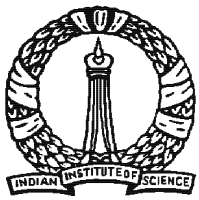
Schema-based Statistics and Storage for XML

A Thesis
Submitted for the Degree of
Doctor of Philosophy
in the Faculty of Engineering

By
Maya Ramanath



Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

April 2006

Acknowledgements

First and foremost, I would like to thank my advisor, Jayant Haritsa. Thanks to Jayant for teaching me about research, writing about my research as well as presenting my research. I thank him for setting high standards in the lab and for providing a great work environment with all possible facilities one would ever need. Finally, he was and continues to be an example to the rest of us with his hard work and dedication to research.

I thank Juliana Freire who has not only been my mentor and collaborator throughout my Ph.D years, but also a good friend. She is the most hard-working, smartest and toughest person I know and continues to be an inspiration to me in many ways. Discussions with her helped me clarify my ideas and writing papers with her gave me confidence in my work.

My other collaborators, Prasan Roy, Jerome Simeon and Lingzhi Zhang have helped at various phases in my Ph.D. Apart from their technical input which was always excellent, I thank them for teaching me how to work in groups and enriching my PhD experience.

I thank Prof. N. Balakrishnan, Prof. Matthew Jacob, Prof. R. Govindarajan and Prof. S.K. Nandy for their help and support. Special thanks to the office people who were a tremendous help with the administrative side of things – Rajalakshmi, Sarala, Nagamanjari, Triveni, Mallika, Kavitha, Govindaswamy, Gopakumar, Shashi, Shekhar, Shivanna and many others. Thanks to the CMC staff – Anant, Raju, Gajanan – who fixed my machines on time :-).

Working in the same lab for so many years can be difficult without a lot of friends. Thanks to Kumaran, Vikram and Suresha for giving me company in my Ph.D. years. Thanks to Aditya, Bharat, Chaitra, Kumaran and Vikram for giving me amazing company

for amazing amounts of time at the tea board and coffee board. I miss those daytime as well as midnight trips when we talked about everything under the sun (and moon)!

Srikanta – first a friend, then a close friend and finally my life partner! I did not include him in the above lists because he deserves a few pages of gratitude all for himself. Without him, I may not have completed my Ph.D. Always there for me in my moments of despair as well as my moments of joy. We struggled together for our PhDs and that makes our bond even more special. He supported me in all the ways that I could possibly want and then some!

Thanks to my mother-in-law who was the most kind and non-demanding mother-in-law that one could ever wish for. It made my academic life smooth and my married life a joy. I will miss her. Thanks to my father-in-law for his understanding and support. Thanks to Prasad and Gayathri, for taking on many responsibilities when we were busy with our seemingly never-ending student lives. Thanks to my cute little nephew Vishnu – though he did not give me any technical input, his smiles and friendliness were input enough :-))!!

No words can express my gratitude to my parents without whom none of this would have been possible. Thanks to my father for teaching me about life and philosophy and instilling in me the confidence and the ambition to work towards a Ph.D. He sets the highest possible standards for himself and it is my constant endeavour to live up to those standards that has made me the person I am today. Thanks to my mother for her love and affection, her confidence in my abilities and her pride in my achievements. Always eager to see me succeed and always willing take on more trouble to allow me to concentrate on my studies has ensured that I never lost focus (and always had good food to eat ;-). All that I am, I owe to my parents. And thanks of course to my brother, Madhava, who took away the TV remote from me so that I worked even when at home ;-)).

Finally, I would like to thank my daughter – oh wait, she wasn't born yet at that time – she still had 3 months to go!!

Abstract

XML (eXtensible Markup Language) is a highly flexible text format that has become the defacto standard for electronic publishing and data exchange, especially on the Web. XML data is tree-structured and can be described by an XML Schema, which provides types and regular expression constructs to concisely describe the document format. XQuery, a declarative query language, supports the extraction and transformation of XML data.

Due to its flexible and nested format, the storage and query processing of XML data throws up a variety of new challenges, not addressed by the classical relational approach which is predicated on rigid flat-structured schemas. In this thesis, we address three important issues arising in the XML context: First, we propose StatiX, a framework for XML data summarization and query result cardinality estimation, that organically handles both structural and value predicates. A rich set of schema transformations are used to improve the accuracy of the summary. Through detailed experimentation with a representative set of XQuery queries on both real and synthetic XML data, we show that StatiX produces concise, accurate and flexible summaries.

Second, to cater to dynamic XML applications that frequently update their underlying data, we propose the IMAX algorithm for incrementally maintaining StatiX summaries. IMAX incorporates novel techniques for estimating both the positions of the update and the ordinals of elements in the update fragment – a crucial requirement in the ordered data model of XML. An experimental evaluation shows that it provides comparable accuracy to a compute-from-scratch approach at a fraction of the runtime cost.

Finally, to design efficient storage layouts for XML data on the ubiquitous relational database engines, we propose FleXMap, a framework that uses a wide range of transformations on the XML Schema description for characterizing the equivalent relational configuration space. A variety of greedy search algorithms built on the FleXMap framework are experimentally shown to output cost-efficient relational configurations as compared to prior approaches.

In summary, this thesis presents a toolkit for effectively supporting the highly popular XML world-view on the underlying storage and processing engines.

Publications

- "IMAX: Incremental Maintenance of Schema-based XML Statistics"
  M. Ramanath, L. Zhang, J. Freire and J. Haritsa
  Proc. of the 21st IEEE Intl. Conf. on Data Engineering (ICDE), Tokyo, Japan, April 2005, pgs. 273-284

- "A Flexible Infrastructure for Gathering XML Statistics and Estimating Query Cardinality" (demo)
  J. Freire, M. Ramanath and L. Zhang
  Proc. of the 20th IEEE Intl. Conf. on Data Engineering (ICDE), Boston, USA, March 2004, pg. 857

- "Searching for Efficient XML-to-Relational Mappings"
  M. Ramanath, J. Freire, J. Haritsa and P. Roy
  Proc. of the 1st Intl. XML Database Symposium (XSym), Berlin, Germany, September 2003, pgs. 19-36

- "Bridging the XML-Relational Divide with LegoDB: A Demonstration" (demo)
  P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy and J. Simeon
  Proc. of the 19th IEEE Intl. Conf. on Data Engineering (ICDE), Bangalore, India, March 2003, pgs. 759-761

- "LegoDB: Customizing Relational Storage for XML Documents" (demo)
  P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy and J. Simeon
  Proc. of the 28th Intl. Conf. on Very Large Data Bases (VLDB), Hong Kong, China, August 2002, pgs. 1091-1094

- "StatiX: Making XML Count"
  J. Freire, J. Haritsa, M. Ramanath, P. Roy and J. Simeon
  Proc. of the ACM SIGMOD Intl. Conf. on Management of Data , Madison, Wisconsin, USA, June 2002, pgs. 181-192

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Because of the huge popularity and reach of the World Wide Web, the potential to exploit it in diverse areas such as industry, governance and academics is tremendous. Web-based applications include e-business, e-governance, medical informatics, bio-informatics, etc.

For example, Amazon [2], the well-known distributor of books and related material, enables developers, merchants, and partners to provide customers with information retrieved from Amazon's databases in real-time over the web. Amazon opened up several of its features like the catalog, shopping cart, and personalization engine through their *web services* platform to the public. More than a million associates can now access Amazon's product listings through the web and provide consumers with value-added services, while also uploading their own products to be advertised and sold on Amazon's web-site [41].

For applications such as the above to succeed, there are several requirements, including: (i) a *common* format for publishing the data should be agreed upon by the concerned parties, (ii) applications should be able to *automatically* consume this data, (iii) humans should be able to read the data so that they can design customized applications, and (iv) the publishing format has to be platform- and vendor-independent.

The exchange and manipulation of documents conforming to a common format has been in practice for over 40 years. For example, IBM developed GML (Generalized Markup Language) so that the *same* markup for, say, a manual, could be used to produce a book, electronic editions, reports, etc. The need for *standardizing* the way in which

markup could be specified, defined and used in documents was recognized when wider varieties of *document types* had to be specified, each with their own set of tags and structure. The result was the development of *SGML (Standard Generalized Markup Language)*, published as ISO 8879 [33]. SGML can be used for defining and using *portable* document formats and can handle complex documents. SGML supports the following important features: (i) extensibility – the ability to add new tags, (ii) structure – the flexibility to specify deeply nested structures, possibly containing missing and repeated elements, and (iii) validation – enabling the application which consumes the SGML document to check whether the document is *valid* with respect to its document type.

When the web was still in its nascent stages, electronic documents were published on the web mainly to be read by web browsers and this comparatively simple application did not require the power of SGML. And so was born the *Hypertext Markup Language (HTML)* [31]. HTML is also an application of SGML and defines its own specific tagset. It is hugely popular because it is easy to learn and use.

