum_{i=p}^{i \leq q} d_i + \sum_{i=p}^{i \leq q} t_i \quad (\text{Eq. 5.9})$$

where d_i is the time required to delete the materialization due to i th step, t_i is the time taken to evaluate the materialization due to i th step. Note that $d_i > 0$ only if step i depends on the results of step p . That is, only the materializations whose result depends on the corrected step will be deleted. Also, $t_i > 0$ only if

CHAPTER 5. QUERY OPTIMIZATION

$d_i > 0$. That is, only the deleted materializations are regenerated as these are the only ones dependent on the corrected step. Furthermore, $T_{mc} > 0$ only if the ongoing materialization depends on the step being corrected. Next, using the above materialization reevaluation cost we compute the best and worst case value of EC_PRT .

Theorem 5.3 *Let T_{pe} , T_δ , T_{fe} be the EC_PRT , network latency and result display time, and EC_QFT respectively. Then,*

$$\begin{aligned} & (T_{mc} + t_{md_q} + t_{mr_q} + \sum_{j=1}^{n-1} t_j - T_{fe}) + T_\delta \leq T_{pe} \\ & \leq (T_{mc} + \sum_{i=1}^{n-1} d_i + \sum_{i=1}^{n-1} t_i - 0.7) + T_\delta \end{aligned}$$

where n is the total number of query formulation steps and $0 < q < n$.

Proof: We prove the above theorem by computing T_{pe} for the best case and the worst case scenario. The best case scenario occurs when the following conditions are satisfied. First, the error realization distance is minimum ($\epsilon_{min} = 1$). That is, the user made a mistake at step q and realized it before he/she formulates the next step and rectifies the mistake. Second, the materialization due to step q is independent of the materialized results created before the step q . Therefore, $C_{me} = T_{mc} + t_{md_q} + t_{mr_q}$ where t_{md_q} and t_{mr_q} are times taken to delete the materialized result created at step q and the time taken to regenerate the deleted materialization respectively (based on Equation Eq. 5.8). Note that $t_{md_q} > 0$ if $T_{mc} = 0$ and viceversa. Hence, $T_{pe} = (C_{me} + T_{mat}' - T_{fe}) + T_\delta$ where T_{mat}' is the total time taken to generate intermediate materializations for formulation steps except the one which is regenerated at $(p + 1)$ th step. Therefore, $T_{mat}' = \sum_{j=1}^{n-1} t_j$. This is equivalent to the LHS of the above theorem. Note that T_{fe} can be computed using Equation Eq. 5.7.

The worst case scenario occurs when the following conditions are satisfied. First, the error realization distance is maximum ($\epsilon_{max} = n - 2$). That is, the user made a mistake at the first step and realized it just before he/she clicks the "Run" icon. Second, all the materialization needs to be reevaluated as they are all dependent on the materialization results of the first step. This results in the

CHAPTER 5. QUERY OPTIMIZATION

Data	URL	Size	Node Count	Leaf Count	Depth
Swiss-Prot	http://us.expasy.org	600 MB	26,035,596	17,305,288	6
EMBL	http://ebi.ac.uk	600 MB	15,255,784	13,462,624	6
Enzyme	http://ebi.ac.uk	3 MB	86,413	74,892	5
Total		1203 MB	41,387,293	30,832,704	

(a) Data Set

#	Query	Characteristics	T_p (s)	Result Size
1	for \$b in /spttr/entry where \$b/protein/name = 'Sanguitrapne Cyclase' return \$b/accession	- Database: Swiss-Prot - Single Predicate	0.7	3
2	for \$b in /spttr/entry where \$b/entryType = 'transmembrane region') and \$b/organism/name = 'human' return \$b/accession	- Database: Swiss-Prot - Two Predicates - Conjunction	9.4	2836
4	for \$b in /spttr/entry where (\$b/keyword = 'Chloride Channel' or \$b/comment/text = 'skeletal muscle') return \$b/accession,\$b/sequence	- Database: Swiss-Prot - Two Predicates - Disjunction	9.4	145
5	for \$b in /spttr/entry where (\$b/keyword = 'Chloride Channel' or \$b/comment/text = 'skeletal muscle') and \$b/organism/name='human' return \$b/accession,\$b/sequence	- Database: Swiss-Prot - Three Predicates - Disjunction - Conjunction	10.1	43
6	for \$b in /embl/entry where \$b/keyword = 'Igasea' return \$b/accession	- Database: EMBL - Single Predicate	0.7	3349

(b) Queries

#	Query	Characteristics	T_p (s)	Result Size
6	for \$b in /embl/entry where \$b/source/organism='Homo Sapiens' and \$b/keyword = 'Igasea' return \$b/accession	- Database: EMBL - Two Predicates - Conjunction	9.4	3278
7	for \$b in /embl/entry where \$b/descr = 'Igasea' or \$b/keyword = 'Igasea' return \$b/accession	- Database: EMBL - Two Predicates - Disjunction	9.4	3883
8	for \$b in /embl/entry where (\$b/descr = 'Igasea' or \$b/keyword = 'Igasea') and \$b/source/organism = 'Homo Sapiens' return \$b/accession	- Database: EMBL - Three Predicates - Disjunction/Conjunction	18.1	3596
9	for \$b in /spttr/entry, \$c in /embl/entry where \$b/protein/name = 'Sanguitrapne Cyclase' and \$b/dbReference[6id-\$c/ accession] return \$b/accession, \$c/accession	- Database: Swiss-Prot, EMBL - Single Predicate - Predicate on Swiss-Prot	5.2	3
10	for \$b in /spttr/entry, \$c in / enzyme_pathway/entry, \$d in /embl/entry where \$d/keyword = 'Igasea' and \$b/ accession-\$c/swissprot_reference/ reference and \$b/ dbReference[6id-\$d/accession] return \$b/accession, \$c/accession	- Database: Swiss-Prot, Enzyme and EMBL - Two Predicates - Predicate on EMBL and Swiss-Prot	10.4	60

(c) Queries (contd.)

Figure 5.65: Data set and queries.

largest number of materialization regenerations coupled with the least amount of available formulation time (0.7s for clicking the "Run" icon) for prefetching. Therefore, in the worst-case $T_{pe} = (T_{mc} + T_{md} + T_{mr} - 0.7) + T_{\delta}$. Replacing the values of T_{md} and T_{mr} , we will get the RHS of the inequality in Theorem 2. ■

5.3.5 Performance Evaluation

The prototype system of GUI-driven prefetching technique was implemented using JDK1.5. The visual interface was built as a plug-in for the Eclipse [50] platform. The RDBMS used was SQL Server 2000 running on a P4 1.4GHz machine with 256MB RAM.

Data Set

The experiments were carried out with three data sets of size 300MB, 600MB and 1200MB respectively generated by combining the data sets shown in Figure 5.65(a). The 300MB data sets was generated using 150MB each of the SWISS-PROT and EMBL data sets. The 600MB data set was generated using 300MB each and the 1200MB data set was generated using the complete data sets. The 3MB ENZYME data set was used in all experiments. It is not reflected in the respective sizes due to its much smaller size. Ten queries were used to test the system. These queries vary in the number of predicates, conjunctions/disjunctions and result size. The list of queries together with their EO_QFT values and query results size for 1200MB data is shown in Figure 5.65.

CHAPTER 5. QUERY OPTIMIZATION

5.66.a: 300 MB

5.66.b: 600 MB

5.66.c: 1200 MB

Figure 5.66: Materialization replacement cost.

5.67.a: 300 MB

5.67.b: 600 MB

5.67.c: 1200 MB

Figure 5.67: NET vs. TPT.

Experimental Results

Materialization Replacement Cost: Figure 5.66 shows the results of materialization replacement cost. Here the running times of individual materialization operations are presented. Each section of the stacked columns represents the running time associated with the corresponding materialization. For example, Q1 has two formulation steps and, therefore, two sections in the corresponding stacked column. There are two main observations. First, the increase in the running times as the data set size increases is less than linear. Therefore, the cost associated with materialization replacement is scalable. Second, the replacement cost for disjunctions is less than that for conjunctions. This is reflected in the results for queries involving disjunction (Q3, Q4, Q7 and Q8) as opposed to queries involving conjunction (Q2, Q6, Q9 and Q10). This is expected as the materialization selection algorithm selects materializations with disjunctions (Heuristic 1 in Section 5.3.4). As a result, evaluating conjunctions would involve an additional step.

NET vs TPT: This experiment compared the total time taken for all prefetching operations (Total Prefetching Time), denoted as TPT, with normal query

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.68: NET vs EO-PRT.

execution time (denoted as NET). This experiment is required to test the viability of prefetching. A large TPT value relative to NET would indicate that prefetching adds a significant overhead and is, therefore, not viable. In addition, TPT also represents the cost associated with materialization replacement.

Figure 5.67 shows the results for this experiments. There are three main observations. The first is that the difference is not significant indicating that prefetching is a viable option. The second observation is that the conjunctive queries show a smaller difference than disjunctive ones. This is because conjunctive queries are evaluated from the corresponding disjunction based on materialization selection/replacement algorithms. This means that conjunctive queries will have a more significant prefetching overhead. This observation can be extended to queries that proceed from less selective partial queries to more selective final queries during formulation. The final observation is that for some of the queries the sum of the prefetching operations is less than the actual query execution time. This difference increases with data set size. This is because the relational query optimizer is now able to generate better query plans.

NET vs EO_PRT: The next experiment compares the normal execution time (NET) with the *error-oblivious* perceived execution time (EO_PRT). This comparison is done as a percentage of improvement over normal execution. It is measured as $improvement = (1 - \frac{EO_PRT}{NET}) \times 100$. Figure 5.68 show the results for the three data sets. There are two main observations. First, the improvement in performance is more for larger data sets. For the 300MB data set the improvement range is 7-76%. This range increases to 16-89% for the 600MB data set and 47-96% for the 1200MB data set. The second observation is that

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.69: EC_PRT vs NET.

Figure 5.70: EC_PRT vs EO_PRT (1).

simple queries (Q1, Q5 and Q9) with one predicate and small result sets benefit the least. Queries with multiple predicates and large result sets benefit the most. This is indeed encouraging as query response time is more critical for large data set. Also queries with disjunctions benefit more than the queries with conjunctions. This is expected as the materialization selection algorithm selects disjunctions as the intermediate results. Q2 seems to go against this observation. As mentioned earlier, this is due to the wide gap in the optimality of the query plans generated in the two approaches.

NET vs EC_PRT: In this experiment we evaluate the effect of QFE on perceived response time (EC_PRT) over normal execution time (NET). This comparison is done as a percentage of improvement over normal execution. It is measured as $\text{improvement} = (1 - \frac{EC_PRT}{NET}) \times 100$. In this experiment we present the worst-case value for EC_PRT as discussed in Section 5.3.4. This will give us an estimate of the upper-bound of the effect QFE can have on prefetching. The results for are presented in Figure 5.69. We only take the smallest and the largest data sets (300MB and 1200MB) for this experiment. The main obser-

CHAPTER 5. QUERY OPTIMIZATION

Figure 5.71: EC_PRT vs EO_PRT (2).

vation is that EC_PRT is still significantly better than NET for most queries. Also observe that similar to EO_PRT, there is larger improvement for larger data size and smaller improvement for simpler queries. Hence, QFEs do not significantly affect the query performance improvement achieved by GUI-driven prefetching.

EC_PRT vs EO_PRT: This comparison is done to measure the penalty on PRT due to QFE. It is measured as $penalty = \frac{EC_PRT - EO_PRT}{EO_PRT} \times 100$. Again, the worst case value of EC_PRT is used for comparison. Particularly, we measure EC_PRT for $q = n - 1$ and vary error realization distance.

Figures 5.70 and 5.71 show the results for the 300MB and 1200MB data sets respectively. Figure 5.70(a) shows the results for queries that have three formulation steps (two predicates and a conjunction/disjunction) other than clicking on "Run" and Figure 5.70(b) shows the results for queries with five formulation steps. The three values shown for each query in Figure 5.70(a) measure the penalty when the error was committed at the first step, the second step and the third step respectively. The *penalty* axis starts at -5 to allow the display of cases where $penalty = 0$.

The results shown highlight two main points. First, QFE generally has a greater effect with the increase in error realization distance. This is expected as an early mistake will lead to more materializations being recalculated. However, there are some exceptions. The query Q2 for the 1200MB data set shows an increase as the evaluation of the second predicate is more expensive than the first. Similar phenomenon is observed for queries Q4, Q8 and Q10. Second,

CHAPTER 5. QUERY OPTIMIZATION

the impact of QFE increases with data set size. The 1200MB data set shows a maximum increase of 316%. The 300MB data set shows a maximum increase of 187%. The impact of QFE is felt on only four queries for the 300MB data set whereas all queries are effected for the 1200MB data set. This can be attributed to the higher cost of reevaluating materializations for the larger data set.

5.3.6 Summary

In this section, we have discussed how the latency offered by visual query formulation can be utilized to prefetch partial results so that the final query can be answered in a shorter time. We have implemented our approach on top of a novel efficient schema-oblivious storage approach for XML data. Experimental evaluation indicates that prefetching is viable as the combined time taken by all the prefetching operations is not significantly more than normal query execution time. In fact, for some queries the total time taken by all prefetching operations is less than the normal execution time due to a better query plan generated by the relational query optimizer. Also, the overhead due to prefetching reduces with increase in data size. Prefetching benefits complex queries with large results the most. Such queries proceed through multiple materializations. Consequently, the final partial query is on a much smaller data set. Simple queries with very small result sets benefit the least. Our experiments also show that prefetching improves the perceived query response time by 7-96% with a greater improvement for larger data sets. In addition, QFE has no influence on the perceived response time - at least for the queries studied here.

5.4 Summary

In this chapter we have presented optimization techniques to speed up query processing in SUCXENT++. We demonstrated the inadequacy of the relational query optimizer and strategies to overcome it. In addition, a query workload-based data partitioning approach was also presented. Our experiments indicate that these optimizations can result in an improvement of up to 450 times with an increasing improvement as data size increases. In some cases, the improvement is significant enough to outperform Shared-Inlining.

CHAPTER 5. QUERY OPTIMIZATION

We also presented our work on GUI latency-driven prefetching. Experiments indicate that this is a viable option and can result in an improvement of up to 96% in the perceived response time. This improvement becomes more significant as the data size increases.

We must highlight here that the optimization approaches discussed here can be applied to other relational-storage strategies e.g. XParent, Suixent and Shared-Inlining as well. The query plan alteration approach can easily be applied to XParent, Suixent and other approaches that store path expressions. The data-partitioning approach can be applied to any storage strategy.

In the next chapter we will conclude the thesis and present directions for future work.

Chapter 6

Conclusions and Future Work

The popularity of XML has lead to a plethora of data that is less structured and does not follow a pre-defined schema. As a result, there is a growing need for data management systems that allow the storage and querying of such data. One avenue is the use of existing database technology. The goal of this thesis is to investigate relational storage approaches for XML data.

In this chapter we first highlight the contributions made in the thesis. Next, we present the conclusions of our implementations and experiments. Finally, based on our findings possible directions for future work are highlighted.

6.1 Contributions

We have extensively investigated the use of relational database systems for XML data management. To that end, the following contributions have been made.

- (i) We have designed and implemented a schema-oblivious relational storage approach called SUCXENT. It has several novel features, the most important of which is that it is based on determining leaf-node path intersection relationships. Prior work in this area has focused on evaluating ancestor-descendant relationships when executing queries.
- (ii) SUCXENT, though better than existing schema-oblivious approaches in terms of query performance has several drawbacks. We have implemented another approach - SUCXENT++ - that follows a similar approach to query processing but overcomes the disadvantages of SUCXENT by using a novel approach to detecting path intersection relationships.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

- (iii) In order to utilize the flexibility offered by XML, XQuery queries need to be executed transparently in both SUCXENT and SUCXENT++. To that end, we have proposed and implemented algorithms that translate XQuery based queries to the corresponding SQL queries.
- (iv) We studied the performance of SUCXENT and SUCXENT++ and compared it with existing approaches - including schema-conscious approaches such as Shared-Inlining. We found that the reasons for the poor performance of schema-oblivious approaches were twofold. First, the relational query optimizer is unable to generate optimal query plans for the translated SQL queries. Second, schema-conscious approaches can take advantage of the fragmented nature in which XML is stored and direct queries to specific portions of the data. To resolve these performance bottlenecks we have proposed and implemented two optimization strategies.
 - (a) The first optimization strategy modifies the query in a manner that reduces the number of joins and allows the relational query optimizer to generate optimal query plans.
 - (b) The second strategy utilizes query workload-based partitioning. Existing partitioning approaches usually partition tables without regard to the query workload.

Current work in the area of relational storage of XML tables has shown that a schema-conscious approach such as Shared-Inlining performs better than schema-oblivious approaches. Our initial experiments verified that this is true for most types of queries. However, certain types of recursive queries perform better in SUCXENT++. With optimizations, SUCXENT++ performs substantially better for some of these recursive queries and is comparable to Shared-Inlining for most types of non-recursive queries as well.

- (v) We have devised a novel GUI-based query system. We have also formulated a cost-based prefetching model to utilize the latency offered by visual query formulation to prefetch query results.

6.2 Conclusions

We carried out several experiments based on the contributions discussed in the previous section. The key findings of these studies are as follows.

- (i) A query execution strategy based on evaluating path intersection relationships rather than ancestor-descendant relationships is a viable approach. In fact, both SUCXENT and SUCXENT++ perform better than existing schema-oblivious approaches by up to 2 and 150 times respectively.
- (ii) Schema-conscious approaches perform better than schema-oblivious approaches for most types of queries if no optimizations are employed. As mentioned earlier this is mainly due to the inadequacy of the relational query optimizer and the inability of schema-oblivious approaches to direct queries at smaller (and more relevant) portions of the data.
- (iii) Schema-oblivious approaches can perform better than schema-conscious approaches for recursive queries. This is due to the fragmented nature of the data stored in a schema-conscious approach.
- (iv) Optimizations to SUCXENT++ yield tremendous improvements in performance. The first optimization itself results in an improvement of up to 45 times. Coupled with data partitioning this can go up to 450 times.
- (v) In spite of the optimizations SUCXENT++ does not outperform Shared-Inlining for all types of queries. In fact, without partitioning, it outperforms Shared-Inlining for only 12 out of the 22 queries. This indicates that an approach that is entirely schema-conscious or schema-oblivious may not be appropriate for all types of applications. A hybrid approach that combines the benefits of schema-oblivious and schema-conscious approaches would be a better option. Another approach would be to select the relational schema based on the application type.
- (vi) GUI-based prefetching is a viable optimization option. It leads to an improvement in the perceived response time of up to 96%. This improvement increases with an increase in the size of the data being queried.

6.3 Future Works

Based on our findings we believe that the following present promising directions for future work.

- (i) Extending the query translation algorithms. The current query translation algorithms are fairly limited in the types of XQuery queries they can translate. Both the storage approaches allow for the expression of most types of XQuery queries. Therefore, the query translation algorithms need to be extended to take full advantage of SUCXENT and SUCXENT++.
- (ii) The detection of intersection relationships between paths in SUCXENT++ requires an operation fairly similar to the detection of ancestor-descendant relationships (except over a much smaller set of data) - the θ -join. Prior work has highlighted the inadequacy of the relational query engine to process this type of a join efficiently. Instead structural join algorithms have been proposed. An interesting direction for future work could be the adaptation of such algorithms to the SUCXENT++ approach.
- (iii) The current query workload-based partitioning algorithm does not generate optimal partitions as it is not a cost-based approach. A formal cost-model needs to be developed to facilitate this.
- (iv) A framework needs to be developed to ease the maintenance of the partitions generated as a result of the query workload-based algorithm. This is especially required when the application has varying query load.
- (v) The current visual query interface does not allow for the visual formulation of several types of queries e.g., quantification, aggregation, conditions on non-leaf nodes etc.. Effective ways of visual expressing these query types need to be studied and the GUI needs to be extended.
- (vi) The effectiveness of the current materialization selection algorithm needs to be analyzed further. Experiments to evaluate the efficacy of this algorithm and the materialization replacement algorithm need to be devised and conducted. We have not done this yet as there is no basis for comparison.

Appendix A

SQL Queries

A.1 SQL Queries for XBenck in SUCXENT

```
Q1: SELECT
      V2.*
    FROM
      PATHVALUE V1,
      PATHVALUE V2,
      PATH P1,
      PATH P2,
      ANCESTORINFO A1,
      ANCESTORINFO A2
    WHERE
      P1.PATH = '.ORDER.@ID' AND
      P1.[ID]=V1.PATHID AND
      V1.[VALUE]='1' AND
      P2.PATH = '.ORDER.CUSTOMER_ID' AND
      P2.[ID]=V2.PATHID AND
      V1.DOCID = A1.DOCID AND
      V2.DOCID = A2.DOCID AND
      A1.ANCESTORLEVEL = 1 AND
      A2.ANCESTORLEVEL = 1 AND
      A1.DOCID = A2.DOCID AND
      A1.ANCESTORORDER = A2.ANCESTORORDER
```

```
Q2: SELECT
      V1.*,
      V4.*,
      V5.*,
      V6.*
    FROM
      PATHVALUE V1,
      PATH P1,
      PATHVALUE V2,
      PATH P2,
      PATHVALUE V3,
      PATH P3,
      PATHVALUE V4,
      PATH P4,
      PATHVALUE V5,
      PATH P5,
      PATHVALUE V6,
      PATH P6,
      ANCESTORINFO A1,
      ANCESTORINFO A2,
      ANCESTORINFO A3,
      ANCESTORINFO A4,
      ANCESTORINFO A5,
```

APPENDIX A

```

ANCESTORINFO A6
WHERE
P1.PATH = 'ORDER.@ID' AND
V1.PATHID = P1.[ID] AND
V1.[VALUE]='6' AND
P2.PATH = 'ORDER.CUSTOMER_ID' AND
V2.PATHID = P2.[ID] AND
P3.PATH = 'CUSTOMERS.CUSTOMER.@ID' AND
V3.PATHID = P3.[ID] AND
V3.[VALUE]=V2.[VALUE] AND
P4.PATH = 'CUSTOMERS.CUSTOMER.FIRST_NAME' AND
V4.PATHID = P4.[ID] AND
P5.PATH = 'ORDER.ORDER_STATUS' AND
V5.PATHID = P5.ID AND
P6.PATH = 'CUSTOMERS.CUSTOMER.LAST_NAME' AND
P6.ID = V5.PATHID AND
V1.DOCID = A1.DOCID AND
V2.DOCID = A2.DOCID AND
A1.ANCESTORLEVEL = 1 AND
A2.ANCESTORLEVEL = 1 AND
A1.DOCID = A2.DOCID AND
A1.ANCESTORORDER = A2.ANCESTORORDER AND
V5.DOCID = A5.DOCID AND
A5.ANCESTORLEVEL = 1 AND
A1.DOCID = A5.DOCID AND
A1.ANCESTORORDER = A5.ANCESTORORDER AND
V2.DOCID = A3.DOCID AND
V4.DOCID = A4.DOCID AND
A3.ANCESTORLEVEL = 1 AND
A4.ANCESTORLEVEL = 1 AND
A3.DOCID = A4.DOCID AND
A3.ANCESTORORDER = A4.ANCESTORORDER AND
V6.DOCID = A6.DOCID AND
A6.ANCESTORLEVEL = 1 AND
A3.DOCID = A6.DOCID AND
A3.ANCESTORORDER = A6.ANCESTORORDER

Q3: SELECT
V1.*
FROM
PATHVALUE V1,
PATHVALUE V2,
PATH P1,
PATH P2,
ANCESTORINFO A1,
ANCESTORINFO A2
WHERE
P1.PATH = 'ORDER.@ID' AND
P1.[ID]=V1.PATHID AND
V1.[VALUE]='2' AND
P2.PATH LIKE 'ORDER.ORDER_LINES.ORDER_LINE%' AND
P2.[ID]=V2.PATHID AND
V1.DOCID = A1.DOCID AND
V2.DOCID = A2.DOCID AND
A1.ANCESTORLEVEL = 1 AND
A2.ANCESTORLEVEL = 1 AND
A1.DOCID = A2.DOCID AND
A1.ANCESTORORDER = A2.ANCESTORORDER

Q4: SELECT
V2.*
FROM
PATHVALUE V1,
PATHVALUE V2,
PATH P1,
PATH P2,
ANCESTORINFO A1,

```


APPENDIX A

```

ANCESTORINFO A2
WHERE
P1.PATH = '.ORDER.@ID' AND
P1.[ID]=V1.PATHID AND
V1.[VALUE]='2' AND
P2.PATH = '.ORDER.ORDER_LINES.ORDER_LINE.ITEM_ID' AND
P2.[ID]=V2.PATHID AND
V1.DOCID = A1.DOCID AND
V2.DOCID = A2.DOCID AND
A1.ANCESTORLEVEL = 1 AND
A2.ANCESTORLEVEL = 1 AND
A1.DOCID = A2.DOCID AND
A1.ANCESTORORDER = A2.ANCESTORORDER

Q5: SELECT
V3.*
FROM
PATHVALUE V1,
PATHVALUE V2,
PATH P1,
PATH P2,
PATHVALUE V3,
PATH P3,
,
ANCESTORINFO A1,
ANCESTORINFO A2,
,
ANCESTORINFO A3
WHERE
V1.PATHID = P1.PATHID AND
P1.PATH = '.ORDER.TOTAL AND
V2.PATHID = P2.PATHID AND
P2.PATH = '.ORDER.SHIP_TYPE' AND
(
P3.PATH = '.ORDER.@ID' OR
P3.PATH = '.ORDER.ORDER_DATE' OR
P3.PATH = '.ORDER.SHIP_TYPE'
) AND
P3.PATHID = V3.PATHID AND
CONVERT(V1.[VALUE], FLOAT) > 11000 AND
V1.DOCID = A1.DOCID AND
V2.DOCID = A2.DOCID AND
A1.ANCESTORLEVEL = 1 AND
A2.ANCESTORLEVEL = 1 AND
A1.DOCID = A2.DOCID AND
A1.ANCESTORORDER = A2.ANCESTORORDER AND
V3.DOCID = A3.DOCID AND
A3.ANCESTORLEVEL = 1 AND
A1.DOCID = A3.DOCID AND
A1.ANCESTORORDER = A3.ANCESTORORDER

Q6: SELECT
V2.*
FROM
PATHVALUE V1,
PATHVALUE V2,
PATH P1,
PATH P2,
ANCESTORINFO A1,
ANCESTORINFO A2
WHERE
P1.PATH = '.ORDER.@ID' AND
P1.[ID]=V1.PATHID AND
V1.[VALUE]='2' AND
P2.PATH = '.ORDER%' AND
P2.[ID]=V2.PATHID AND
V1.DOCID = A1.DOCID AND

```

APPENDIX A

```
V2.DOCID = A2.DOCID AND
A1.ANCESTORLEVEL = 1 AND
A2.ANCESTORLEVEL = 1 AND
A1.DOCID = A2.DOCID AND
A1.ANCESTORORDER = A2.ANCESTORORDER
```

```
Q7: SELECT
      V2.*
FROM
      PATHVALUE V1,
      PATH P1,
      PATHVALUE V2,
      PATH P2,
      ANCESTORINFO A1,
      ANCESTORINFO A2
WHERE
      V1.PATHID = P1.[ID] AND
      P1.PATH = '.CATALOG.ITEM.@ID' AND
      V1.[VALUE]='I6' AND
      P2.PATH LIKE '.CATALOG.ITEM%' AND
      V2.PATHID = P1.ID AND
      V1.DOCID = A1.DOCID AND
      V2.DOCID = A2.DOCID AND
      A1.ANCESTORLEVEL = 2 AND
      A2.ANCESTORLEVEL = 2 AND
      A1.DOCID = A2.DOCID AND
      A1.ANCESTORORDER = A2.ANCESTORORDER
```

```
Q8: SELECT
      V4.*
FROM
      PATHVALUE V3,
      PATH P3,
      PATHVALUE V4,
      ANCESTORINFO A3,
      ANCESTORINFO A4
WHERE
      V3.PATHID = P3.[ID] AND
      P3.PATH = '.CATALOG.ITEM.@ID' AND
      V4.PATH LIKE '.CATALOG.ITEM%' AND
      V4.PATHID = P4.PATHID AND
      V3.DOCID = A3.DOCID AND
      V4.DOCID = A4.DOCID AND
      A3.ANCESTORLEVEL = 1 AND
      A4.ANCESTORLEVEL = 1 AND
      A3.DOCID = A4.DOCID AND
      A3.ANCESTORORDER = A4.ANCESTORORDER
      V3.[VALUE] NOT IN
      (
      SELECT
            V1.[VALUE]
      FROM
            PATHVALUE V1,
            PATH P1,
            PATH P2,
            PATHVALUE V2,
            ANCESTORINFO A1,
            ANCESTORINFO A2
      WHERE
            V1.PATHID = P1.[ID] AND
            P1.PATH = '.CATALOG.ITEM.@ID' AND
            V2.PATHID = P2.[ID] AND
            P2.PATH = '.CATALOG.ITEM.AUTHORS.AUTHOR
            .CONTACT_INFORMATION.MAILING_ADDRESS.NAME_OF_COUNTRY' AND
            V2.[VALUE] <> 'CANADA' AND
            V1.DOCID = A1.DOCID AND
            V2.DOCID = A2.DOCID AND
            A1.ANCESTORLEVEL = 2 AND
```