However, HTML's tagset is mainly limited to describing how the document should be *displayed* on a browser, rather than describing its *content*. And this is the main reason why HTML cannot be used for applications such as those supported by Amazon. The markup and structures that need to be defined for these applications is industry-specific and can be supported not by HTML, but by SGML. And so, the need to bring SGML to the web resulted in the development of **XML** (eXtensible Markup Language), a meta-language which is based on SGML, but without some of the more difficult-to-learn and difficult-to-implement features of SGML.

XML [23] is a highly flexible text format that has become the defacto standard for electronic publishing and data exchange on the web. Its simplicity makes it a tool of choice in various applications requiring *data representation*, *integration* and *transformation* (see for example, [77], for a long list of such applications). Moreover, the emergence of supporting XML-related standards such as schemas for describing classes of XML documents and query languages for accessing and transforming XML data has made XML a powerful tool, making it the key component in applications such as those supported by

the web-services platform of Amazon.

1.1 XML and Related Standards

XML emerged as a standard in 1998, and is a recommendation of the World Wide Web Consortium (W3C) [73]. Figure 1.1 shows a fragment of data from the Database and Logic Programming Bibliography website (DBLP) [18]. The data contains elements (tags) such as `article` and `author`, as well as attributes, such as `key`. All these tags are specific to describing the DBLP data. The data representation is flexible since it allows for *optional* elements (for example, one of the `article`s has a `url`, while the other does not), and repeated elements (multiple `author` elements in one `inproceedings`, while the other contains a single `author`), etc. Different elements in the document can be *linked* to each other through the use of IDs (keys) and IDREFs (keyrefs). For example, the `key` attribute in each of the elements acts as the *identification* or *key* of that element. This key can then be referred to by a *key reference* such as `crossref`.

Each start tag has a corresponding end tag, making this XML fragment *well-formed*. The nested structure of the document can be regarded as a *tree*, with the internal nodes containing the elements and attributes and the leaf nodes containing the *values*, as shown in Figure 1.2[1]. This representation is commonly used when manipulating XML data, especially from APIs such as the Document Object Model (DOM) API [20].

Other standards essential for the effective use of XML such as schemas and query languages are being developed, and some of them have become recommendations. Chief among the schema languages proposed for XML are the DTD (Document Type Definition) [23] and XML Schema [71]. Both these languages make use of *regular expressions* to provide flexibility in describing a class of documents. XML Schemas are more powerful than DTDs since they have a *type system* associated with them and allow the *decoupling* of tag names from type names. A snippet of the XML Schema which describes the DBLP data is shown in Figure 1.3. The flexibility of regular expressions is suitable for expressing

[1]Note that we may also choose to represent key/keyrefs as edges, making the data graph-structured.

```
<dblp>

  <mastersthesis mdate="2002-01-03" key="ms/Brown92">
    <author>Kurt P. Brown</author>
    <title>PRPL: A Database Workload Specification Language, v1.3.</title>
    <year>1992</year>
    <school>Univ. of Wisconsin-Madison</school>
  </mastersthesis>

  <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
    <ee>db/labs/dec/SRC1997-018.html</ee>
    <ee>http://www.mcjones.org/System_R/SQL_Reunion_95/</ee>
    <editor>Paul R. McJones</editor>
    <title>The 1995 SQL Reunion: People, Project, and Politics, May 29, 1995.</title>
    <journal>Digital System Research Center Report</journal>
    <volume>SRC1997-018</volume>
    <year>1997</year>
    <cdrom>decTR/src1997-018.pdf</cdrom>
  </article>

  <article mdate="2002-01-03" key="tr/gte/TR-0263-08-94-165">
    <ee>db/labs/gte/TR-0263-08-94-165.html</ee>
    <author>Frank Manola</author>
    <title>An Evaluation of Object-Oriented DBMS Developments: 1994 Edition.</title>
    <journal>GTE Laboratories Incorporated</journal>
    <volume>TR-0263-08-94-165</volume>
    <month>August</month>
    <year>1994</year>
    <url>db/labs/gte/index.html#TR-0263-08-94-165</url>
    <cdrom>GTE/MANO94a.pdf</cdrom>
  </article>

  <inproceedings mdate="2002-01-23" key="conf/b/Sekerinski98">
    <author>Emil Sekerinski</author>
    <booktitle>Graphical Design of Reactive Systems.</booktitle>
    <crossref>conf/b/1998</crossref>
    <ee>http://link.springer.de/link/service/series/0558/bibs/1393/13930182.htm</ee>
    <pages>182-197</pages>
    <url>db/conf/b/b1998.html#Sekerinski98</url>
    <year>1998</year>
  </inproceedings>

  <inproceedings mdate="2002-01-23" key="conf/b/BehmBM98">
    <author>Patrick Behm</author>
    <author>Lilian Burdy</author>
    <author>Jean-Marc Meynadier</author>
    <booktitle>Well Defined B.</booktitle>
    <crossref>conf/b/1998</crossref>
    <ee>http://link.springer.de/link/service/series/0558/bibs/1393/13930029.htm</ee>
    <pages>29-45</pages>
    <url>db/conf/b/b1998.html#BehmBM98</url>
    <year>1998</year>
  </inproceedings>

</dblp>
```

Figure 1.1: Sample DBLP Data

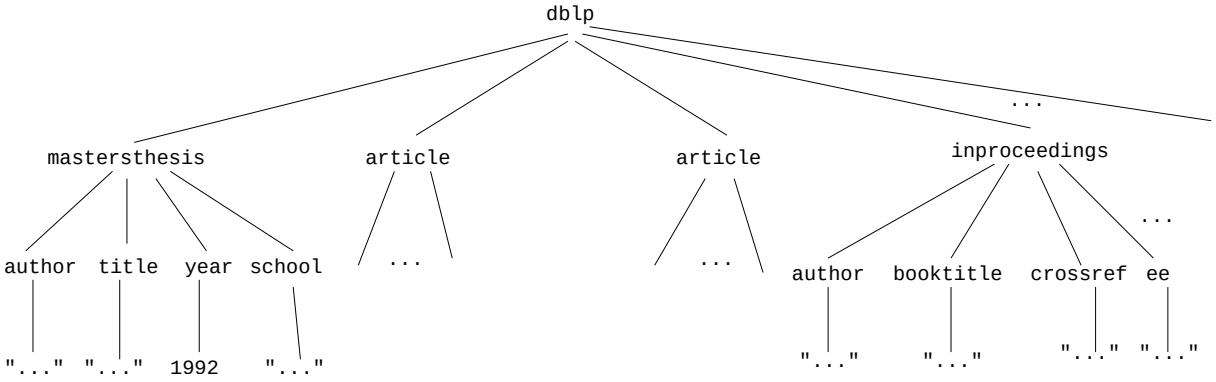


Figure 1.2: Tree Representation of the DBLP Data

features such as optional and repeated elements. For example, the type Article contains several *optional* elements, such as Url (the `?` qualifier in Figure 1.3 indicates this) – the second `article` in the data contains a `url` while the first one does not. The XML Schema also has types which may occur *multiple* times, such as Author (indicated by the `*` qualifier) – the first `inproceedings` in the XML data contains a single `author` element while the second one contains three such elements. Moreover, *simple data types* such as *integer* for the type Year and *string* for the type Author can be specified.

Query languages for XML include the declarative XPath [79] and XQuery [70]. XQuery is a language with the power of first order predicate calculus. It can query XML data, and additionally provides constructs to transform and integrate multiple sources of XML data. Extensions for enabling XQuery to support updates are currently under development [80]. XPath, also a full-fledged query language on its own, forms the core of XQuery and mainly provides the navigational constructs required to query XML. The following XQuery query extracts all articles published before 1995 and returns the key, title and authors of each article in a new format.

```
for $i in /dblp/article
where $i/year < ‘‘1995’’
return
  <ARTICLE>
    <KEY> $i/@key </KEY>
    <TITLE> $i/title </TITLE>
    <AUTHORS>
       <AUTHOR> $i/author </AUTHOR>
    </AUTHORS>
```

```
define element dblp {
    type Mastersthesis+, type Article+, type Inproceedings+
}
define type Mastersthesis {
    element mastersthesis {
        attribute Mdate, attribute Key, type Author,
        type Title, type Year, type School
} }
define type Article {
    element article {
        attribute Mdate, attribute Key, type Ee*, type Author*,
        type Editor*, type Title, type Journal, type Volume,
        type Month?, type Year, type Publisher?, type Url?,
        type Cdrom?
} }
define type Inproceedings {
    element inproceedings {
        attribute Key, attribute Mdate, type Author*, type Booktitle,
        type Cdrom?,type Crossref*, type Ee*,
        type Number?, type Pages*, type Url*,
        type Year
} }
define type Crossref { element crossref {xsd:string } }
define type Author { element author {xsd:string } }
define type Booktitle { element booktitle {xsd:string } }
define type Year { element year {xsd:integer } }
```

Figure 1.3: Snippet of an XML Schema for DBLP

```
</ARTICLE>
```

This XQuery contains three clauses: `for`, `where` and `return`. The `return` clause constructs a new fragment of XML. Note that *path expressions* such as `/dblp/article` are an essential part of XQuery. This expression indicates that starting at the root (denoted by the first "/" symbol), the tag `dblp` has to be matched. From the nodes which match `dblp`, the *children* of those nodes (denoted by the second"/" symbol) have to be traversed. The "/" symbol denotes the *navigational axis*. Several other axes, such as *descendant* (denoted by "//"), *parent*, *ancestor*, *left-sibling*, etc. can be specified. The path expression can also contain *branches*, as in the following example: `/dblp/article[year < ‘‘1995’’]` which asks for all `articles` whose publication `year` is less than 1995. Such *branching path expressions* with *value* as well as *boolean* predicates (including *and*, *or* and *not*) form the crux of XPath.