APPENDIX A

```

        A2.ANCESTORLEVEL = 2 AND
        A1.DOCID = A2.DOCID AND
        A1.ANCESTORORDER = A2.ANCESTORORDER
    )

Q9: SELECT
    V2.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    ANCESTORINFO A1,
    ANCESTORINFO A2
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH = '.CATALOG.ITEM.@ID' AND
    V1.[VALUE]='I6' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH LIKE '.CATALOG.ITEM.AUTHORS.AUTHOR
.CONTACT_INFORMATION.MAILING_ADDRESS%' AND
    V1.DOCID = A1.DOCID AND
    V2.DOCID = A2.DOCID AND
    A1.ANCESTORLEVEL = 2 AND
    A2.ANCESTORLEVEL = 2 AND
    A1.DOCID = A2.DOCID AND
    A1.ANCESTORORDER = A2.ANCESTORORDER

Q10:SELECT
    VO.[VALUE]
FROM
    PATHVALUE VO,
    PATH PO,
    PATHVALUE V4,
    PATH P4,
    ANCESTORINFO AO,
    ANCESTORINFO A4
WHERE
    VO.PATHID = PO.[ID] AND
    PO.PATH = '.CATALOG.ITEM.PUBLISHER.NAME' AND
    V4.PATHID = P4.[ID] AND
    P4.PATH = '.CATALOG.ITEM.DATE_OF_RELEASE' AND
    V4.[VALUE] > '1990-01-01' AND
    V4.[VALUE] < '1991-01-01' AND
    VO.DOCID = AO.DOCID AND
    V4.DOCID = A4.DOCID AND
    AO.ANCESTORLEVEL = 1 AND
    A4.ANCESTORLEVEL = 1 AND
    AO.DOCID = A4.DOCID AND
    AO.ANCESTORORDER = A4.ANCESTORORDER AND
    VO.[VALUE] NOT IN(
        SELECT
            V3.[VALUE]
        FROM
            PATHVALUE V2, PATH P2, PATHVALUE V3, PATH P3
            , ANCESTORINFO A2, ANCESTORINFO A3
        WHERE
            V2.PATHID = P2.[ID] AND
            P2.PATH = '.CATALOG.ITEM.PUBLISHER
.CONTACT_INFORMATION.FAX_NUMBER' AND
            V3.DOCID = A3.DOCID AND
            V2.DOCID = A2.DOCID AND
            A3.ANCESTORLEVEL = 2 AND
            A2.ANCESTORLEVEL =2 AND
            A3.DOCID = A2.DOCID AND
            A3.ANCESTORORDER = A2.ANCESTORORDER )

```

APPENDIX A

```

Q11:SELECT
      V3.[VALUE]
FROM
      PATHVALUE V0,
      PATH P0,
      PATHVALUE V1,
      PATH P1,
      PATHVALUE V2,
      PATH P2,
      PATHVALUE V3,
      PATH P3,
      ANCESTORINFO A0,
      ANCESTORINFO A1,
      ANCESTORINFO A2,
      ANCESTORINFO A3
WHERE
      V0.PATHID = P0.[ID] AND
      P0.PATH = '.CATALOG.ITEM.ATTRIBUTES
      .SIZE_OF_BOOK.LENGTH' AND
      V1.PATHID = P1.[ID] AND
      P1.PATH = '.CATALOG.ITEM.ATTRIBUTES
      .SIZE_OF_BOOK.WIDTH' AND
      V2.PATHID = P2.[ID] AND
      P2.PATH = '.CATALOG.ITEM.ATTRIBUTES
      .SIZE_OF_BOOK.HEIGHT' AND
      V3.PATHID = P3.[ID] AND
      P3.PATH = '.CATALOG.ITEM.TITLE' AND
      CAST(V0.[VALUE] AS FLOAT)*CAST(V1.[VALUE]
      AS FLOAT)*CAST(V2.[VALUE] AS FLOAT) > 50000 AND
      V1.DOCID = A1.DOCID AND
      V2.DOCID = A2.DOCID AND
      A1.ANCESTORLEVEL = 2 AND
      A2.ANCESTORLEVEL = 2 AND
      A1.DOCID = A2.DOCID AND
      A1.ANCESTORORDER = A2.ANCESTORORDER AND
      V1.DOCID = A1.DOCID AND
      V0.DOCID = A0.DOCID AND
      A1.ANCESTORLEVEL = 2 AND
      A0.ANCESTORLEVEL = 2 AND
      A1.DOCID = A0.DOCID AND
      A1.ANCESTORORDER = A0.ANCESTORORDER AND
      V1.DOCID = A1.DOCID AND
      V3.DOCID = A3.DOCID AND
      A3.ANCESTORLEVEL = 2 AND
      A3.ANCESTORLEVEL = 2 AND
      A1.DOCID = A3.DOCID AND
      A1.ANCESTORORDER = A3.ANCESTORORDER

```

```

Q12:SELECT
      V2.*
FROM
      PATHVALUE V1,
      PATH P1,
      PATHVALUE V2,
      PATH P2,
      ANCESTORINFO A1,
      ANCESTORINFO A2
WHERE
      V1.PATHID = P1.[ID] AND
      P1.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
      V1.[VALUE]='PADMA SILLINCE' AND
      V2.PATHID = P2.[ID] AND
      P2.PATH = '.ARTICLE.PROLOG.TITLE' AND
      V1.DOCID = A1.DOCID AND
      V2.DOCID = A2.DOCID AND
      A1.ANCESTORLEVEL = 2 AND
      A2.ANCESTORLEVEL = 2 AND

```

APPENDIX A

```
A1.DOCID = A2.DOCID AND
A1.ANCESTORORDER = A2.ANCESTORORDER
```

```
Q13:SELECT
  V2.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2,
  ANCESTORINFO A1,
  ANCESTORINFO A2
WHERE
  V1.PATHID = P1.[ID] AND
  P1.PATH = '.ARTICLE.@ID' AND
  V1.[VALUE]='8' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.ARTICLE.BODY.SECTION.@HEADING' AND
  v2.[VALUE] = 'INTRODUCTION' AND
  V1.DOCID = A1.DOCID AND
  V2.DOCID = A2.DOCID AND
  A1.ANCESTORLEVEL = 1 AND
  A2.ANCESTORLEVEL = 1 AND
  A1.DOCID = A2.DOCID AND
  A1.ANCESTORORDER = A2.ANCESTORORDER
```

```
Q14:SELECT
  V2.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2,
  ANCESTORINFO A1,
  ANCESTORINFO A2
WHERE
  V1.PATHID = P1.[ID] AND
  P1.PATH = '.ARTICLE.BODY.ABSTRACT.P' AND
  V1.[VALUE] LIKE '%THE%HOCKEY%' OR
  V1.[VALUE] LIKE '%HOCKEY%THE%' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.ARTICLE.PROLOG.TITLE' AND
  V1.DOCID = A1.DOCID AND
  V2.DOCID = A2.DOCID AND
  A1.ANCESTORLEVEL = 1 AND
  A2.ANCESTORLEVEL = 1 AND
  A1.DOCID = A2.DOCID AND
  A1.ANCESTORORDER = A2.ANCESTORORDER
```

```
Q15:SELECT
  V2.*,
  V3.*,
  V4.*,
  V5.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2,
  PATHVALUE V3,
  PATH P3,
  PATHVALUE V4,
  PATH P4,
  PATHVALUE V5,
  PATH P5,
  ANCESTORINFO A1,
```

APPENDIX A

```

ANCESTORINFO A2,
ANCESTORINFO A3,
ANCESTORINFO A4,
ANCESTORINFO A5
WHERE
V1.PATHID = P1.[ID] AND
P1.PATH = '.ARTICLE.@ID' AND
V1.[VALUE]='5' AND
V2.PATHID = P2.[ID] AND
P2.PATH = '.ARTICLE.PROLOG.TITLE' AND
V3.PATHID = P3.[ID] AND
P3.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
V4.PATHID = P4.[ID] AND
P4.PATH = '.ARTICLE.PROLOG.DATELINE.DATE' AND
V5.PATHID = P5.[ID] AND
P5.PATH = '.ARTICLE.BODY.ABSTRACT' AND
V1.DOCID = A1.DOCID AND
A1.ANCESTORLEVEL = 1 AND
V2.DOCID = A2.DOCID AND
A2.ANCESTORLEVEL = 1 AND
V3.DOCID = A3.DOCID AND
A3.ANCESTORLEVEL = 1 AND
V4.DOCID = A4.DOCID AND
A4.ANCESTORLEVEL = 1 AND
V5.DOCID = A5.DOCID AND
A5.ANCESTORLEVEL = 1 AND
A1.DOCID = A2.DOCID AND
A1.ANCESTORORDER = A2.ANCESTORORDER AND
A1.DOCID = A3.DOCID AND
A1.ANCESTORORDER = A3.ANCESTORORDER AND
A1.DOCID = A4.DOCID AND
A1.ANCESTORORDER = A4.ANCESTORORDER AND
A1.DOCID = A5.DOCID AND
A1.ANCESTORORDER = A5.ANCESTORORDER

Q16:SELECT
[VALUE]
FROM
PATHVALUE VO,
PATH PO
WHERE
PO.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
VO.PATHID = PO.[ID] AND
VO.[VALUE] NOT IN
(
SELECT
DISTINCT(V2.[VALUE])
FROM
PATHVALUE V1,
PATH P1,
PATHVALUE V2,
PATH P2,
ANCESTORINFO A1,
ANCESTORINFO A2
WHERE
V1.PATHID = P1.[ID] AND
P1.PATH LIKE '.ARTICLE.PROLOG.AUTHORS.AUTHOR.CONTACT.%' AND
V2.PATHID = P2.[ID] AND
P2.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
V1.DOCID = A1.DOCID AND
V2.DOCID = A2.DOCID AND
A1.ANCESTORLEVEL = 1 AND
A2.ANCESTORLEVEL = 1 AND
A1.DOCID = A2.DOCID AND

```

APPENDIX A

```

V2.*,
V3.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2,
  PATHVALUE V3,
  PATH P3,
  ANCESTORINFO A1,
  ANCESTORINFO A2,
  ANCESTORINFO A3
WHERE
  V1.PATHID = P1.[ID] AND
  P1.PATH LIKE '.ARTICLE%.P' AND
  V1.[VALUE] LIKE '%HOCKEY' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.ARTICLE.PROLOG.TITLE' AND
  V1.DOCID = A1.DOCID AND
  V2.DOCID = A2.DOCID AND
  A1.ANCESTORLEVEL = 1 AND
  A2.ANCESTORLEVEL = 1 AND
  A1.DOCID = A2.DOCID AND
  A1.ANCESTORORDER = A2.ANCESTORORDER AND
  A3.ANCESTORLEVEL = 1 AND
  A1.DOCID = A3.DOCID AND
  A1.ANCESTORORDER = A3.ANCESTORORDER

Q18:SELECT
  V3.*,
  V1.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2,
  PATHVALUE V3,
  PATH P3,
  ANCESTORINFO A1,
  ANCESTORINFO A2,
  ANCESTORINFO A3
WHERE
  V1.PATHID = P1.[ID] AND
  P1.PATH = '.DICTIONARY.E.SS.S.QP.Q.QD' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.DICTIONARY.E.HWG.HW' AND
  V2.[VALUE]='DEALING' AND
  A1.ANCESTORLEVEL = 3 AND
  A2.ANCESTORLEVEL = 3 AND
  A1.DOCID = A2.DOCID AND
  A1.ANCESTORORDER = A2.ANCESTORORDER AND
  A3.ANCESTORLEVEL = 3 AND
  A1.DOCID = A3.DOCID AND
  A1.ANCESTORORDER = A3.ANCESTORORDER

Q19:SELECT
  V2.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2,
  ANCESTORINFO A1,
  ANCESTORINFO A2
WHERE
  V1.PATHID = P1.[ID] AND
  P1.PATH LIKE '.DICTIONARY.E.%' AND

```


APPENDIX A

```
V2.PATHID = P2.[ID] AND
P2.PATH = '.DICTIONARY.E.HWG.HW' AND
V1.[VALUE] LIKE '%HOCKEY%' AND
V1.DOCID = A1.DOCID AND
V2.DOCID = A2.DOCID AND
A1.ANCESTORLEVEL = 2 AND
A2.ANCESTORLEVEL = 2 AND
A1.DOCID = A2.DOCID AND
A1.ANCESTORORDER = A2.ANCESTORORDER
```

```
Q20:SELECT
    V1.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    ANCESTORINFO A1,
    ANCESTORINFO A2
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH = '.DICTIONARY.E.QID' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH = '.DICTIONARY.E.HWG.HW' AND
    V2.[VALUE]='DEALING' AND
    V1.DOCID = A1.DOCID AND
    V2.DOCID = A2.DOCID AND
    A1.ANCESTORLEVEL = 2 AND
    A2.ANCESTORLEVEL = 2 AND
    A1.DOCID = A2.DOCID AND
    A1.ANCESTORORDER = A2.ANCESTORORDER
```

```
Q21:SELECT
    V1.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    ANCESTORINFO A1,
    ANCESTORINFO A2
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH = '.DICTIONARY.E.SS.S.QP.Q.QD' AND
    V1.[VALUE]='1900' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH = '.DICTIONARY.E.HWG.HW' AND
    V1.DOCID = A1.DOCID AND
    V2.DOCID = A2.DOCID AND
    A1.ANCESTORLEVEL = 2 AND
    A2.ANCESTORLEVEL = 2 AND
    A1.DOCID = A2.DOCID AND
    A1.ANCESTORORDER = A2.ANCESTORORDER
```

```
Q22:SELECT
    V1.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    ANCESTORINFO A1,
    ANCESTORINFO A2
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH LIKE '.DICTIONARY.E.%HW' AND
```