1.2 Challenges in XML Data Management

The proliferation of the web and the emergence of XML as a popular means of data representation and data exchange has resulted in the need for managing large volumes of XML data. The primary data management issues which need to be addressed include:

Storage: XML data can be stored in a variety of ways including backends such as file systems, relational systems or native XML stores. The choice of backend depends on the application. Building a full-fledged XML storage manager is often an attractive option since the layout can be optimized for efficient query processing. But, many applications expect XML data to be either stored in or generated from already existing relational systems. Hence the storage of XML in relational database is an interesting alternative that has attracted considerable attention from the research community. In addition to storage in such legacy systems, the complementary problem of *XML publishing*, where XML data is produced from relational systems is also important.

Query processing: Access to XML involves the retrieval, transformation and update of XML data using XML query languages such as XPath and XQuery. Query processing as a whole involves the investigation of several other issues. Chief among them are: (i) development of efficient algorithms to process the various query predicates, including updates; (ii) development of cost models for XML query processing; (iii) cardinality estimation of query fragments which serve as inputs to the query optimizer; (iv) building query optimizers to generate optimal *query plans*, etc.

Except for XML publishing which is a problem mainly motivated from XML applications which need to publish data from existing sources, the other data management issues of storage and query processing have already been addressed in the context of relational database systems. However, direct application of techniques developed for relational systems to XML data management is not always possible due to the *fundamental* mismatch in the data models of XML and relational data. We list several differences below between XML and relational data which makes clear the necessity for revisiting several data management issues in the context of XML.

Data: XML data is tree-structured (if keys and keyrefs are treated as edges, then XML is graph-structured), while relational data is flat and is made up of tables and columns. Moreover, while relational systems are predicated on the existence of a schema, XML data *need not* be accompanied by a schema.

Schema: Relational databases have a rigid schema associated with them, which clearly defines the tables, their columns and the data types of these columns. However, XML schemas are significantly more flexible since they describe *classes of documents* through means of *regular expressions*, which allow for a considerable variety of data conforming to the same schema.

Queries: Relational databases are queried through the declarative query language SQL which specifies the tables and columns from which data is to be retrieved and how they are to be combined. On the other hand, since XML is tree structured, XML query languages such as XPath and XQuery have *navigational* primitives in addition

to *value* primitives. Also, the output of an XML query could be an arbitrary sized tree.

1.3 Problems Addressed in the Thesis

In this thesis, we address two important problems that go to the heart of XML data management due to their substantive impact on efficient XML query processing: *XML data summarization* and *XML data storage*. Data summarization provides crucial cardinality estimation inputs to the query optimizer which helps in execution plan generation, while data storage directly impacts the query processing. The work presented in this thesis is divided mainly into three parts: (i) *Statistics Collection and Cardinality Estimation*, (ii) *Statistics Maintenance*, and (iii) *XML Storage*. Our solutions to each of these problems is based on the existence of an XML Schema that decides document validity. As mentioned previously, XML Schemas are more powerful than DTDs and are now becoming commonplace for most XML applications, and are widely used (see [48] for descriptions of several applications and standardized XML Schemas). Another crucial feature in our work is the use of *schema transformations* on the XML Schema to make our solutions both effective and efficient.

1.3.1 Statistics Collection and Cardinality Estimation

A critical component of an XML data management system is the *result estimator*, which estimates the selectivity of user queries. Its importance arises from the fact that estimated cardinalities serve as inputs in many aspects of XML data management systems such as cost-based storage design and query optimization.

A large body of literature is available for result size estimators in traditional database systems. In essence, summary statistics such as the distribution of values in a given column, the minimum and maximum value, the number of distinct values, etc. are stored and used in an *estimation framework* in order to estimate the result size. However, in the XML domain, the design of such result estimators becomes more complex because of the

fact that XML inherently has *structure* associated with it and does not consist merely of values as in the case of relational systems. Moreover, the query languages designed for XML have *tree navigation* as a first class primitive. Additionally, since XML Schema provides regular expression constructs, the structure of the data has a considerable amount of flexibility in terms of presence/absence of certain elements, number of occurrences of a given element, etc. This may give rise to skew in the structure of the data as well as the values. In effect, any solution to the problem of cardinality estimation for XML has to provide solutions which deal with both structure and value on an equal footing.

The first part of the thesis proposes **StatiX** (**Stati**stics for **X**ML), a framework for XQuery cardinality estimation and statistics collection for XML data. StatiX is a novel, schema-based framework that exploits the structure in the XML Schema in order to decide what statistics should be collected. StatiX proposes the use of *schema transformations* in order to vary the granularity of the statistics collected (in consequence, the accuracy of the cardinality estimation is affected). It is based on the following two principles: i) the use of standard XML technology (mainly, schema validators), where possible, in order to collect the statistics efficiently, and ii) the use of histograms to summarize both the structure as well as the values in an XML document. This aspect of StatiX enables the reuse of histogram multiplication techniques in order to estimate the query cardinalities of various queries. Currently, StatiX can handle a significant subset of XQuery, specifically, branching path expressions with value predicates. Through a detailed performance evaluation, StatiX is shown to be accurate as well as concise for different synthetic as well as real data sets.

1.3.2 Statistics Maintenance

A large number of XML applications are *dynamic* and frequently *update* the underlying data. For example, an XML workflow application that keeps track of customer purchase orders may dynamically update book-keeping information about the status of the order as it navigates through the order-processing cycle. While StatiX provides solutions to the problem of *statistics production*, it is equally important to address the problem of *statistics*

maintenance when the underlying data is subject to inserts, deletes or modifications.

Periodically recomputing the statistics from scratch on the updated documents is an obvious choice to cater to the XML update problem. But since recomputation requires *the whole document to be parsed*, it can be prohibitively expensive [47] if recomputations occur frequently, especially for large documents. Further, if recomputations are not adequately timed, stale statistical summaries may lead to unacceptable estimation errors. And so, techniques which can *maintain* the statistics in parallel with the receipt of data need to be considered.

Incremental maintenance of data statistics per se is not a new issue to the database community, having been previously addressed in the context of relational database systems (see *e.g.,* [30]). However, what is novel in the XML context is that statistics about both *structure* and *value* have to be maintained. That is, while in an RDBMS, there is *no* difference, as far as the statistics go, between the insertion of a tuple in the middle of a relation or the appending of the same tuple at the end, the *location of the update* is always an issue in XML. Secondly, the *size of the update* in an RDBMS can only be either a single tuple or a set of tuples. But, in an XML environment, the update could be an arbitrarily complex XML fragment, or sets of fragments. For example, the update could require inserting sub-trees at various locations in the original document. Thus, maintaining accurate statistics for XML databases poses a fresh set of problems as compared to those tackled in prior systems.

In the second part of the thesis, we propose an algorithm for incremental maintenance of XML Statistics, named **IMAX** (**I**ncremental **Ma**intenance of **X**ML Statistics). IMAX proposes efficient strategies for incrementally maintaining StatiX summaries as and when updates are applied to the data. IMAX addresses the issues of: i) estimating how many updates will take place and, ii) estimating the specific locations where the updates will take place. The second issue is specific to XML since the *order* of elements has to be taken into account.

1.3.3 Relational Storage for XML

As applications manipulate an increasing amount of XML, there is a growing interest in storing XML data in relational databases. The main advantage of storing XML in relational systems is that the large amount of research which has enabled relational systems to grow into a mature technology can be leveraged. Basic data management services such as concurrency control and recovery are already present in relational systems and need not be re-invented for XML. In addition, many XML applications are expected to be produced from or stored in relational databases. The need for such close interaction with relational data make relational database systems an attractive choice for storing XML. An added advantage is that relational systems are already widely used.