APPENDIX A

```

V1.[VALUE]='AND' AND
V2.PATHID = P2.[ID] AND
P2.PATH = '.DICTIONARY.E.SS.S.QP.%QT' AND
V1.DOCID = A1.DOCID AND
V2.DOCID = A2.DOCID AND
A1.ANCESTORLEVEL = 2 AND
A2.ANCESTORLEVEL = 2 AND
A1.DOCID = A2.DOCID AND
A1.ANCESTORORDER = A2.ANCESTORORDER

```

A.2 SQL Queries for XBench in SUCXENT++

```

Q1: SELECT
      V2.*
FROM
      PATHVALUE V1,
      PATHVALUE V2,
      PATH P1,
      PATH P2
WHERE
      P1.PATH = '.ORDER.@ID' AND
      P1.[ID]=V1.PATHID AND
      V1.[VALUE]='1' AND
      P2.PATH = '.ORDER.CUSTOMER_ID' AND
      P2.[ID]=V2.PATHID AND
      V1.DOCID = V2.DOCID

Q2: SELECT
      V1.*,
      V4.*,
      V5.*,
      V6.*
FROM
      PATHVALUE V1,
      PATH P1,
      PATHVALUE V2,
      PATH P2,
      PATHVALUE V3,
      PATH P3,
      PATHVALUE V4,
      PATH P4,
      PATHVALUE V5,
      PATH P5,
      PATHVALUE V6,
      PATH P6,
      DOCUMENTRVALUE D2
WHERE
      P1.PATH = '.ORDER.@ID' AND
      V1.PATHID = P1.[ID] AND
      V1.[VALUE]='6' AND
      P2.PATH = '.ORDER.CUSTOMER_ID' AND
      V2.PATHID = P2.[ID] AND
      P3.PATH = '.CUSTOMERS.CUSTOMER.@ID' AND
      V3.PATHID = P3.[ID] AND
      V3.[VALUE]=V2.[VALUE] AND
      P4.PATH = '.CUSTOMERS.CUSTOMER.FIRST_NAME' AND
      V4.PATHID = P4.[ID] AND
      P5.PATH = '.ORDER.ORDER_STATUS' AND
      V5.PATHID = P5.ID AND
      P6.PATH = '.CUSTOMERS.CUSTOMER.LAST_NAME' AND
      P6.ID = V5.PATHID AND
      V1.DOCID = V2.DOCID AND
      V1.DOCID = V5.DOCID      V3.DOCID = V4.DOCID AND
      V3.DOCID = V6.DOCID AND
      D2.DOCID = V3.DOCID AND
      D2.LEVEL = 1 AND

```

APPENDIX A

```
ABS(V3.BRANCHORDERSUM-V4.BRANCHORDERSUM) < D2.RVALUE AND
ABS(V3.BRANCHORDERSUM-V6.BRANCHORDERSUM) < D2.RVALUE
```

```
Q3: SELECT
      V2.*
FROM
      PATHVALUE V1,
      PATHVALUE V2,
      PATH P1,
      PATH P2
WHERE
      P1.PATH = '.ORDER.ØID' AND
      P1.[ID]=V1.PATHID AND
      V1.[VALUE]='2' AND
      P2.PATH LIKE '.ORDER.ORDER_LINES.ORDER_LINE%' AND
      P2.[ID]=V2.PATHID AND
      V1.DOCID = V2.DOCID

Q4: SELECT
      V2.*
FROM
      PATHVALUE V1,
      PATHVALUE V2,
      PATH P1,
      PATH P2
WHERE
      P1.PATH = '.ORDER.ØID' AND
      P1.[ID]=V1.PATHID AND
      V1.[VALUE]='2' AND
      P2.PATH = '.ORDER.ORDER_LINES.ORDER_LINE.ITEM_ID' AND
      P2.[ID]=V2.PATHID AND
      V1.DOCID = V2.DOCID

Q5: SELECT
      V3.*,
      V4.*,
      V2.*
FROM
      PATHVALUE V1,
      PATHVALUE V2,
      PATH P1,
      PATH P2,
      PATHVALUE V3,
      PATH P3,
      PATHVALUE V4,
      PATH P4
WHERE
      V1.PATHID = P1.PATHID AND
      P1.PATH = '.ORDER.TOTAL AND
      V2.PATHID = P2.PATHID AND
      P2.PATH = '.ORDER.SHIP_TYPE' AND
      P3.PATH = '.ORDER.ØID' AND
      P4.PATH = '.ORDER.ORDER_DATE'
      P3.PATHID = V3.PATHID AND
      P4.PATHID = V4.PATHID
      CONVERT(V1.[VALUE], FLOAT) > 11000 AND
      V1.DOCID = V2.DOCID AND
      V1.DOCID = V3.DOCID AND
      V1.DOCID = V4.DOCID

Q6: SELECT
      V2.*
FROM
      PATHVALUE V1,
      PATHVALUE V2,
```

APPENDIX A

```

        PATH P1,
        PATH P2
    WHERE
        V1.PATHID = P1.PATHID AND
        P1.PATH = '.ORDER.ORD' AND
        V1.[VALUE]='6' AND
        V2.PATHID = P2.PATHID AND
        P2.PATH LIKE '.ORDER.TOTAL' AND
        V1.DOCID = V2.DOCID

Q7: SELECT
    V2.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    DOCUMENTVALUE D
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH = '.CATALOG.ITEM.ORD' AND
    V1.[VALUE]='16' AND
    P2.PATH LIKE '.CATALOG.ITEM%' AND
    V2.PATHID = P1.ID AND
    V2.DOCID = V1.DOCID AND
    D.DOCID = V1.DOCID AND
    D.LEVEL = 1 AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q8: SELECT
    V4.*
FROM
    PATHVALUE V3,
    PATH P3,
    PATHVALUE V4,
    DOCUMENTVALUE D
WHERE
    V3.PATHID = P3.[ID] AND
    P3.PATH = '.CATALOG.ITEM.ORD' AND
    V4.PATH LIKE '.CATALOG.ITEM%' AND
    V4.PATHID = P4.PATHID AND
    V4.DOCID = V3.DOCID AND
    D.DOCID = V1.DOCID AND
    D.LEVEL = 1 AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE V3.[VALUE] NOT IN
    (
        SELECT
            V1.[VALUE]
        FROM
            PATHVALUE V1,
            PATH P1,
            PATH P2,
            PATHVALUE V2,
            DOCUMENTVALUE D2
        WHERE
            V1.PATHID = P1.[ID] AND
            P1.PATH = '.CATALOG.ITEM.ORD' AND
            V2.PATHID = P2.[ID] AND
            P2.PATH = '.CATALOG.ITEM.AUTHORS.AUTHOR
            .CONTACT_INFORMATION.MAILING_ADDRESS.NAME_OF_COUNTRY' AND
            V2.[VALUE] <> 'CANADA' AND
            V2.DOCID = V1.DOCID AND
            D1.DOCID = V1.DOCID AND
            D1.LEVEL = 1 AND
            ABS(V2.BRANCHORDERSUM-V1.BRANCHORDERSUM) < D1.RVALUE
    )

```

APPENDIX A

```
Q9: SELECT
      V2.*
    FROM
      PATHVALUE V1,
      PATH P1,
      PATHVALUE V2,
      PATH P2,
      DOCUMENTVALUE D
    WHERE
      V1.PATHID = P1.[ID] AND
      P1.PATH = '.CATALOG.ITEM.QID' AND
      V1.[VALUE]='I6' AND
      V2.PATHID = P2.[ID] AND
      P2.PATH LIKE '.CATALOG.ITEM.AUTHORS.AUTHOR
      .CONTACT_INFORMATION.MAILING_ADDRESS%' AND
      V2.DOCID = V1.DOCID AND
      D.DOCID = V1.DOCID AND
      D.LEVEL = 1 AND
      ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```

```
Q10:SELECT
      VO.[VALUE]
    FROM
      PATHVALUE VO,
      PATH PO,
      PATHVALUE V4,
      PATH P4,
      DOCUMENTVALUE D
    WHERE
      VO.PATHID = PO.[ID] AND
      PO.PATH = '.CATALOG.ITEM.PUBLISHER.NAME' AND
      V4.PATHID = P4.[ID] AND
      P4.PATH = '.CATALOG.ITEM.DATE_OF_RELEASE' AND
      V4.[VALUE] > '1990-01-01' AND
      V4.[VALUE] < '1991-01-01' AND
      VO.DOCID = V4.DOCID AND
      D.DOCID = VO.DOCID AND
      D.LEVEL = 1 AND
      ABS(VO.BRANCHORDERSUM-V4.BRANCHORDERSUM) < D.RVALUE AND
      VO.[VALUE] NOT IN(
        SELECT
          V3.[VALUE]
        FROM
          PATHVALUE V2, PATH P2, PATHVALUE V3, PATH P3
          , DOCUMENTVALUE D1
        WHERE
          V2.PATHID = P2.[ID] AND
          P2.PATH = '.CATALOG.ITEM.PUBLISHER
          .CONTACT_INFORMATION.FAX_NUMBER' AND
          V3.PATHID = P3.[ID] AND
          P3.PATH = '.CATALOG.ITEM.PUBLISHER.NAME' AND
          V2.DOCID = V3.DOCID AND
          D1.DOCID = V2.DOCID AND
          D.LEVEL = 1 AND
          ABS(V3.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE )
```

```
Q11:SELECT
      V3.[VALUE]
    FROM
      PATHVALUE VO,
      PATH PO,
      PATHVALUE V1,
      PATH P1,
      PATHVALUE V2,
      PATH P2,
      PATHVALUE V3,
      PATH P3,
```

APPENDIX A

```

DOCUMENTRAVALUE D
WHERE
  V0.PATHID = P0.[ID] AND
  P0.PATH = '.CATALOG.ITEM.ATTRIBUTES
.SIZE_OF_BOOK.LENGTH' AND
  V1.PATHID = P1.[ID] AND
  P1.PATH = '.CATALOG.ITEM.ATTRIBUTES
.SIZE_OF_BOOK.WIDTH' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.CATALOG.ITEM.ATTRIBUTES
.SIZE_OF_BOOK.HEIGHT' AND
  V3.PATHID = P3.[ID] AND
  P3.PATH = '.CATALOG.ITEM.TITLE' AND
  CAST(V0.[VALUE] AS FLOAT)*CAST(V1.[VALUE] AS FLOAT)
*CAST(V2.[VALUE] AS FLOAT) > 50000 AND
  D.DOCID = V0.DOCID AND
  D.LEVEL = 1 AND
  V0.DOCID = V1.DOCID AND
  ABS(V0.BRANCHORDERSUM-V1.BRANCHORDERSUM) < D.RVALUE AND
  V0.DOCID = V2.DOCID AND
  ABS(V0.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE AND
  V0.DOCID = V3.DOCID AND
  ABS(V0.BRANCHORDERSUM-V3.BRANCHORDERSUM) < D.RVALUE

```

```

Q12:SELECT
  V2.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2,
  DOCUMENTRVALUE D
WHERE
  V1.PATHID = P1.[ID] AND
  P1.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
  V1.[VALUE]='PADMA SILLINCE' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.ARTICLE.PROLOG.TITLE' AND
  V1.DOCID = V2.DOCID AND
  V1.DOCID = D.DOCID AND
  D.LEVEL = 1 AND
  ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

```

```

Q13:SELECT
  V2.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2
WHERE
  V1.PATHID = P1.[ID] AND
  P1.PATH = '.ARTICLE.@ID' AND
  V1.[VALUE]='S' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.ARTICLE.BODY.SECTION.@HEADING' AND
  V2.[VALUE] = 'INTRODUCTION' AND
  V1.DOCID = V2.DOCID

```

```

Q14:SELECT
  V2.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2

```

APPENDIX A

```

WHERE
  V1.PATHID = P1.[ID] AND
  P1.PATH = '.ARTICLE.BODY.ABSTRACT.P' AND
  V1.[VALUE] LIKE '%THE%HOCKEY%' OR
  V1.[VALUE] LIKE '%HOCKEY%THE%' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.ARTICLE.PROLOG.TITLE' AND
  V1.DOCID = V2.DOCID
\end{verbatim}

\begin{verbatim}
Q15:SELECT
  V2.*,
  V3.*,
  V4.*,
  V5.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2,
  PATHVALUE V3,
  PATH P3,
  PATHVALUE V4,
  PATH P4,
  PATHVALUE V5,
  PATH P5
WHERE
  V1.PATHID = P1.[ID] AND
  P1.PATH = '.ARTICLE.@ID' AND
  V1.[VALUE]='5' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.ARTICLE.PROLOG.TITLE' AND
  V3.PATHID = P3.[ID] AND
  P3.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
  V4.PATHID = P4.[ID] AND
  P4.PATH = '.ARTICLE.PROLOG.DATELINE.DATE' AND
  V5.PATHID = P5.[ID] AND
  P5.PATH = '.ARTICLE.BODY.ABSTRACT'

Q16:SELECT
  [VALUE]
FROM
  PATHVALUE V0,
  PATH P0
WHERE
  P0.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
  V0.PATHID = P0.[ID] AND
  V0.[VALUE] NOT IN
  (
    SELECT
      DISTINCT(V2.[VALUE])
    FROM
      PATHVALUE V1,
      PATH P1,
      PATHVALUE V2,
      PATH P2,
      DOCUMENTVALUE R1
    WHERE
      V1.PATHID = P1.[ID] AND
      P1.PATH LIKE '.ARTICLE.PROLOG.AUTHORS.AUTHOR.CONTACT.%" AND
      V2.PATHID = P2.[ID] AND
      P2.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
      V1.DOCID = V2.DOCID AND
      V1.DOCID = R1.DOCID AND
      R1.[LEVEL]=3 AND
      ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < R1.RVALUE
  )

```


APPENDIX A

)