Due to the mismatch between the complexity of XML's tree structure and the simplicity of flat relational tables, there are many ways to store the same document in an RDBMS, and a number of *heuristic* techniques have been proposed. However, a single fixed mapping is unlikely to work well for all different applications. The case for a *cost based* approach was made in [6, 7, 8] wherein the *application characteristics* in the form of the XML Schema, XML data statistics and XML query workload were considered when coming up with a relational configuration. The system, named *LegoDB*, used schema transformations to generate different relational configurations and used a *relational optimizer* [61] to estimate the costs of the generated candidate configurations.

The third part of the thesis – named **FleXMap** (**Fle**xible **X**ML **Map**pings) – builds on LegoDB and investigates the utility of the schema transformations in more detail. Many of these schema transformations are also used in StatiX to improve the granularity of the summary. Issues which arise in the context of cost-based storage, such as the propagation of statistics from one relational configuration to another during the search process are addressed. Using FleXMap has two advantages: i) a much larger space of configurations can be explored by the addition of new transformations or by choosing an appropriate subset of the transformations already proposed, and, ii) because of the way in which the search algorithms are derived from the framework, a larger space is searched at a marginal increase in response time. Several optimizations to substantially improve the

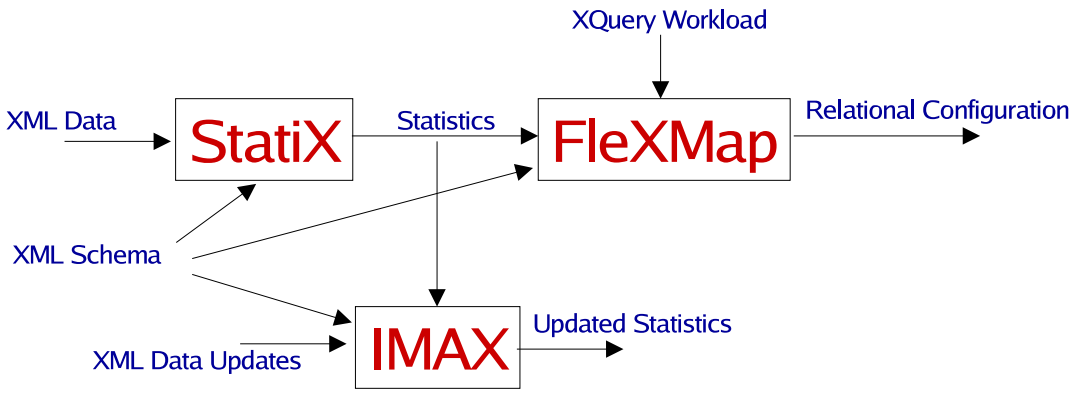


Figure 1.4: The Big Picture: Problems addressed in the Thesis

efficiency of the search process while maintaining the quality of the output are proposed.

1.3.4 Summary of Contributions

In summary, we propose a set of three tools in this thesis – StatiX, IMAX and FleXMap – which address the issues of statistics production, statistics maintenance, and storage of XML data, respectively. These three systems are schematically shown in Figure 1.4. StatiX takes an XML Schema and XML data as inputs, and outputs a statistics summary. This summary can then be used to estimate the cardinality of various XQuery queries. IMAX takes as input the XML Schema and the currently existing StatiX summary for a data set. IMAX tracks the updates to the underlying data and appropriately updates the summary. FleXMap takes as input an XML Schema, a StatiX summary and a query workload, and outputs a relational configuration that can answer the queries in the query workload efficiently. Experimental evaluation of all three tools over real and synthetic data sets shows that this toolkit is both effective as well as efficient.

1.4 Organization

The rest of this thesis is organized as follows. In Chapter 2, we review related work on statistics and storage of XML and highlight the differences and similarities with the work presented in this thesis. In Chapter 3, we introduce *schema transformations* which are

the basis of the solutions presented. In Chapter 4, we present **StatiX**, a framework for XML statistics production and cardinality estimation. In Chapter 5, we present **IMAX**, a technique for maintaining XML statistics in the presence of updates to the underlying data. In Chapter 6, we present **FleXMap**, a cost-based search system to find efficient relational configurations for XML data. We conclude and identify avenues of future research in Chapter 7.

Chapter 2

Related Work

2.1 Introduction

There have been several proposals on XML query selectivity estimation in the literature [1, 14, 26, 27, 40, 54, 55, 56, 62, 63, 75]. They differ in many different aspects, such as use of schema information, summary structure, supported queries, etc. However, a common limitation of the proposals mentioned above is that they do not support *statistics maintenance*. In addition to IMAX, the *Bloom Histogram* technique proposed in [72] is the only other work we are aware of that deals with the problem of maintaining statistics in the presence of updates to the underlying XML data.

Moving on to the problem of storing XML data, there is a large amount of literature available on primarily two ways of storing XML data: (i) building a native XML database system, and (ii) storing XML in an already existing database system such as a relational database system. While there are many native XML databases available (for example, [34, 46, 66], see [78] for a list of such databases), our focus in this thesis is on storing XML data in relational systems.

There are several proposals for storing XML in relational systems. For example, [8, 12, 13, 17, 19, 21, 25, 35, 38, 43, 50, 52, 64, 68, 81, 82]. These proposals differ in many ways – whether they are order-preserving and constraint-preserving, whether they are schema-aware or schema-oblivious, whether they *automatically* generate a relational

mapping, whether they are *heuristic* or *cost-based*, etc.

In this Chapter, we survey the related work on XML statistics in Section 2.2 – statistics production as well as maintenance. And then in Section 2.3, we discuss related work on mapping XML to relational systems.

2.2 XML Statistics

Proposal	**Input**	**Summary Structure**	**Order**	**Structure**	**Value**	**Updates**
[14]	Data	Correlated sub-path tree	No	Tree pattern	No	No
[1]	Data	Path tree and Markov tables	No	Simple paths	No	No
XPathLearner [40]	Query feedback	Markov tables	No	Simple Paths	Yes	No
[75]	Data and schema	Position Histogram	Yes	Twigs	Yes	No
XSketches [54, 55, 56]	Data	XSketch graph	No	Tree Pattern	Yes	No
StatiX [26, 27]	Data and Schema	Histogram	Yes	Tree Pattern	Yes	No
[62, 63]	Data and Schema	Tagged region graph	No	Tree pattern	Yes	No
Bloom Histograms [72]	Data	Bloom Filter	No	Simple Paths	No	Yes
IMAX [60]	Data and Schema	Histogram	Yes	Tree pattern	Yes	Yes

Table 2.1: Summary of Related Work on Statistics Production and Maintenance

Table 2.1 summarizes the differences and features among the various proposals for

XML statistics production and maintenance.

2.2.1 Statistics Production

In [14], Chen et al propose a scheme that captures *correlations between paths*, resulting in accurate estimation of *twig* queries. Twig queries are queries which can be expressed as small, node-labeled trees that match portions of the data. Their strategy consists of gathering counts for frequently occurring twiglets in the data, and then assigning each twiglet a "set hash" signature that captures the correlations between the subpaths in the data tree. Query selectivity is then estimated by combining the hash signatures of the twiglets occurring in the query.

In [1], Aboulnaga et al propose two techniques for estimating selectivity of simple path expressions: summarized *path trees* and summarized *Markov tables*. A summarized path tree is a representation of all paths in the XML document. At each node in the path tree, the frequency of paths from the root to that node is maintained. The frequency at the node is the selectivity of the path from the root to that node. Since the path tree can be very large, several methods of compressing it are proposed. The main idea is to delete or coalesce low frequency nodes. Markov tables are tables which contain the frequency information for all simple paths upto a given length k. The low frequency paths are again deleted to make the table compact. The selectivity of any simple path query is computed based on the selectivities of "subpaths" of the query – in effect, it is assumed that the selectivity of any tag in the path depends *only* on at most the previous $k-1$ tags, in effect modeling the path as a Markov process of order $k-1$ (hence the name "Markov" table).

The XPathLearner [40] technique proposes the use of *query feedback* in order to collect statistics about simple path expressions. The basic summary structure is the *Markov histogram*, which simply summarizes the counts of simple paths with lengths less than some parameter k, as well as values. This summarization is done *online* by observing the selectivities of queries fired on the database, rather than by scanning the data offline. The principle of computing the selectivities of simple path expressions is the same as that

described in [1].

In [75], the authors propose "position histograms" to capture selectivity information. Each node in the XML tree is first labeled with intervals – that is, each node is assigned a tuple of a start position and an end position. Any descendant node has an interval strictly contained in any of its ancestor nodes. Next, several predicates of interest are identified – a predicate could be as simple as "element name = AUTHOR". For each such predicate, a position histogram is built. A position histogram is a two dimensional histogram which contains the start values in the x-axis and end values in the y-axis. Armed with these position histograms, predicates such as $P_1//P_2$ can be computed. For branching patterns, the query is first decomposed into simple paths and the results are combined. Certain properties of position histograms, such as identifying certain regions to be empty, help in effective evaluation of these predicates. Schema information is made use of when available – for example, if it is known that two predicates P_1 and P_2 reside on different branches of the tree, then the selectivity of $P_1//P_2$ is 0 and can be reported without any computation.