```

Q17:SELECT
    V2.*,
    V3.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    PATHVALUE V3,
    PATH P3
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH LIKE '.ARTICLE%.P' AND
    V1.[VALUE] LIKE '%HOCKEY' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH = '.ARTICLE.PROLOG.TITLE' AND
    V3.PATHID = P3.[ID] AND
    P3.PATH = '.ARTICLE.PROLOG.ABSTRACT' AND
    V1.DOCID = V2.DOCID AND
    V1.DOCID = V3.DOCID

Q18:SELECT
    V1.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    DOCUMENTRVALUE D
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH = '.DICTIONARY.E.QID' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH = '.DICTIONARY.E.HWG.HW' AND
    V2.[VALUE]='the' AND
    V1.DOCID = V2.DOCID AND
    V1.DOCID = D.DOCID AND
    D.LEVEL = 1 AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q19:SELECT
    V1.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    DOCUMENTRVALUE D
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH LIKE '.DICTIONARY.E.%HW' AND
    V1.[VALUE]='AND' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH LIKE '.DICTIONARY.E.SS.S.QP.%QT' AND
    V1.DOCID = V2.DOCID AND
    V1.DOCID = D.DOCID AND
    D.LEVEL = 1 AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q20:SELECT
    V2.*
FROM
    PATHVALUE V1,
    PATH P1,

```

APPENDIX A

```

        PATHVALUE V2,
        PATH P2,
        DOCUMENTRVALUE R
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH LIKE '.DICTIONARY.E.HWG.HW' AND
    V1.[VALUE]='THE' AND
    V1.DOCID = R.DOCID AND
    R.[LEVEL]=3 AND
    V1.DOCID = V2.DOCID AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < R.RVALUE AND
    V3.PATHID = P3.[ID] AND
    P3.PATH = '.DICTIONARY.E.SS.S.QP.Q.A' AND
    V1.DOCID = V3.DOCID AND
    ABS(V1.BRANCHORDERSUM-V3.BRANCHORDERSUM) < R.RVALUE
ORDER BY
    V1.[VALUE]

Q21:SELECT
    V2.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    DOCUMENTRVALUE D
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH LIKE '.DICTIONARY.E.%' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH = '.DICTIONARY.E.HWG.HW' AND
    V1.[VALUE] LIKE '%HOCKEY%' AND
    V1.DOCID = V2.DOCID AND
    V1.DOCID = D.DOCID AND
    D.LEVEL = 2 AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q22:SELECT
    V1.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    DOCUMENTRVALUE D
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH = '.DICTIONARY.E.SS.S.QP.Q.QD' AND
    V1.[VALUE]='1900' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH = '.DICTIONARY.E.HWG.HW' AND
    V1.DOCID = V2.DOCID AND
    V1.DOCID = D.DOCID AND
    D.LEVEL = 2 AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

```

A.3 SQL Queries for X Bench after Partitioning in SUCXENT++

```

Q1: SELECT
    V2.*
FROM
    PV1 V1,
    PV2 V2,

```

APPENDIX A

```

        PATH P1,
        PATH P2
    WHERE
        P1.PATH = 'ORDER.®ID' AND
        P1.[ID]=V1.PATHID AND
        V1.[VALUE]='1' AND
        P2.PATH = 'ORDER.CUSTOMER_ID' AND
        P2.[ID]=V2.PATHID AND
        V1.DOCID = V2.DOCID

Q2: SELECT
    V1.*,
    V4.*,
    V5.*,
    V6.*
FROM
    PV1 V1,
    PATH P1,
    PV2 V2,
    PATH P2,
    PV81 V3,
    PATH P3,
    PV8 V4,
    PATH P4,
    PV71 V5,
    PATH P5,
    PV9 V6,
    PATH P6,
    DOCUMENTRVALUE D2
WHERE
    P1.PATH = 'ORDER.®ID' AND
    V1.PATHID = P1.[ID] AND
    V1.[VALUE]='6' AND
    P2.PATH = 'ORDER.CUSTOMER_ID' AND
    V2.PATHID = P2.[ID] AND
    P3.PATH = 'CUSTOMERS.CUSTOMER.®ID' AND
    V3.PATHID = P3.[ID] AND
    V3.[VALUE]=V2.[VALUE] AND
    P4.PATH = 'CUSTOMERS.CUSTOMER.FIRST_NAME' AND
    V4.PATHID = P4.[ID] AND
    P5.PATH = 'ORDER.ORDER_STATUS' AND
    V5.PATHID = P5.ID AND
    P6.PATH = 'CUSTOMERS.CUSTOMER.LAST_NAME' AND
    P6.ID = V5.PATHID AND
    V1.DOCID = V2.DOCID AND
    V1.DOCID = V5.DOCID      V3.DOCID = V4.DOCID AND
    V3.DOCID = V6.DOCID AND
    D2.DOCID = V3.DOCID AND
    D2.LEVEL = 1 AND
    ABS(V3.BRANCHORDERSUM-V4.BRANCHORDERSUM) < D2.RVALUE AND
    ABS(V3.BRANCHORDERSUM-V6.BRANCHORDERSUM) < D2.RVALUE

Q3: SELECT
    V2.*
FROM
    PV1 V1,
    PV3 V2,
    PATH P1,
    PATH P2
WHERE
    P1.PATH = 'ORDER.®ID' AND
    P1.[ID]=V1.PATHID AND
    V1.[VALUE]='2' AND
    P2.PATH LIKE 'ORDER.ORDER_LINES.ORDER_LINE%' AND
    P2.[ID]=V2.PATHID AND
    V1.DOCID = V2.DOCID

```

APPENDIX A

```
Q4: SELECT
    V2.*
FROM
    PV1 V1,
    PV3 V2,
    PATH P1,
    PATH P2
WHERE
    P1.PATH = '.ORDER.@ID' AND
    P1.[ID]=V1.PATHID AND
    V1.[VALUE]='2' AND
    P2.PATH = '.ORDER.ORDER_LINES.ORDER_LINE.ITEM_ID' AND
    P2.[ID]=V2.PATHID AND
    V1.DOCID = V2.DOCID
```

```
Q5: SELECT
    V3.*,
    V4.*,
    V5.*
FROM
    PV5 V1,
    PV7 V2,
    PATH P1,
    PATH P2,
    PV1 V3,
    PATH P3,
    PV6 V4,
    PATH P4,
WHERE
    V1.PATHID = P1.PATHID AND
    P1.PATH = '.ORDER.TOTAL' AND
    V2.PATHID = P2.PATHID AND
    P2.PATH = '.ORDER.SHIP_TYPE' AND
    P3.PATH = '.ORDER.@ID' AND
    P4.PATH = '.ORDER.ORDER_DATE'
    P3.PATHID = V3.PATHID AND
    P4.PATHID = V4.PATHID
    CONVERT(V1.[VALUE], FLOAT) > 11000 AND
    V1.DOCID = V2.DOCID AND
    V1.DOCID = V3.DOCID AND
    V4.DOCID = V3.DOCID
```

```
Q6: SELECT
    V2.*
FROM
    PV1 V1,
    PV5 V2,
    PATH P1,
    PATH P2
WHERE
    V1.PATHID = P1.PATHID AND
    P1.PATH = '.ORDER.@ID' AND
    V1.[VALUE]='6' AND
    V2.PATHID = P2.PATHID AND
    P2.PATH LIKE '.ORDER.TOTAL' AND
    V1.DOCID = V2.DOCID
```

```
Q7: SELECT
    V2.*
FROM
    PV11 V1,
    PATH P1,
    PV11 V2,
    PATH P2,
    DOCUMENTVALUE D
WHERE
```

APPENDIX A

```
V1.PATHID = P1.[ID] AND
P1.PATH = '.CATALOG.ITEM.@ID' AND
V1.[VALUE]='I6' AND
P2.PATH LIKE '.CATALOG.ITEM%' AND
V2.PATHID = P1.ID AND
V2.DOCID = V1.DOCID AND
D.DOCID = V1.DOCID AND
D.LEVEL = 1 AND
ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```

```
Q8: SELECT
    V4.*
FROM
    PV11 V3,
    PATH P3,
    PV111 V4,
    DOCUMENTRVALUE D
WHERE
    V3.PATHID = P3.[ID] AND
    P3.PATH = '.CATALOG.ITEM.@ID' AND
    V4.PATH LIKE 'CATALOG.ITEM%' AND
    V4.PATHID = P4.PATHID AND
    V4.DOCID = V3.DOCID AND
    D.DOCID = V1.DOCID AND
    D.LEVEL = 1 AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE V3.[VALUE] NOT IN
(
    SELECT
        V1.[VALUE]
    FROM
        PV11 V1,
        PATH P1,
        PATH P2,
        PV12 V2,
        DOCUMENTRAVLUE D2
    WHERE
        V1.PATHID = P1.[ID] AND
        P1.PATH = '.CATALOG.ITEM.@ID' AND
        V2.PATHID = P2.[ID] AND
        P2.PATH = '.CATALOG.ITEM.AUTHORS.AUTHOR
        .CONTACT_INFORMATION.MAILING_ADDRESS.NAME_OF_COUNTRY' AND
        V2.[VALUE] <> 'CANADA' AND
        V2.DOCID = V1.DOCID AND
        D1.DOCID = V1.DOCID AND
        D1.LEVEL = 1 AND
        ABS(V2.BRANCHORDERSUM-V1.BRANCHORDERSUM) < D1.RVALUE
)
```

```
Q9: SELECT
    V2.*
FROM
    PV11 V1,
    PATH P1,
    PV12 V2,
    PATH P2,
    DOCUMENTRAVLUE D
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH = '.CATALOG.ITEM.@ID' AND
    V1.[VALUE]='I6' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH LIKE '.CATALOG.ITEM.AUTHORS.AUTHOR
    .CONTACT_INFORMATION.MAILING_ADDRESS%' AND
    V2.DOCID = V1.DOCID AND
    D.DOCID = V1.DOCID AND
    D.LEVEL = 1 AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```

APPENDIX A

```

Q10:SELECT
    VO.[VALUE]
FROM
    PV141 VO,
    PATH PO,
    PV13 V4,
    PATH P4,
    DOCUMENTRVALUE D
WHERE
    VO.PATHID = PO.[ID] AND
    PO.PATH = '.CATALOG.ITEM.PUBLISHER.NAME' AND
    V4.PATHID = P4.[ID] AND
    P4.PATH = '.CATALOG.ITEM.DATE_OF_RELEASE' AND
    V4.[VALUE] > '1990-01-01' AND
    V4.[VALUE] < '1991-01-01' AND
    VO.DOCID = V4.DOCID AND
    D.DOCID = VO.DOCID AND
    D.LEVEL = 1 AND
    ABS(VO.BRANCHORDERSUM-V4.BRANCHORDERSUM) < D.RVALUE AND
    VO.[VALUE] NOT IN(
        SELECT
            V3.[VALUE]
        FROM
            PATHVALUE V2, PATH P2, PATHVALUE V3, PATH P3
            , DOCUMENTRVALUE D1
        WHERE
            V2.PATHID = P2.[ID] AND
            P2.PATH = '.CATALOG.ITEM.PUBLISHER
            .CONTACT_INFORMATION.FAX_NUMBER' AND
            V3.PATHID = P3.[ID] AND
            P3.PATH = '.CATALOG.ITEM.PUBLISHER.NAME' AND
            V2.DOCID = V3.DOCID AND
            D1.DOCID = V2.DOCID AND
            D.LEVEL = 1 AND
            ABS(V3.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE )

Q11:SELECT
    V3.[VALUE]
FROM
    PV15 VO,
    PATH PO,
    PV16 V1,
    PATH P1,
    PV17 V2,
    PATH P2,
    PV18 V3,
    PATH P3,
    DOCUMENTRVALUE D
WHERE
    VO.PATHID = PO.[ID] AND
    PO.PATH = '.CATALOG.ITEM.ATTRIBUTES
    .SIZE_OF_BOOK.LENGTH' AND
    V1.PATHID = P1.[ID] AND
    P1.PATH = '.CATALOG.ITEM.ATTRIBUTES
    .SIZE_OF_BOOK.WIDTH' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH = '.CATALOG.ITEM.ATTRIBUTES
    .SIZE_OF_BOOK.HEIGHT' AND
    V3.PATHID = P3.[ID] AND
    P3.PATH = '.CATALOG.ITEM.TITLE' AND
    CAST(VO.[VALUE] AS FLOAT)*CAST(V1.[VALUE] AS FLOAT)
    CAST(V2.[VALUE] AS FLOAT) > 50000 AND
    V3.DOCID = VO.DOCID AND
    D.LEVEL = 1 AND
    V3.DOCID = V1.DOCID AND
    ABS(VO.BRANCHORDERSUM-V1.BRANCHORDERSUM) < D.RVALUE AND
    V3.DOCID = V2.DOCID AND

```

APPENDIX A

```
ABS(V0.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE AND
V0.DOCID = V3.DOCID AND
ABS(V0.BRANCHORDERSUM-V3.BRANCHORDERSUM) < D.RVALUE
```

```
Q12:SELECT
      V2.*
FROM
      PV19 V1,
      PATH P1,
      PV20 V2,
      PATH P2,
      DOCUMENTRVALUE D
WHERE
      V1.PATHID = P1.[ID] AND
      P1.PATH = 'ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
      V1.[VALUE]='PADMA SILLINCE' AND
      V2.PATHID = P2.[ID] AND
      P2.PATH = 'ARTICLE.PROLOG.TITLE' AND
      V1.DOCID = V2.DOCID AND
      V1.DOCID = D.DOCID AND
      D.LEVEL = 1 AND
      ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```

```
Q13:SELECT
      V2.*
FROM
      PV21 V1,
      PATH P1,
      PV22 V2,
      PATH P2
WHERE
      V1.PATHID = P1.[ID] AND
      P1.PATH = 'ARTICLE.@ID' AND
      V1.[VALUE]='8' AND
      V2.PATHID = P2.[ID] AND
      P2.PATH = 'ARTICLE.BODY.SECTION.@HEADING' AND
      V2.[VALUE] = 'INTRODUCTION' AND
      V1.DOCID = V2.DOCID
```

```
Q14:SELECT
      V2.*
FROM
      PV23 V1,
      PATH P1,
      PV20 V2,
      PATH P2
WHERE
      V1.PATHID = P1.[ID] AND
      P1.PATH = 'ARTICLE.BODY.ABSTRACT.P' AND
      V1.[VALUE] LIKE '%THE%HOCKEY%' OR
      V1.[VALUE] LIKE '%HOCKEY%THE%' AND
      V2.PATHID = P2.[ID] AND
      P2.PATH = 'ARTICLE.PROLOG.TITLE' AND
      V1.DOCID = V2.DOCID
```

```
Q15:SELECT
      V2.*,
      V3.*,
      V4.*,
      V5.*
FROM
      PV21 V1,
      PATH P1,
      P20 V2,
      PATH P2,
      PV19 V3,
```