Polyzotis et al [54, 55, 56] propose graph synopses structures called "XSketch"es as the summary structure. XSketches are based on the notion of backward and forward bisimilarity of a graph. The two extreme XSketches are the "label-split" graph (the coarsest summary) and the "backward-and-forward bisimilar" graph (BF-bisimilar graph, the most detailed summary). Construction for the BF-bisimlar graph is known in the literature [51]. For values, they propose building value histograms at the leaf level. Since the label-split graph is small but too inaccurate and the BF-bisimilar graph is highly accurate, but potentially too large, a greedy algorithm is proposed to find an appropriate "intermediate" XSketch synopsis. The construction algorithm outlined starts from the label-split graph and successively *refines* edges to make them backward or forward (or both) stable based on the increase in accuracy that they offer on a given query workload. XSketches are able to handle branching path expressions with value predicates as well as twig queries which require combining results from multiple path expressions. The cardinality estimation makes use of independence and uniform distribution assumptions when necessary. XSketches come closest in spirit to StatiX even though they do not use

any schema information and do not store parent-child distributions (only the cardinality is stored). The refinement operations proposed in XSketch are similar to the schema transformations that we propose for StatiX.

In [62, 63] the author proposes a "metamodel" for estimating the cardinality of XML queries which may include the "for" and "let" bindings. The framework is based on the notion of regions of data where each region contains the cardinality of a set of nodes sharing a common feature (such as tags or types). A "match occurence" is a sequence of regions which satisfies a certain structural or value predicate. Given a set of bind variables in the *for* or *let* bindings, a list of match occurences is computed for each variable. Based on common parent or ancestor regions, correlations are determined and the cardinality divided appropriately. The framework can identify not only twig estimates, but also group cardinalities which occur when *let* bindings are used. As the author states, StatiX can be a part of this model if the region can be defined appropriately (for example, the node type and its bucket in the structural histogram can constitute a region for a given node in the data).

In summary, the above proposals outline novel approaches to solve the selectivity estimation problem. They vary in the summary structure they use and the type of queries they can support (simple path expressions, branching path expressions, twigs, etc.). Most of them are data-based techniques and do not exploit the XML Schema to optimize their summaries. This is in contrast to StatiX which not only collects statistics based on the schema, but also utilizes schema information to substantially reduce the size of the summary. Moreover, another significant advantage in StatiX is the use of histograms (in addition to the XML Schema) as the summary structure of choice to capture parent-child and value distributions and the estimation is based on histogram multiplication techniques. This aspect may make the techniques proposed for StatiX easily adoptable in relational systems which already use histograms for capturing value distributions.

2.2.2 Statistics Maintenance

We are aware of only one other work which supports cardinality estimation in the presence of updates. In [72], the authors propose the use of *bloom histograms* as a summary structure. Given a table of paths and their frequencies, a bloom histogram summarizes them into buckets. Paths with similar frequencies are grouped into a single bucket and their average count is maintained. In order to represent all the paths in a given bucket, a bloom filter is utilized. A bloom filter is a data structure which can be used to represent sets and supports set membership queries. In order to maintain the bloom histogram, an intermediate "dynamic summary" is maintained which is used to *periodically* recompute the bloom histogram. IMAX differs from bloom histograms in several ways: (i) Bloom histograms need to be periodically recomputed, and as a consequence, may suffer from highly inaccurate estimates if the periodicity of recomputations is not properly set (the authors do not explicitly comment on the periodicity of recomputations), while IMAX supports *incremental* maintenance – that is, the summary is updated *as and when* the updates are received – and recomputations are used only as a backup mechanism, (ii) the recomputations on the Bloom histogram happen over an *intermediate* data structure, such as the path tree table, while the recomputations in IMAX happen over the backend data – hence the recomputations in the bloom histogram technique may be much cheaper, but *maintaining* the intermediate structure requires a parsing of the data into paths, (iii) the class of queries covered is restricted to simple path expressions without values in the bloom histograms while IMAX supports the full functionality of StatiX, and (iv) StatiX and consequently, IMAX, utilize schema information to build and maintain the summary while bloom histograms make use of only the data.

2.3 XML Storage

Table 2.3 gives an overview of features of various techniques to store XML in relational systems. Related work that doesn't fit into the above table (for example, there are papers which theoretically analyze the nature of XML-to-relational mappings) are described in

Techniques	Schema-aware or oblivious SA/SO	Cost-based	Constraint-preserving	Order-preserving	Automatic or Manual (A/M)
STORED [19]	SO	No	No	Yes	A
Edge [25]	SO	No	No	Yes	A
XRel [81]	SO	No	No	Yes	A
[68]	SA	No	No	Yes	A
[64]	SO	No	No	No	A
XParent [35, 36]	SO	No	No	Yes	A
[65]	SA	No	No	No	A
[12, 16, 11]	SA	No	Yes	No	A
[38],[42]	SA	No	No	No	A
LegoDB and FleXMap [8, 59, 7, 6]	SA	Yes	No	No	A
[82]	SA	Yes	No	No	A
[13]	SA	Yes	No	No	A
ShreX [21, 22]	SA	No	No	Yes	M
ELIXIR [52]	SA	Yes	Yes	No	A
Oracle XML DB [50]	SA/SO	No	No	Yes	A/M
DB2 XML Extender [17]	SA/SO	No	No	Yes	A/M
MS SQL Server [43]	SA/SO	No	No	Yes	A/M

Table 2.2: Summary of Related Work on Storage in Relational Backends

the text.

2.3.1 Storage Methods for Schemaless XML Data

In STORED [19], a mapping between a semi-structured database instance and a relational schema is *automatically* chosen and expressed in a declarative language called STORED. Data mining techniques are utilized to generate this mapping. Parts of the data which do not fit into the schema are stored in an overflow graph.

In [24, 25], several mapping schemes are proposed. According to the Edge 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. The tuple consists of the source and destination ids of the nodes, the label, and the value (if the destination is a leaf node) in addition to the node's ordinal. In a variant, known as the Attribute approach, the edge table is horizontally partitioned on the tag name yielding a separate table for each element/attribute. Two other alternatives, the Universal table approach, corresponding to an outer-join of all the tables in the attribute approach, and the Normalized Universal approach, where multi-valued attributes in the Universal Table approach are stored separately, are also proposed.

The binary association approach [64] is a path-based approach. An association is a relationship between two nodes, such as node-node (denoting a parent-child), node-value and node-attribute value, and so on. All associations between two types of nodes based on their path from the root are stored in a single relation which is named for that path. Evaluation of path expressions now involves joins among the corresponding tables.

The XRel approach [81] is another path-based approach. It constructs a single table for each node type (one each for elements, attributes and text, respectively), and one table to store all the paths that occur in the document along with a path id. In the element, attribute and text tables, the region of the node as well as its ordinal and path id are stored. The region of a node is an interval consisting of a start and end position, determined by the pre-order and post-order traversal of the document tree. The advantage of this approach is that a path expression can be easily evaluated by comparing path ids.

XParent [35, 36] uses the edge mapping scheme and stores XML documents in four separate tables – LabelPath, DataPath, Element and Data. LabelPath stores all paths along with a unique id and the length of the path; DataPath stores all parent-child relationships by storing pairs of node ids in each tuple; Element stores each element in the document by identifying it with a unique identifier and ordinal, and references the LabelPath table to indicate what its path was, and, similarly Data stores values in the document along with a unique id and ordinal, and references the LabelPath table to identify its path.

In summary, except for STORED, each of the other schema-oblivious storage schemes

provide *generic* techniques to store XML in relational systems. That is, both the markup (elements) as well as values are treated as *data* and the techniques proposed can, in general, be adapted to store any labeled graph-structured data. This is an extremely useful feature for schemaless XML data. STORED, however, *infers* a schema for the XML data and can come up with a more space-efficient storage as compared to the other techniques. But, the main drawback of all these methods is that, they are all *heuristic-based* and do not take into account the query workload while constructing the relational schema.

2.3.2 Storage Methods using XML Schemas

In [65], a DTD is used to map XML into a relational schema. Several simplifying, but lossy transformations (that is, the transformed DTD may validate a superset of documents as compared to the original DTD) are used on the regular expressions in the DTD to make it more amenable to relational storage. Three different inlining techniques – *basic*, *shared*, and *hybrid* inlining are described. Each technique differs in the way it chooses the elements to be inlined. While basic inlining creates a separate table for every element in the DTD, the shared inlining technique ensures that a given element is represented in exactly one relation. The hybrid inlining technique, which is similar to shared inlining, additionally inlines elements which are shared, but not repeating or recursive.

Methods to store and retrieve ordered XML are studied in [68]. Three order-encoding methods – Global order, Local order and Dewey order – are described and evaluated. Both the schemaless and schema-aware cases are considered.

In [12], the authors propose an algorithm which preserves the *key* and *keyref* constraints specified in an XML schema when XML data is stored in relations. In order to check the key and keyref constraints, it is enough to check the key and foreign key constraints specified in the relational schema derived by their algorithm.

In [16], the work in [12] is extended. Methods to refine the relational schema given a set of keys in the schema are proposed. Given a universal relation corresponding to the XML schema and a set of keys, the authors propose methods to determine the minimum

number of functional dependencies that must hold in the relational schema (that is, the minimum cover of all functional dependencies). This helps in appropriately decomposing the universal relation such that the XML keys are correctly propagated.

Redundancy reducing XML storage is considered in [11]. The techniques outlined make use of semantic constraints specified in the XML schema. For example, if there are value based keys for a particular element, there is no need to generate id values for that element.

The theory of regular tree grammars is used in [42] to convert XML schemas to relational schemas. A normal form of representation for XML schemas which eliminates the use of the union (|) operator is defined. Using this normal form as the basis, the authors outline a language independent representation of several features of XML schema including basic data types and IDREFs. Several simplifying regular expressions are utilized when the XML schema constraints (such as *order* of elements in (A,B,A)*) cannot be captured in the relational domain.

In [38], the authors propose a method to convert XML DTDs to relational schemas where semantic constraints implicit in the DTD are translated to the relational schema via inclusion dependencies. For example, if papers are nested elements under a conference in the DTD, then each tuple in the table for papers should reference a tuple in the table for conference and this should be a foreign key relation. The authors propose a constraint preserving mapping algorithm to map such constraints.

A generic mapping tool called ShreX is proposed in [21, 22]. ShreX can incorporate many different mapping schemes proposed in the literature, including Edge [25], order-preserving [68], and many from [8]. The input to ShreX is an annotated XML Schema that contains details about how the mapping to the relational backend should take place, and the XML document which is to be shredded. ShreX checks the validity of the mappings and automatically shreds the document to store it in the appropriate relations. It also provides APIs to query information about the mappings, and this information can then be used for query translation.

In [3, 4], "information-preserving" mappings are defined. That is, mappings which

allow: (i) every XML query over the document to be mapped to a query over the database and, (ii) only valid updates (updates resulting in valid documents). The paper shows that existing techniques do not always preserve information and proposes an algorithm to derive relational configurations which are information-preserving.

In [69], the performance of various techniques of storing XML, including three techniques of storing XML in RDBMSs are compared. The strategies evaluated are the heuristic approach from [65] and the edge as well as the attribute approach from [25]. The authors report that [65] resulted in a much more compact data representation than either the edge or the attribute approach. Moreover, given a path expression, a large number of joins were required in the SQL query when the edge approach was chosen, making this approach sensitive to the complexity of the path expression. But, breaking up the edge table in the attribute approach contributed to a considerable reduction in the number of joins and was therefore more efficient.

In summary, as with the schema-oblivious techniques described in the previous section, the schema-aware techniques reviewed above are all *heuristic*, and do not consider a space of several possible relational mappings to choose the optimal one. However, many of them address specific XML issues such as constraint-aware mappings [16], order-preserving mappings [68], information-preserving mappings [4], etc. Not all these issues can be easily addressed in a cost-based context and need to be studied further.

The problem of converting XML schemas into relations has been formally studied in [37]. The authors specifically concentrate on the inter-relationships in the XML to relational schema translation algorithm (decomposition), the XQuery to SQL translation algorithm (query translation), and the optimality of the generated relational schema with respect to a few simple cost metrics. They show that the choice of metric along with the translation algorithms has a big impact on the quality of the final relational configuration and that practical XML-to-relational conversion algorithms should not consider the decomposition problem in isolation. FleXMap currently provides a simple query translation algorithm. However, because of its modular design, any other translation algorithm can be easily plugged into the system.

2.3.3 Cost-based Solutions

LegoDB [6, 7, 8] was the first cost-based solution described for the problem of storing XML in relations. FleXMap is built on top of the LegoDB framework with several important extensions: (i) FleXMap defines a formal framework for schema transformations (this is described separately in Chapter 3), (ii) Some of the subtleties in performing these transformations are highlighted, and consequently, different greedy algorithms are formulated to search the huge space of relational configurations, (iii) The implications of schema transformations on statistics propagation are studied and, (iv) A comprehensive experimental evaluation with different kinds of query workloads shows that the search space is often considerably reduced by a judicious choice of the greedy algorithm. In addition, several optimizations to reduce the run-time of the search process are proposed.

In [82], a hill-climbing algorithm to select a good relational configuration is proposed. Four different transformations are outlined – V-cut, V-merge, H-cut and H-merge – to be applied on an initial XML schema. The four transformations defined are a subset of transformations proposed for LegoDB and FleXMap. The main differences in their approach and ours are: (i) they use the hill-climbing algorithm while we use the greedy algorithm and, (ii) they estimate costs based on selectivity estimates for simple path expressions while we use a relational optimizer.

In [13], the authors make use of several transformations from FleXMap, but explore the impact of *physical design* on the storage efficiency. The utility of searching the combined logical and physical search space in a greedy manner is shown. The physical search space includes physical design structures such as indexes, materialized views, etc.

The ELIXIR system [52] builds on top of FleXMap to support constraints, views and triggers. ELIXIR makes use of the cost-based methodology of FleXMap, but is augmented with appropriate mapping alternatives to support the various constraints defined in the XML Schema. The goal of the work is to provide an industrial-strength cost-based system to map XML data into relations.

2.3.4 Commercial Solutions

We review three standard commercial systems which provide support for XML storage and query processing: IBM DB2's XML Extender [17], Oracle's XMLDB [49] and Microsoft's SQLXML [43].

DB2 [17] provides two options for storing XML. The first option is to store XML in a single column as a CLOB, Varchar or XMLFile (which basically stores the data in a separate file). The second option is to specify a mapping from an XML DTD to relational tables/columns using the Data Access Definition (DAD) language provided by DB2.

Oracle's XMLDB [49] provides unstructured and structured storage for XML. The unstructured storage is made possible through a special type called XMLType to store XML documents as CLOBs in a single column. Structured storage is done when an XML Schema is provided. XMLDB can *automatically* map the types in the XML Schema into relations and columns with the appropriate base type when applicable. In addition, users can control this mapping by annotating the XML Schema.

Microsoft's SQLXML [43] also provides for annotations in the XML Schema through XSD (XML Schema Definition). If no annotations are provided, then a default mapping is automatically used. In addition, XML can also be stored using the generic edge technique, or by compiling it into an internal DOM representation and then providing XPath expressions to map values into tuples.

All three database systems provide control over the mapping of XML to relational through the mechanism of schema annotations. FleXMap can be easily adapted to automatically generate such annotated XML Schemas after identifying the most efficient mapping.

In summary, there are various techniques to store XML in relational databases. They differ in several aspects, notably, (i) whether they use a schema to help in the mapping, (ii) whether they are manual or automatic and (iii) whether they are heuristic or cost-based. Research prototypes such as ShreX, as well as commercial solutions discussed above provide flexibility to the user by allowing him/her to specify the XML to relational mapping by annotating the XML schema. The schema transformations proposed in LegoDB and

FleXMap have also been used by other systems to extend the scope of cost-based search to include physical storage with views and indexes [13] as well as constraint preservation (ELIXIR [52]).

Chapter 3

Schema Transformations

3.1 Introduction

In this chapter we abstract out the essentials required for the understanding of the rest of this thesis. While all the work reported in this thesis make use of XML Schema, not all the features of XML Schema are supported or utilized. However, the primary requirement of our work is the ability of XML Schema to decouple type names from element names – a distinction not present in DTDs.

As mentioned in the Introduction, our solutions are based on the existence of an XML Schema. In addition to describing the types of documents that will be encountered, the XML Schema also allows us to perform some amount of *optimization* to the solutions, *without* having to process the data. Each of the solutions proposed in this thesis are thus made dependent on the size and complexity of the *schema*, rather than the *data*, which can be orders of magnitude larger.

We introduce some notation used in the thesis and then define and give examples of *schema transformations*, the basic building blocks for the solutions we propose. The implications of performing these schema transformations on validation are then discussed. Finally, we describe the implementation of the transformations.

```
<complex type> ::=
          <simple type>
        || <complex type> , <complex type>
        || <complex type> | <complex type>
        || <complex type> *
        || <complex type> ?
        || <tagname> [<complex type>]
<simple type> ::=
          string
        || integer
```

Figure 3.1: Using Type Constructors to Represent XML Schema Types

3.2 Basic Framework

XML Schemas are *extended context free grammars*. Two features of XML Schema that are of interest to us are the following: (i) It provides a *type system* which includes basic types such as integers, and (ii) It *decouples* the type names from the tag names.