APPENDIX A

```

PATH P3,
PV24 V4,
PATH P4,
PV23 V5,
PATH P5
WHERE
V1.PATHID = P1.[ID] AND
P1.PATH = '.ARTICLE.@ID' AND
V1.[VALUE]='5' AND
V2.PATHID = P2.[ID] AND
P2.PATH = '.ARTICLE.PROLOG.TITLE' AND
V3.PATHID = P3.[ID] AND
P3.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
V4.PATHID = P4.[ID] AND
P4.PATH = '.ARTICLE.PROLOG.DATELINE.DATE' AND
V5.PATHID = P5.[ID] AND
P5.PATH = '.ARTICLE.BODY.ABSTRACT'

```

```

Q16:SELECT
[VALUE]
FROM
PV19 V0,
PATH P0
WHERE
P0.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
V0.PATHID = P0.[ID] AND
V0.[VALUE] NOT IN
(
SELECT
DISTINCT(V2.[VALUE])
FROM
PV25 V1,
PATH P1,
PV19 V2,
PATH P2,
DOCUMENTRVALUE R1
WHERE
V1.PATHID = P1.[ID] AND
P1.PATH LIKE '.ARTICLE.PROLOG.AUTHORS.AUTHOR.CONTACT.%' AND
V2.PATHID = P2.[ID] AND
P2.PATH = '.ARTICLE.PROLOG.AUTHORS.AUTHOR.NAME' AND
V1.DOCID = V2.DOCID AND
V1.DOCID = R1.DOCID AND
R1.[LEVEL]=3 AND
ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < R1.RVALUE
)

```

```

Q17:SELECT
V2.*,
V3.*
FROM
PV26 V1,
PATH P1,
PV20 V2,
PATH P2,
PV23 V3,
PATH P3
WHERE
V1.PATHID = P1.[ID] AND
P1.PATH LIKE '.ARTICLE%.P' AND
V1.[VALUE] LIKE '%HOCKEY' AND
V2.PATHID = P2.[ID] AND
P2.PATH = '.ARTICLE.PROLOG.TITLE' AND
V3.PATHID = P3.[ID] AND
P3.PATH = '.ARTICLE.PROLOG.ABSTRACT' AND
V1.DOCID = V2.DOCID AND
V1.DOCID = V3.DOCID

```

APPENDIX A

```

Q18:SELECT
    V1.*
FROM
    PV26 V1,
    PATH P1,
    PV26 V2,
    PATH P2,
    DOCUMENTRVALUE D
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH = '.DICTIONARY.E.@ID' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH = '.DICTIONARY.E.HWG.HW' AND
    V2.[VALUE]='the' AND
    V1.DOCID = V2.DOCID AND
    V1.DOCID = D.DOCID AND
    D.LEVEL = 1 AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q19:SELECT
    V1.*
FROM
    PV26 V1,
    PATH P1,
    PV26 V2,
    PATH P2,
    DOCUMENTRVALUE D
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH LIKE '.DICTIONARY.E.%HW' AND
    V1.[VALUE]='AND' AND
    V2.PATHID = P2.[ID] AND
    P2.PATH LIKE '.DICTIONARY.E.SS.S.QP.%QT' AND
    V1.DOCID = V2.DOCID AND
    V1.DOCID = D.DOCID AND
    D.LEVEL = 1 AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q20:SELECT
    V2.*
FROM
    PV26 V1,
    PATH P1,
    PV26 V2,
    PATH P2,
    DOCUMENTRVALUE R
WHERE
    V1.PATHID = P1.[ID] AND
    P1.PATH LIKE '.DICTIONARY.E.HWG.HW' AND
    V1.[VALUE]='THE' AND
    V1.DOCID = R.DOCID AND
    R.[LEVEL]=3 AND
    V1.DOCID = V2.DOCID AND
    ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < R.RVALUE AND
    V3.PATHID = P3.[ID] AND
    P3.PATH = '.DICTIONARY.E.SS.S.QP.Q.A' AND
    V1.DOCID = V3.DOCID AND
    ABS(V1.BRANCHORDERSUM-V3.BRANCHORDERSUM) < R.RVALUE
ORDER BY
    V1.[VALUE]

Q21:SELECT
    V2.*
FROM
    PV26 V1,
    PATH P1,

```

APPENDIX A

```

PV26 V2,
PATH P2,
DOCUMENTRVALUE D
WHERE
  V1.PATHID = P1.[ID] AND
  P1.PATH LIKE '.DICTIONARY.E.%' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.DICTIONARY.E.HWG.HW' AND
  V1.[VALUE] LIKE '%HOCKEY%' AND
  V1.DOCID = V2.DOCID AND
  V1.DOCID = D.DOCID AND
  D.LEVEL = 2 AND
  ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q22:SELECT
      V1.*
JM                                     FR
PV26 V1,
PATH P1,
PV26 V2,
PATH P2,
DOCUMENTRVALUE D
RE                                     WH
  V1.PATHID = P1.[ID] AND
  P1.PATH = '.DICTIONARY.E.SS.S.QP.Q.QD' AND
  V1.[VALUE]='1900' AND
  V2.PATHID = P2.[ID] AND
  P2.PATH = '.DICTIONARY.E.HWG.HW' AND
  V1.DOCID = V2.DOCID AND
  V1.DOCID = D.DOCID AND
  D.LEVEL = 2 AND
  ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

```

A.4 SQL Queries for Recursive XQuery in SUCX-ENT

```

Q1: SELECT
      V2.*
FROM
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,
  PATH P2,
  ANCESTORINFO A1,
  ANCESTORINFO A2
WHERE
  P1.PATH LIKE '%TOPIC.TITLE' AND
  P1.ID = V1.PATHID AND
  V1.[VALUE]='PHOTOGRAPHY' AND
  P2.PATH LIKE '%TOPIC.DESRIPTION%' AND
  P2.ID = V2.PATHID AND
  V1.DOCID = A1.DOCID AND
  V2.DOCID = A2.DOCID AND
  A1.ANCESTORLEVEL = CHARINDEX('TOPIC', P1.PATH) AND
  A1.DOCID = A2.DOCID AND
  A2.ANCESTORLEVEL = A2.ANCESTORLEVEL AND
  A1.ANCESTORORDER = A2.ANCESTORORDER

LECT                                     Q2:SE
      V2.*
ROM                                     F
  PATHVALUE V1,
  PATH P1,
  PATHVALUE V2,

```

APPENDIX A

```

    PATH P2,
    ANCESTORINFO A1,
    ANCESTORINFO A2
WHERE
    P1.PATH LIKE '%TOPIC.TITLE' AND
    V1.[VALUE]='PHOTOGRAPHY' AND
    P1.ID = V1.PATHID AND
    P2.PATH LIKE '%TOPIC.DESRIPTION%' AND
    P2.ID = V2.PATHID AND
    V1.DOCID = A1.DOCID AND
    V2.DOCID = A2.DOCID AND
    A1.ANCESTORLEVEL >= CHARINDEX('TOPIC', P1.PATH) AND
    A1.DOCID = A2.DOCID AND
    A2.ANCESTORLEVEL = A2.ANCESTORLEVEL AND
    A1.ANCESTORORDER = A2.ANCESTORORDER

```

```

Q3: SELECT
    V2.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    ANCESTORINFO A1,
    ANCESTORINFO A2
WHERE
    P1.PATH LIKE '%TOPIC.LASTUPDATE' AND
    P1.ID = V1.PATHID AND
    MONTH(CONVERT(V1.[VALUE], SMALLDATETIME)) >= 10 AND
    P2.PATH LIKE '%TOPIC.DESRIPTION%' AND
    P2.ID = V2.PATHID AND
    V1.DOCID = A1.DOCID AND
    V2.DOCID = A2.DOCID AND
    A1.ANCESTORLEVEL = CHARINDEX('TOPIC', P1.PATH) AND
    A1.DOCID = A2.DOCID AND
    A2.ANCESTORLEVEL = A2.ANCESTORLEVEL AND
    A1.ANCESTORORDER = A2.ANCESTORORDER

```

```

Q4: SELECT
    V2.*
FROM
    PATHVALUE V1,
    PATH P1,
    PATHVALUE V2,
    PATH P2,
    ANCESTORINFO A1,
    ANCESTORINFO A2
WHERE
    P1.PATH LIKE '%TOPIC.LASTUPDATE' AND
    P1.ID = V1.PATHID AND
    MONTH(CONVERT(V1.[VALUE], SMALLDATETIME)) >= 10 AND
    P2.PATH LIKE '%TOPIC.DESRIPTION%' AND
    P2.ID = V2.PATHID AND
    V1.DOCID = A1.DOCID AND
    V2.DOCID = A2.DOCID AND
    A1.ANCESTORLEVEL >= CHARINDEX('TOPIC', P1.PATH) AND
    A1.DOCID = A2.DOCID AND
    A2.ANCESTORLEVEL = A2.ANCESTORLEVEL AND
    A1.ANCESTORORDER = A2.ANCESTORORDER

```

```

Q5: SELECT
    COUNT(DISTINCT(V1.[VALUE]))
FROM
    PATHVALUE AS V1, PATH P1
WHERE
    P1.PATH LIKE '.SITE.REGIONS%ITEM.NAME' AND
    V1.PATHID = P1.ID

```

APPENDIX A

```

Q5: SELECT
      COUNT(DISTINCT(V1.[VALUE]))
FROM
      PATHVALUE AS V1, PATH P1
WHERE
      P1.PATH LIKE '.SITE.REGIONS%ITEM.NAME' AND
      V1.PATHID = P1.ID

Q6: SELECT
      COUNT(DISTINCT(V1.[VALUE]))
FROM
      PATHVALUE AS V1, PATH P1
WHERE
      (
        (
          P1.PATH LIKE '.SITE.%ANNOTATION' AND
          V1.BRANCHORDER <= CHARINDEX('ANNOTATION', P1.PATH)) OR
          (
            P1.PATH LIKE '.SITE.%DESCRIPTION' AND
            V1.BRANCHORDER <= CHARINDEX('DESCRIPTION', P1.PATH)
          ) OR
          (
            P1.PATH LIKE '.SITE.%EMAIL' AND
            V1.BRANCHORDER <= CHARINDEX('EMAIL', P1.PATH)
          ) AND
          V1.PATHID = P1.ID
      )

Q7: SELECT
      V2.*
FROM
      PATHVALUE V1,
      PATH P1,
      PATHVALUE V2,
      PATH P2,
      ANCESTORINFO A1,
      ANCESTORINFO A2
WHERE
      P1.PATH LIKE '.SITE.REGIONS.AFRICA.ITEM.%DESCRIPTION%' AND
      P1.ID = V1.PATHID AND
      V1.[VALUE] LIKE '%GOLD%' AND
      P2.PATH = '.SITE.REGIONS.AFRICA.ITEM.NAME' AND
      P2.ID = V2.PATHID AND
      V1.DOCID = A1.DOCID AND
      A1.ANCESTORLEVEL = 4 AND
      V2.DOCID = A2.DOCID AND
      A1.DOCID = A2.DOCID AND
      A2.ANCESTORLEVEL = A2.ANCESTORLEVEL AND
      A1.ANCESTORORDER = A2.ANCESTORORDER

Q8: SELECT
      V2.*
FROM
      PATHVALUE V1,
      PATHVALUE V2,
      PATH P1,
      PATH P2,
      ANCESTORINFO A1,
      ANCESTORINFO A2
WHERE
      P1.PATH LIKE '.SPTR.ENTRY.REFERENCE.%AUTHORLIST.PERSON.@NAME' AND
      P1.ID = V1.PATHID AND
      V1.[VALUE]='MUELLER P.' AND
      V2.PATH = '.SPTR.ENTRY.ACCESSION' AND
      P2.ID = V2.PATHID AND
      V1.DOCID = A1.DOCID AND
      A1.ANCESTORLEVEL = 2 AND

```

APPENDIX A

```

V2.DOCID = A2.DOCID AND
A1.DOCID = A2.DOCID AND
A2.ANCESTORLEVEL = A2.ANCESTORLEVEL AND
A1.ANCESTORORDER = A2.ANCESTORORDER

```

```

Q9: SELECT
    V2.*
FROM
    PATHVALUE V1,
    PATHVALUE V2,
    PATH P1,
    PATH P2,
    ANCESTORINFO A1,
    ANCESTORINFO A2
WHERE
    P1.PATH LIKE '.SPTR.ENTRY.REFERENCE.%AUTHORLIST.PERSON.©NAME' AND
    P1.ID = V1.PATHID AND
    V1.[VALUE]='HERMANN R.' AND
    V2.PATH LIKE '.SPTR.ENTRY.REFERENCE%' AND
    P2.ID = V2.PATHID AND
    V1.DOCID = A1.DOCID AND
    A1.ANCESTORLEVEL = 3 AND
    V2.DOCID = A2.DOCID AND
    A1.DOCID = A2.DOCID AND
    A2.ANCESTORLEVEL = A2.ANCESTORLEVEL AND
    A1.ANCESTORORDER = A2.ANCESTORORDER

```

```

Q10:SELECT
    V2.*
FROM
    PATHVALUE V1,
    PATHVALUE V2,
    PATH P1,
    PATH P2,
    ANCESTORINFO A1,
    ANCESTORINFO A2
WHERE
    P1.PATH LIKE '.SPTR.ENTRY.REFERENCE.%@TYPE' P1.ID = V1.PATHID AND
    V1.[VALUE]='JOURNAL ARTICLE' AND
    V2.PATH = '.SPTR.ENTRY.ACCESSION' AND
    P2.ID = V2.PATHID AND
    V1.DOCID = A1.DOCID AND
    A1.ANCESTORLEVEL = 2 AND
    V2.DOCID = A2.DOCID AND
    A1.DOCID = A2.DOCID AND
    A2.ANCESTORLEVEL = A2.ANCESTORLEVEL AND
    A1.ANCESTORORDER = A2.ANCESTORORDER