An XML Schema can be regarded as a *complex type* represented using the type constructors for: *sequence* (","), *repetition* ("∗"), *option* ("?"), *union* ("|"), <*tagname*> (corresponding to a tag) and <*simple type*> corresponding to base types (*e.g.,* integer, string, etc.). Figure 3.1 gives a simplified grammar for the construction of types.

We make use of the following compact text notation to describe the different type constructors.

Tag Constructor: $E(label, t, n)$, where *label* is name of the tag, t is the complex type which occurs as part of the constructor, and n is the type name. Note that the type name in any of the constructors can be *null*.

Sequence, Union, Option and Repetition Constructors: Each of these constructors are defined respectively as: $C(t_1, t_2, n)$, $U(t_1, t_2, n)$, $O(t, n)$, and $R(t, a, b, n)$, respectively, where t_1, t_2 and t are complex types and n is the type name. For the repetition constructor, a and b denote the minimum and maximum occurrences of the type.

Simple Type Constructor: Simple types are represented as $S(base, n)$ where *base* is the type of the simple type (*e.g.,* integer) and n is its name.

As a simple example, consider the following fragment of XML Schema (expressed in the XQuery type syntax notation) of the IMDB (Internet Movie Database) [32] website.

```
define element IMDB { type Show* }
define type Show { element Show { type Title, type Year } }
define type Title { element TITLE { xsd:string } }
define type Year { element YEAR { xsd:integer } }
```

In the compact notation, the above fragment can be represented as:

$$
\begin{aligned}
&E(\texttt{IMDB}, t_1, null)\\
&t_1 := R(t_2, 0, unlimited, null)\\
&t_2 := E(\texttt{SHOW}, t_3, \mathsf{Show})\\
&t_3 := C(E(\texttt{TITLE}, S(string, null), \mathsf{Title}),\\
&\qquad E(\texttt{YEAR}, S(integer, null), \mathsf{Year}),\\
&\qquad null)
\end{aligned}
$$

3.3 Schema Transformations

We make use of two types of schema transformations: (i) manipulation of type names – this is possible since XML Schema decouples the type name from the tag name and, (ii) using equivalent regular expressions – that is, replacing a regular expression M with another expression N such that L(M) = L(N), where L(M) and L(N) denote the languages accepted by the automata of M and N, respectively. An important property satisfied by the schema transformations that we define here is that they do not alter the set of documents validated by the original schema (this is discussed in more detail in Section 3.5). We now define and give examples of the schema transformations. Several of these transformations were first outlined in [8].

3.3.1 Inline and Outline

The inline operation corresponds to *removing* the name of a type (most commonly, elements) in the schema. Conversely, the outline operation provides a name to a type (again, most commonly, elements) in the schema. More formally, the inline and outline operations can be represented as follows:

$E(label, t, n) \rightarrow Inline \rightarrow E(label, t, \mathbf{null})$

$E(label, t, \mathbf{null}) \rightarrow Outline \rightarrow E(label, t, n)$

Examples of the inline and outline operations are shown below. Consider the following fragment of schema:

```
define type Show {element SHOW
    {element TITLE {xsd:string }, element YEAR {xsd:integer }}}
```

The tag structure of this fragment has a tag `SHOW` with two children `TITLE` and `YEAR` in that order. It is possible to introduce *type names*, for both `TITLE` and `YEAR` *without* changing the tag structure – that is, *outline* both `TITLE` and `YEAR` – as follows:

```
define type Show { element SHOW { type Title, type Year } }
define type Title { element TITLE { xsd:string } }
define type Year { element YEAR { xsd:integer } }
```

Starting from the second schema which contains type names for `TITLE` and `YEAR`, it is possible to *inline* them both to get the original schema.

3.3.2 Type Split and Type Merge

When two different types T_1 and T_2 have a child T with the *same name*, T is said to be *shared* by T_1 and T_2. The *type split* operation eliminates the shared type by giving different names to the shared type. Conversely, *type merge* identifies types with the same structure, but different names, and makes them *shared* by giving them the same name. Formally, let T_1 and T_2 be two types. Then, type split is defined as follows:

case T_1, T_2 of

| $E(label, t, a)$, $E(label', t', a) \rightarrow$

IF $label = label'$ AND $t = t'$, THEN replace T_1 by $E(label, t, a_1)$

| $C(t_1, t_2, a)$, $C(t'_1, t'_2, a) \rightarrow$

IF $t_1 = t'_1$ AND $t_2 = t'_2$ THEN replace T_1 by $C(t_1, t_2, a_1)$

| $U(t_1, t_2, a)$, $U(t'_1, t'_2, a) \rightarrow$

IF $t_1 = t'_1$ AND $t_2 = t'_2$ THEN replace T_1 by $U(t_1, t_2, a_1)$

| $R(t, m, n, a)$, $R(t', m', n', a) \rightarrow$

IF $t = t'$ THEN replace T_1 by $R(t, m, n, a_1)$

| $O(t, a)$, $O(t', a) \rightarrow$

IF $t = t'$ THEN replace T_1 by $O(t, a_1)$

| $S(b, a)$, $S(b', a) \rightarrow$

IF $b = b'$ THEN replace T_1 by $S(b, a_1)$

Note that if any of the above cases violates the IF condition, then there is an error in the schema which has given the same name to two different type structures. Type merge is defined as:

case T_1, T_2 of

| $E(label, t, a)$, $E(label', t', a') \rightarrow$

IF $label = label'$ AND $t = t'$, THEN replace a' by a

| $C(t_1, t_2, a)$, $C(t'_1, t'_2, a') \rightarrow$

IF $t_1 = t'_1$ AND $t_2 = t'_2$ THEN replace a' by a

| $U(t_1, t_2, a)$, $U(t'_1, t'_2, a') \rightarrow$

IF $t_1 = t'_1$ AND $t_2 = t'_2$ THEN replace a' by a

| $R(t, m, n, a)$, $R(t', m', n'a') \rightarrow$

IF $t = t'$ THEN replace a' by a

| $O(t, a)$, $O(t', a') \rightarrow$

IF $t = t'$ THEN replace a' by a

| $S(b, a)$, $S(b', a') \rightarrow$

IF $b = b'$ THEN replace a' by a

The definition of both operations requires that we know how to determine when two types are equal. We define equality of two types as follows:

Definition 3.1 *Syntactic Equality*

Two types T_1 *and* T_2 *are syntactically equal – denoted by* $T_1 \cong T_2$ *– if the following holds:*

case T_1, T_2 *of*

$| E(label, t, a),\ E(label', t', a') \rightarrow$

$\quad label = label'$ *AND* $a = a'$ *AND* $t \cong t'$

$| C(t_1, t_2, a),\ C(t'_1, t'_2, a') \rightarrow$

$\quad a = a'$ *AND* $t_1 \cong t'_1$ *AND* $t_2 \cong t'_2$

$| U(t_1, t_2, a),\ U(t'_1, t'_2, a') \rightarrow$

$\quad a = a'$ *AND* $t_1 \cong t'_1$ *AND* $t_2 \cong t'_2$

$| R(t, m, n, a),\ R(t', m', n'a') \rightarrow$

$\quad a = a'$ *AND* $t \cong t'$

$| O(t, a),\ O(t', a') \rightarrow$

$\quad a = a'$ *AND* $t \cong t'$

$| S(b, a),\ S(b', a') \rightarrow$

$\quad a = a'$ *AND* $b = b'$

As an example, consider the following fragment of the IMDB schema:

define element IMDB { *type* Actor*, *type* Director* }

define type Actor { *element* ACTOR { *type* Name, *type* Biography } }

define type Director { *element* DIRECTOR { *type* Name, *type* Directed* } }

define type Name { *element* NAME { *xsd:string* } }

The type Name is *shared* by both Actor and Director. The type split operation separates the two occurrences by renaming the type Name as follows:

```
define element IMDB { type Actor*, type Director* } }
define type Actor { element ACTOR { type Name, type Biography } }
define type Director { element DIRECTOR { type DirectorName, type Directed* } }
define type Name { element NAME { xsd:string } }
define type DirectorName { element NAME { xsd:string } }
```

The converse operation of type merge would start from the second schema and recognize that the *type structure* of Name and DirectorName are syntactically equal and give them a common name to get back the original schema.

3.3.3 Union Distribution and Union Factorization

Union distribution and union factorization *change* the structure of the schema by introducing new types. For example, consider the following fragment of IMDB which contains a union of Movie and Tv.

```
define type Show { element SHOW {
   type Title, type Year, type Aka*, type Review*, (type Movie | type Tv) }}
```

Distributing this union gives us the following fragment in which Movie-related information has the potential to be separated from the Tv-related information.

```
define type Show {
   (element SHOW { type Title, type Year, type Aka*, type Review*, Tv} ) |
   (element SHOW { type Title, type Year, type Aka*, type Review*, Movie} )
}
```

Performing two outlines and four type splits on all the shared types ensures that the distribution of the union completely separates the Movie and Tv information. That is,

```
define type Show1 { element SHOW {
   type Title1, type Year1, type Aka1*, type Review1*, Tv) }}
define type Show2 { element SHOW {
   type Title2, type Year2, type Aka2*, type Review2*, Movie }}
```

The converse operation of union factorization is analogous to union distribution. Union factorization may involve a *type merge* – analogous to the type split operation in the union distribution. That is, Title1 and Title2, Year1 and Year2, Aka1 and Aka2, Review1 and Review2 would all have to be first type merged before Show1 and Show2 can be factorized into Show.