```

A.5 SQL Queries for Recursive XQuery in SUCX-ENT++

```

Q1: SELECT
    V2.*
FROM
    PATHVALUE V1, PATH P1, PATHVALUE V2, PATH P2, DOCUMENTRVALUE D
WHERE
    P1.PATHEXP LIKE '%TOPIC.TITLE' AND
    P1.ID = V1.PATHID AND
    V1.LEAFVALUE='PHOTOGRAPHY' AND
    P2.PATHEXP LIKE '%TOPIC.DESRIPTION%' AND
    P2.ID = V2.PATHID AND
    V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
    D.LEVEL = INDEXOF('TOPIC', P1.PATHEXP) - 1
    AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

```

APPENDIX A

```

Q2: SELECT
      V2.*
    FROM
      PATHVALUE V1, PATH P1, PATHVALUE V2, PATH P2, DOCUMENTRVALUE D
    WHERE
      P1.PATHEXP LIKE '%TOPIC.TITLE' AND
      P1.ID = V1.PATHID AND
      V1.LEAFVALUE='PHOTOGRAPHY' AND
      P2.PATHEXP LIKE '%TOPIC.DESRIPTION%' AND
      P2.ID = V2.PATHID AND
      V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
      D.LEVEL = INDEXOF('TOPIC', P1.PATHEXP) - 1
      AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q3: SELECT
      V2.*
    FROM
      PATHVALUE V1, PATH P1, PATHVALUE V2, PATH P2, DOCUMENTRVALUE D
    WHERE
      P1.PATHEXP LIKE '%TOPIC.LASTUPDATE' AND
      P1.ID = V1.PATHID AND
      MONTH(CONVERT(V1.VALUE, SMALLDATETIME)) >= 10 AND
      P2.PATHEXP LIKE '%TOPIC.DESRIPTION%' AND
      P2.ID = V2.PATHID AND
      V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
      D.LEVEL = INDEXOF('TOPIC', P1.PATHEXP) - 1
      AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q4: SELECT
      V2.*
    FROM
      PATHVALUE V1, PATH P1, PATHVALUE V2, PATH P2, DOCUMENTRVALUE D
    WHERE
      P1.PATHEXP LIKE '%TOPIC.LASTUPDATE' AND
      P1.ID = V1.PATHID AND
      MONTH(CONVERT(V1.VALUE, SMALLDATETIME)) >= 10 AND
      P2.PATHEXP LIKE '%TOPIC.DESRIPTION%' AND
      P2.ID = V2.PATHID AND
      V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
      D.LEVEL = INDEXOF('TOPIC', P1.PATHEXP) - 1
      AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE

Q5: SELECT
      COUNT(DISTINCT(V1.VALUE))
    FROM
      PATHVALUE AS V1, PATH P1
    WHERE
      P1.PATHEXP LIKE '.SITE.REGIONS%ITEM.NAME' AND
      V1.PATHID = P1.ID

Q6: SELECT
      COUNT(DISTINCT(V1.VALUE))
    FROM
      PATHVALUE AS V1, PATH P1
    WHERE
      (
        (
          P1.PATHEXP LIKE '.SITE.%ANNOTATION%' AND
          V1.BRANCHORDER <= INDEXOF('ANNOTATION', P1.PATHEXP)) OR
        (
          P1.PATHEXP LIKE '.SITE.%DESCRIPTION%' AND
          V1.BRANCHORDER <= INDEXOF('DESCRIPTION', P1.PATHEXP)
        ) OR
        (
          P1.PATHEXP LIKE '.SITE.%EMAIL%' AND
          V1.BRANCHORDER <= INDEXOF('EMAIL', P1.PATHEXP)
        ) AND
        V1.PATHID = P1.ID
      )

```


APPENDIX A

```
Q7: SELECT
      V2.*
    FROM
      PATHVALUE V1, PATH P1, PATHVALUE V2, PATH P2, DOCUMENTRVALUE D
    WHERE
      P1.PATHEXP LIKE '.SITE.REGIONS.AFRICA.ITEM.%DESCRIPTION%' AND
      P1.ID = V1.PATHID AND
      V1.LEAFVALUE LIKE '%GOLD%' AND
      P2.PATHEXP = '.SITE.REGIONS.AFRICA.ITEM.NAME' AND
      P2.ID = V2.PATHID AND
      V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
      D.LEVEL = 3 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```

```
Q8: SELECT
      V2.*
    FROM
      PATHVALUE V1, PATHVALUE V2, PATH P1, PATH P2, DOCUMENTRVALUE D
    WHERE
      P1.PATHEXP LIKE '.SPTR.ENTRY.REFERENCE.%AUTHORLIST.PERSON.@NAME' AND
      P1.ID = V1.PATHID AND
      V1.LEAFVALUE='MUELLER P.' AND
      V2.PATHEXP = '.SPTR.ENTRY.ACCESSION' AND
      P2.ID = V2.PATHID AND
      V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
      D.LEVEL = 1 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```

```
Q9: SELECT
      V2.*
    FROM
      PATHVALUE V1, PATHVALUE V2, PATH P1, PATH P2, DOCUMENTRVALUE D
    WHERE
      P1.PATHEXP LIKE '.SPTR.ENTRY.REFERENCE.%PERSON.@NAME'
      AND P1.ID = V1.PATHID AND
      V1.LEAFVALUE='HERMANN R.' AND
      V2.PATHEXP LIKE '.SPTR.ENTRY.REFERENCE%' AND
      P2.ID = V2.PATHID AND
      V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
      D.LEVEL = 1 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```

```
Q10:SELECT
      V2.*
    FROM
      PATHVALUE V1, PATHVALUE V2, PATH P1, PATH P2, DOCUMENTRVALUE D
    WHERE
      P1.PATHEXP LIKE '.SPTR.ENTRY.REFERENCE.%@TYPE' P1.ID = V1.PATHID AND
      V1.LEAFVALUE='JOURNAL ARTICLE' AND
      V2.PATHEXP = '.SPTR.ENTRY.ACCESSION' AND
      P2.ID = V2.PATHID AND
      V1.DOCID = V2.DOCID AND V1.DOCID = D.DOCID AND
      D.LEVEL = 1 AND ABS(V1.BRANCHORDERSUM-V2.BRANCHORDERSUM) < D.RVALUE
```

APPENDIX A

Q1	/order/@id order/customer_id	PV1 PV2
Q2	/order/@id order/customer_id order/order_lines/order_line	PV1 PV2 PV3
Q3	/order/@id order/customer_id order/order_lines/order_line order//item_id	PV1 PV2 PV3 PV4
Q4	/order/@id order/customer_id order/order_lines/order_line order//item_id order/total order/order_date order/ship_type	PV1 PV2 PV3 PV4 PV5 PV6 PV7
Q5	/order/@id order/customer_id order/order_lines/order_line order//item_id order/total order/order_date order/ship_type	PV1 PV2 PV3 PV4 PV5 PV6 PV7
Q6	/order/@id order/customer_id order/order_lines/order_line order//item_id order/total order/order_date order/ship_type order/order_status customers/customer/first_name customers/customer/@id customers/customer/last_name customers/customer/phone_number	PV1 PV2 PV3 PV4 PV5 PV6 PV7 PV71 PV8 PV81 PV9 PV10
Q7	/order/@id order/customer_id order/order_lines/order_line order//item_id order/total order/order_date order/ship_type customers/customer/first_name customers/customer/last_name catalog/item/@id	PV1 PV2 PV3 PV4 PV5 PV6 PV7 PV8 PV9 PV11
Q8	/order/@id order/customer_id order/order_lines/order_line order//item_id order/total order/order_date order/ship_type customers/customer/first_name customers/customer/last_name catalog/item/@id catalog/item/authors/author/contact_information/mailling_address/name_of_country	PV1 PV2 PV3 PV4 PV5 PV6 PV7 PV8 PV9 PV11 PV12
Q9	/order/@id order/customer_id order/order_lines/order_line order//item_id order/total order/order_date order/ship_type customers/customer/first_name customers/customer/last_name catalog/item/@id catalog/item/authors/author/contact_information/mailling_address	PV1 PV2 PV3 PV4 PV5 PV6 PV7 PV8 PV9 PV11 PV12

APPENDIX A

Q10	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order//item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/ mailing_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
Q11	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order//item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/ mailing_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
Q12	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order//item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/ mailing_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
Q13	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order//item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/ mailing_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
	article/@id	PV21
	article/body/section/@heading	PV22

APPENDIX A

Q14	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order/item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/mailling_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
	article/@id	PV21
	article/body/section/@heading	PV22
	article/body/abstract/p	PV23
Q15	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order/item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/mailling_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
	article/@id	PV21
	article/body/section/@heading	PV22
	article/body/abstract	PV23
	article/prolog/dateline/date	PV24
Q16	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order/item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/mailling_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
	article/@id	PV21
	article/body/section/@heading	PV22
	article/body/abstract	PV23
	article/prolog/dateline/date	PV24
	article/prolog/author/contact	PV25

APPENDIX A

Q17	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order//item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/mailling_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
	article/@id	PV21
	article/body/section/@heading	PV22
	article/body/abstract	PV23
	article/prolog/dateline/date	PV24
	article/prolog/author/contact	PV25
	article//p	PV26
Q18	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order//item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/mailling_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
	article/@id	PV21
	article/body/section/@heading	PV22
	article/body/abstract	PV23
	article/prolog/dateline/date	PV24
	article/prolog/author/contact	PV25
	article//p	PV26
Q19	dictionary/e/hwg/hw	PV27
	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order//item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/mailling_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
	article/@id	PV21
	article/body/section/@heading	PV22
	article/body/abstract	PV23
	article/prolog/dateline/date	PV24
	article/prolog/author/contact	PV25
	article//p	PV26
	dictionary/e/*/hw	PV27
	dictionary/e/ss/s/qp/*/qt	PV28

APPENDIX A

Q20	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order//item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/mailling_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
	article/@id	PV21
	article/body/section/@heading	PV22
	article/body/abstract	PV23
	article/prolog/dateline/date	PV24
	article/prolog/author/contact	PV25
	article//p	PV26
	dictionary/e/*/hw	PV27
	dictionary/e/ss/s/qp/*/qt	PV28
	dictionary/e/ss/s/qp/q/qd	PV29
Q21	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order//item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/mailling_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
	article/@id	PV21
	article/body/section/@heading	PV22
	article/body/abstract	PV23
	article/prolog/dateline/date	PV24
	article/prolog/author/contact	PV25
	article//p	PV26
	dictionary/e	PV27

APPENDIX A

Q22	/order/@id	PV1
	order/customer_id	PV2
	order/order_lines/order_line	PV3
	order//item_id	PV4
	order/total	PV5
	order/order_date	PV6
	order/ship_type	PV7
	customers/customer/first_name	PV8
	customers/customer/last_name	PV9
	catalog/item/@id	PV11
	catalog/item/authors/author/contact_information/ mailing_address	PV12
	catalog/item/date_of_release	PV13
	catalog/item/publisher/contact_information/FAX_NUMBER	PV14
	catalog/item/attributes/size_of_book/length	PV15
	catalog/item/attributes/size_of_book/width	PV16
	catalog/item/attributes/size_of_book/height	PV17
	catalog/item/title	PV18
	article/prolog/author/name	PV19
	article/prolog/title	PV20
	article/@id	PV21
	article/body/section/@heading	PV22
	article/body/abstract	PV23
	article/prolog/dateline/date	PV23
	article/prolog/author/contact	PV24
	article//p	PV25
	dictionary/e	PV26

Figure A-1: Queries and Q-Expression-Sets.

Appendix B

Query Timings

APPENDIX B

Data Set	Shared-Inlining	SUCXENT	XParent	SUCXENT++
DCMD (10MB)	194179	527144	778190	137214
DCMD (100MB)	2321582	6273013	9338280	1600235
DCMD (1000MB)	20917453	81549176	140074200	14402115 0
DCSD(10MB)	80719	157578	281973	87088
DCSD(100mb)	320451	1464218	3071337	245481
DCSD(1000mb)	801127	11713744	27642033	589154
TCMD(10MB)	12708	35198	76177	11412
TCMD(100MB)	177912	344940	837947	169591
TCMD(1000MB)	1067472	5027682	5865629	983627
TCSD(10MB)	125135	211259	289103	112231
TCSD(100MB)	1115579	2302723	3266863	1041657
TCSD(1000MB)	2844726	18536920	29434443	2552059

Figure B-1: Insertion times.

Data Set	Shared-Inlining	SUCXENT	XParent	SUCXENT++
DCMD (10MB)	33	31	35	31.1
DCMD (100MB)	268	262	290	262
DCMD (1000MB)	2820	2811	2901	2811
DCSD(10MB)	31	27	30	27
DCSD(100mb)	239	217	231	216
DCSD(1000mb)	1774	1711	1790	1711
TCMD(10MB)	44	37	38	37
TCMD(100MB)	390	282	307	282
TCMD(1000MB)	3711	2639	2932	2639
TCSD(10MB)	47	32	34	32
TCSD(100MB)	455	277	293	277
TCSD(1000MB)	5011	2601	2881	2601

Figure B-2: Extraction times.

Query	Shared -Inlining	SUCXENT	XParent	SUCXENT++	SUCXENT++ Opt1	SUCXENT++ Opt2
Q1	49	50	54	92	55	35
Q2	75	120	120	116	75	8
Q3	85	160	192	175	140	79
Q4	118	4512	6159	1295	251	150
Q5	77	1010	1108	591	294	110
Q6	187	3101	4673	1431	176	105
Q7	131	299	320	604	559	540
Q8	581	3722	6810	1743	641	552
Q9	56	493	614	120	74	19
Q10	136	28100	40019	8371	261	175
Q11	169	9732	15101	1432	178	74
Q12	32	50	50	57	30	5
Q13	114	155	210	110	58	15
Q14	84	179	220	126	33	16
Q15	104	210	310	168	100	69
Q16	92	2379	3108	1432	310	250
Q17	8594	863	1410	411	329	183
Q18	1193	3376	4419	753	113	25
Q19	3198	1928	2716	534	871	112
Q20	140	328	320	158	54	33
Q21	819	310	300	127	50	15
Q22	3473	320	335	140	50	29

Figure B-3: Query timings (ms) for the 10 MB data set (XBench).

APPENDIX B

Query	Shared -Inlining	SUCXENT	XParent	SUCXENT++	SUCXENT++ Opt1	SUCXENT++ Opt2
Q1	69	901	1050	3340	120	53
Q2	110	3108	3108	4346	115	13
Q3	198	2416	3479	3206	191	126
Q4	145	31178	58094	2980	181	197
Q5	85	5979	7196	6610	811	160
Q6	622	15660	35562	4218	400	177
Q7	194	640	733	761	580	550
Q8	3137	577282	1932545	17217	4086	1986
Q9	70	458	711	120	75	50
Q10	830	fails	fails	33539	5132	120
Q11	254	150457	362259	9800	762	391
Q12	42	93	112	114	62	40
Q13	152	326	597	250	140	76
Q14	143	1636	2471	1252	54	5
Q15	130	1911	4164	1168	749	481
Q16	405	3592	6131	2098	803	565
Q17	26641	7617	12325	4923	2912	1755
Q18	4891	37474	64205	6850	237	15
Q19	13112	34897	69252	7511	4760	710
Q20	1490	1030	1909	402	260	115
Q21	2621	1006	1380	540	265	215
Q22	14934	1643	2104	471	170	110

Figure B-4: Query timings (ms) for the 100 MB data set (XBench).

Query	Shared -Inlining	SUCXENT	XParent	SUCXENT++	SUCXENT++ Opt1	SUCXENT++ Opt2
Q1	88	3658	5255	5010	126	79
Q2	210	3247	3247	5127	132	20
Q3	295	4325	8968	6730	229	202
Q4	196	58638	198864	5364	205	257
Q5	123	12556	18186	8593	835	231
Q6	874	127629	658153	4598	404	300
Q7	547	3187	4181	913	592	660
Q8	6752	fails	fails	146100	10624	4290
Q9	84	546	1313	126	79	83
Q10	1877	fails	fails	204588	15909	124
Q11	355	1849113	2719583	69580	1600	700
Q12	61	180	214	222	66	78
Q13	228	846	2852	563	249	158
Q14	218	3354	7653	2567	57	10
Q15	156	18900	89748	6973	1438	683
Q16	1785	5823	16963	3147	940	848
Q17	162427	59003	164155	23750	11678	6027
Q18	16821	492778	fails	14728	510	32
Q19	60122	310930	fails	16524	13423	1129
Q20	2114	8353	10226	3102	2017	1115
Q21	4996	3279	5122	1596	910	419
Q22	59736	7399	12882	1910	870	701

Figure B-5: Query timings (ms) for the 1 GB data set (XBench).

APPENDIX B

Query	Shared -Inlining	SUCXENT	XParent	SUCXENT++	SUCXENT++ Opt1	SUCXENT++ Opt2
Q1	350	610	669	650	500	233
Q2	2405	2010	2213	1995	815	262
Q3	1095	7210	8039	4115	1595	329
Q4	51390	34190	37232	28210	14915	1070
Q5	1610	1700	1898	1594	1194	341
Q6	210	8590	9560	7512	1112	385
Q7	9100	13100	14527	7012	1120	970
Q8	210	520	582	200	154	64
Q9	2121	3210	3601	387	287	98
Q10	1504	10010	11312	4810	740	519

Figure B-6: Query timings (ms) for the recursive data sets.

Query	Increasing Workload	Decreasing Workload
Q1	1540	41
Q2	853	39
Q3	2081	38
Q4	1564	69
Q5	0	113
Q6	1816	0
Q7	2142	45
Q8	11438	29
Q9	47692	432
Q10	25138	72
Q11	5890	97
Q12	1265	49
Q13	12254	60
Q14	3653	36
Q15	1145	751
Q16	2209	51
Q17	1532	65
Q18	1381	47
Q19	4318	84
Q20	92318	72
Q21	177542	269860
Q22	0	0

Figure B-7: Maintenance cost (ms) with change in query workload for partitioning.

Query	Sum of prefetching ops.	Perceived response time	Normal execution time
Q1	415	257	363
Q2	1791	878	3785
Q3	207	455	598
Q4	838	689	1804
Q5	394	222	344
Q6	2095	1002	1343
Q7	1008	646	668
Q8	1871	748	807
Q9	429	207	265
Q10	2743	447	1761

Figure B-8: Prefetching timings (300 MB).

APPENDIX B

Query	Sum of prefetching ops.	Perceived response time	Normal execution time
Q1	624	500	624
Q2	1649	789	5030
Q3	343	674	1365
Q4	1217	714	6627
Q5	598	500	598
Q6	1929	1053	1785
Q7	1598	736	1441
Q8	2522	789	2481
Q9	658	500	658
Q10	2891	500	3310

Figure B-9: Prefetching timings (600 MB).

Query	Sum of prefetching ops.	Perceived response time	Normal execution time
Q1	1020	566	1072
Q2	1642	867	6684
Q3	726	777	3115
Q4	3964	740	24348
Q5	990	537	1040
Q6	1920	1106	2372
Q7	3179	839	3107
Q8	6704	833	7628
Q9	1470	583	1635
Q10	4229	559	6221

Figure B-10: Prefetching timings (1200 MB).

References

- [1] Berkeley DB 4.3. <http://www.berkeleydb.com/>, 2003.
- [2] XQuery 1.0 and XPath 2.0 data model. <http://www.w3c.org/TR/xquery>, 2003.
- [3] X-hive, a native XML database system. <http://www.x-hive.com>, 2003.
- [4] The XML DB initiative/. <http://www.xmldb.org/>, 2003.
- [5] Serge Abiteboul. Xyleme: A dynamic warehouse for XML data of the web. In <http://www.xyleme.wm>, 2002.
- [6] F. Achard. Xml, bioinformatics, and data integration. *Bioinformatics*, 17:115–125, 2001.
- [7] Sanjay Agrawal, S. Chaudhri, and Vivek Narasayya. Automated selection of materialized views and indexes for SQL databases. In *VLDB*, pages 359–370, 2000.
- [8] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *ACM SIGMOD*, pages 359–370, 2004.
- [9] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, and Jignesh M. Patel. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, pages 141–152, 2002.
- [10] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost-based approach to XML storage. In *18th International Conference on Data Engineering*, pages 64–76, 2002.
- [11] Ronald Bourret. Xml database products. <http://www.rpbouret.com/xml/>, 2004.
- [12] Daniela Braga and Alessandro Campi. A graphical environment to query XML data with xquery. In *WISE*, pages 31–40, 2003.
- [13] Stuart K. Card, Thomas P. Moran, and Allen Newell. The keystroke-level model for user performance time with interactive systems. *Commun. ACM*, 23(7):396–410, 1980.

REFERENCES

- [14] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal data partitioning in database design. In ACM SIGMOD, pages 128–136, 1982.
- [15] L. Chen and E. Rundensteiner. Ace-xq: A cache-aware xquery answering system, 2002.
- [16] Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting twig matches in a tree. In ICDE, pages 595–604, 2001.
- [17] S. Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In VLDB, pages 263–274, 2002.
- [18] Microsoft Corporation. SQL server technical resource kit. Technical Report, <http://www.microsoft.com/sql/techinfo/reskit/default.asp>, 2003.
- [19] G. Graefe (ed.). Special issue on query processing in commercial database management systems. IEEE Data Engineering, 16:4, 1993.
- [20] D. Florescu and D. Kossman. Storing and querying XML data using an RDBMS. In IEEE Data Engineering Bulletin, 1999.
- [21] Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jerome Simeon. Statix: making XML count. In ACM SIGMOD, pages 181–191, 2002.
- [22] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. Database system implementation. 2002.
- [23] Rao J., Zhang C., Lohman G., and Megiddo N. Automating physical database design in a parallel database. In ACM SIGMOD, pages 558–569, 2002.
- [24] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. The VLDB Journal, 11(4):274–291, 2002.
- [25] H. Jiang, H. Lu, W. Wang, and J. X. Yu. Path materialization revisited: An efficient storage model for XML data. Australian Institute of Computer Ethics Conference, 2000.
- [26] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. Xr-tree: Indexing XML data for efficient structural joins. In VLDB, pages 253–263, 2003.
- [27] Haifeng Jiang, Hongjun Lu, Wei Wang, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. In VLDB, pages 273–284, 2003.

REFERENCES

- [28] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of XML data. In ICDE, 2000.
- [29] Swiss-Prot Protein knowledgebase. <http://us.expasy.org>. 2003.
- [30] R. Krishnamurthy and V. T. Chakarvarthy et al. Recursive XML schemas, recursive XML queries, and relational storage: XML-to-SQL query translation. In ICDE, pages 42–53, 2004.
- [31] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML to SQL query translation literature: The state of the art and open problem. In XML Database Symposium, *XSym*, pages 31–38, 2003.
- [32] J. Li, S.K. Ng, and L. Wong. Bioinformatics adventures in database research. In Conference on Database Theory, pages 31–46, 2003.
- [33] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In The VLDB Journal, pages 361–370, 2001.
- [34] H. Liefke. Horizontal query optimization on ordered semistructured data. *WebDB (Informal Proceedings)*, pages 61–66, 1999.
- [35] J. McHugh and J. Widom. Query optimization for XML. The VLDB Journal, 1999.
- [36] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. ACM SIGMOD Record, 26(3):54–66, 1997.
- [37] Jim Melton and Andrew Eisenberg. Why use SQL instead of an oodbms. 2003.
- [38] Shamkant B. Navathe and Minyoung Ra. Vertical partitioning for database design: A graphical algorithm. In ACM SIGMOD, pages 440–450, 1989.
- [39] Stratos Papadomanolakis and Anastasia Ailamaki. Autopart: Automatic schema design for large scientific databases using data partitioning. In International Conference on SSDBM., 2002.
- [40] N. Polyzotis and Y. Ioannidis. Speculative query processing. In Biennial Conference on Innovative Data Systems Research, CIDR, 2003.
- [41] Neoklis Polyzotis and Minos Garofalakis. Statistical synopses for graph-structured XML databases. In ACM SIGMOD, pages 358–369, 2002.
- [42] Neoklis Polyzotis and Minos Garofalakis. Structure and value synopses for XML data graphs. In VLDB, pages 466–477, 2002.

REFERENCES

- [43] S. Prakash and S. S. Bhowmick. Efficient XML recursive query processing using relational database systems. *ER*, pages 493–510, 2004.
- [44] S. Prakash and S. S. Bhowmick. SUCXENT: An efficient path-based approach to store and query XML data. *DEXA*, pages 285–295, 2004.
- [45] M. P. Consens, M. T. Özsu, D. DeHaan, D. Toman. A COMPREHENSIVE XQUERY TO SQL TRANSLATION USING DYNAMIC INTERVAL CODING. IN *ACM SIGMOD*, PAGES 623–634, 2003.
- [46] DB2 XML EXTENDER. [HTTP://WWW-3.IBM.COM/SOFTWARE/DATA/DB2/EXTENDERS/XMLEXT/INDEX.HTML](http://www-3.ibm.com/software/data/db2/extenders/xmlext/index.html). 2003.
- [47] ORACLE9i XML DATABASE DEVELOPER'S GUIDE ORACLE XML DB RELEASE 2 (9.2). [HTTP://OTN.ORACLE.COM/TECH/XML/XMLDB/CONTENT.HTML](http://otn.oracle.com/tech/xml/xmlldb/content.html). 2003.
- [48] A. SCHMIDT, M. L. KERSTEN, M. WINDHOUEW, AND F. WAAS. EFFICIENT RELATIONAL STORAGE AND RETRIEVAL OF XML DOCUMENTS. IN *WebDB (Informal Proceedings)*, 2000.
- [49] J. SHANMUGASUNDARAM AND K. TUFT ET AL. RELATIONAL DATABASES FOR QUERYING XML DOCUMENTS: LIMITATIONS AND OPPORTUNITIES. IN *VLDB*, PAGES 302–314, 1999.
- [50] ECLIPSE COMMUNITY SITE. [HTTP://WWW.ECLIPSE.ORG](http://www.eclipse.org). 2003.
- [51] IGOR TATARINOV, STRATIS D. VIGLAS, KEVIN BEYER, JAYAVEL SHANMUGASUNDARAM, EUGENE SHEKITA, AND CHUN ZHANG. STORING AND QUERYING ORDERED XML USING A RELATIONAL DATABASE SYSTEM. IN *ACM Sigmod Record*, PAGES 204–215, 2002.
- [52] F. TIAN, D. DEWITT, J. CHEN, AND C. ZHANG. THE DESIGN AND PERFORMANCE EVALUATION OF ALTERNATIVE XML STORAGE STRATEGIES. *ACM Sigmod Record*, 31(1):5–10, 2002.
- [53] L. WONG. TECHNOLOGIES FOR INTEGRATING BIOLOGICAL DATA. *Bioinformatics*, 3(4), 2002.
- [54] Y. WU, J. M. PATEL, AND H. V. JAGADISH. STRUCTURAL JOIN ORDER SELECTION FOR XML QUERY OPTIMIZATION. *ICDE*, PAGES 443–454, 2003.
- [55] LIANG HUAI YANG, MONG LI LEE, AND WYNNE HSU. EFFICIENT MINING OF XML QUERY PATTERNS FOR CACHING. IN *VLDB*, PAGES 69–80, 2003.
- [56] BENJAMIN BIN YAO, M. TAMER ZSU, AND NITIN KHANDELWAL. XBENCH BENCHMARK AND PERFORMANCE TESTING OF XML DBMS. IN *ICDE*, PAGES 621–633, 2004.

REFERENCES

- [57] MASATOSHI YOSHIKAWA, TOSHIWAKI AMAGASA, T. SHIMURA, AND SHUNSUKE UEMURA. XREL: A PATH-BASED APPROACH TO STORAGE AND RETRIEVAL OF XML DOCUMENTS USING RELATIONAL DATABASES. *ACM Transactions on Internet Technology*, 1(1):110–141, 2001.
- [58] CHUN ZHANG, JEFFREY NAUGHTON, DAVID DEWITT, QIONG LUO, AND GUY LOHMAN. ON SUPPORTING CONTAINMENT QUERIES IN RELATIONAL DATABASE MANAGEMENT SYSTEMS. IN *ACM SIGMOD*, PAGES 425–436. 2001.
- [59] MOSHE M. ZLOOF. QUERY-BY-EXAMPLE: A DATABASE LANGUAGE. IN *IBM Systems Journal*, VOLUME 16(4), PAGES 324–343, 1977.