The formal definition of union distribution and factorization is given below:

$$E(label, C(t_c, U(t_1, t_2, null), null), n)$$

$$\longleftrightarrow$$

$$U(E(label, C(t_c, t_1, null), n_1), E(label, C(t_c, t_2, null), n_2), null)$$

3.3.4 Repetition Split and Repetition Merge

Repetition split, as the name implies, *splits* a given repetition into more than one. That is, it *distinguishes* the *occurrences* of a given type – for example, the first occurrence, second occurrence, etc. In fact, repetition split can be performed an infinite number of times if the repetition itself is infinite. An example of two consecutive repetition splits which distinguish the first two occurrences is given below:

define type Actor = *element* ACTOR {Name, Played+}

→

define type Actor = *element* ACTOR {Name, Played, Played*}

→

define type Actor = *element* ACTOR {Name, Played, Played?, Played*}

As with union distribution, we do a type split operation on Played to establish the distinction of one Played from the other.

define type Actor = *element* ACTOR {Name, Played1, Played2?, Played3*}

Repetition merge is the converse operation of repetition split – the order of repetition merges always follows the *reverse* order of repetition splits – in the example above, Played2 and Played3 would be merged first.

We formally define a repetition split and repetition merge as follows:

$$R(t, 0, unlimited, null)$$
$$\longleftrightarrow$$
$$C(R(t, 0, 1, null), R(t, 0, unlimited, null), null)$$
$$R(t, 1, unlimited, null)$$
$$\longleftrightarrow$$
$$C(R(t, 1, 1, null), R(t, 0, unlimited, null), null)$$

3.3.5 Repetitions to Unions

This operation enables the straighforward application of union distribution to repetitions. For example, consider the following definition of the type Show:

define type Show { *element* SHOW {*type* Title, *type* Aka* } }

There could be lots of shows which *do not* have alternative titles (that is Akas). We utilize the repetitions to unions operation to explicitly state this:

define type Show { *element* SHOW {*type* Title, (ϵ| *type* Aka+) } }

We have now converted the repetition into a union and can perform a union distribution as defined before leading to two types of Shows – those with Akas and those without. Formally, we define this operation as follows:

$$R(t, 0, unlimited, null)$$
$$\rightarrow$$
$$U(Empty, R(t, 1, unlimited, null), null)$$

3.4 Recursion

We provide an *unroll* transformation to unroll the recursion. Note that while the schema itself may be recursive, the data conforming to that schema is *non-recursive* and corresponds to a schema in which the recursion has been unrolled a certain number of times. And so, a mechanism is necessary by which the recursion in the schema is unrolled as

and when required (for example, when validating the XML document, if the same type is revisited in a cycle, then that type can be unrolled on the fly to create a new type). By default, we unroll each recursion exactly once in any recursive schema.

As an example, consider the following fragment from the XMark Schema:

```
define type Text { element text
  { (xsd:string | type Bold | type Text | type Keyword | type Emph })* }
define type Bold { element bold
  { (xsd:string | type Bold | type Text | type Keyword | type Emph })* }
define type Emph { element emph
  { (xsd:string | type Bold | type Text | type Keyword | type Emph })* }
define type Keyword { element keyword
  { (xsd:string | type Bold | type Text | type Keyword | type Emph)* } }
```

Unrolling the above complicated recursion once, will give us the following fragment[1]:

```
define type Text1 { element text
  { (type Bold1 | type Bold2 | type Keyword1 |
  type Keyword2 | type Emph1 | type Emph2 | type Text2 )* }}
define type Bold1 { element bold
  { (type Keyword2 | type Text2 | type Emph2 | type Bold2)* }}
define type Keyword1 { element keyword
  { (type Bold2 | type Text2 | type Emph2 | type Keyword2)* }}
define type Emph1 { element emph
  { (type Bold2 | type Text2 | type Keyword2 | type Emph2)* }}
define type Text2 { element text {xsd:string }}
define type Bold2 { element bold {xsd:string }}
define type Keyword2 { element keyword {xsd:string }}
define type Emph2 { element emph {xsd:string }}
```

[1]Note that the mixed content – indicating the interleaving of text and markup – of the original fragment has been factored out.

3.5 Validation and Schema Transformations

Validation is a process which takes as input a schema S and a document D and checks whether D is *valid* with respect to S. That is, (i) whether D is well-formed and, (ii) whether D obeys the constraints (both structure and value) specified in S.

We first show that applying the transformations defined so far does not change the set of documents validated. Formally, let $\mathcal{D}$ be the set of documents validated by a schema S. Let T be the transformation applied on S to get a new schema $T(S)$. The set of documents validated by $T(S)$ is exactly $\mathcal{D}$.

As mentioned in Section 3.3, we define two types of schema transformations: (i) manipulation of type names and, (ii) using equivalent regular expressions. Clearly, (i) satisfies our property since no change is being made to the tag structure of the XML Schema. We can prove that (ii) also satisfies this property by proving that the equivalent regular expressions used are indeed equivalent to each other. We list each of the structure-changing transformations below:

Union Distribution and Factorization: We prove the property that performing a union distribution does not change the set of validated documents. That is, $L(M|N) = LM|LN$.

Proof:

Let $w \in L(M|N)$. Then, w can be written as w_1w_2 where $w_1 \in L$ and $w_2 \in M|N$. If $w_2 \in M$, then $w_1w_2 \in LM$ and hence to $LM|LN$. Similarly if $w_2 \in N$, then $w_1w_2 \in LN$ and hence to $LM|LN$.

Conversely, let $w \in LM|LN$. If $w \in LM$, then w can be written as w_1w_2 where $w_1 \in L$ and $w_2 \in M$. Therefore, $w_2 \in M|N$ and hence $w_1w_2 \in L(M|N)$. Similarly, if $w \in LN$, then then w can be written as w_1w_2 where $w_1 \in L$ and $w_2 \in N$. Therefore, $w_2 \in M|N$ and hence $w_1w_2 \in L(M|N)$. □

Repetition Split and Merge: $X^* = X?X*$, $X^+ = XX^*$. Proof is immediate.

Repetitions to Unions: $X^* = \epsilon|X+$. Proof is immediate.

An important property of the XML Schema (which is a part of its specifications) is that it should be *1-unambiguous*. That is, it should be possible to assign a type to a token without having to do a look-ahead. This property results in a linear validation algorithm proportional to the size of the data.

In case of most of the schema transformations, validation is straightforward. That is, these transformations result in schemas which are 1-unambiguous. However, there are two transformations, which violate the 1-unambiguity condition: Repetition Split and Union Distribution.

In the case of repetition split, only the case of $X^* = X?X^*$ is ambiguous since it is not clear whether a given x should be validated to $X?$ or X^*. And so, we use the convention of "longest match" to resolve the ambiguity.

In the case of Union Distribution, clearly, there is no ambiguity in the type assignments. But, since union distribution violates the 1-unambiguity property, the validation is no longer linear. For example, when the type Show is distributed in the schema, the type assignment for the tag `SHOW` in the document cannot be immediately determined. The validator has to look ahead until it finds the token `TV` or `MOVIE` to determine whether to assign the type Show1 or Show2 to the tag `SHOW`. Hence, standard XML validators such as Xerces [76], will be unable to validate any of these schemas. But there are other methods which can be used to validate against such schemas [45]. In our work, we make use of the standard validators, whenever possible, for the task of *statistics collection*. We have built our own statistics collection module for non-standard XML Schemas that we generate.

3.6 Implementation of Transforms

In order to implement the transforms, we visualize the XML Schema as a *tree of constructors* – that is, a schema tree.

To illustrate the representation of a schema tree, consider the partial XML Schema in Figure 3.2. Here, Title, Year, Aka and Review are simple types. The schema tree for an excerpt of this schema is shown in Figure 3.3 (note that base types are not shown). Nodes in the tree are *annotated* with the *names* of the types present in the original schema –

```
define element IMDB {
   type Show*, type Director*, type Actor* }
define type Show {
   element SHOW { type Title, type Year, type Aka*, type Review*,
      (type Movie | type Tv) }}
define type Director { element DIRECTOR {
   type Name, type Directed*}}
define type Directed {
   element DIRECTED {type Title, type Year, type Info }}
```

Figure 3.2: The (partial) IMDB Schema

these annotations are shown in sans serif next to the tags (shown in `typewriter` font) in Figure 3.3. Some points are worthy of note. First, there need not be any correspondence between tag names and annotations (type names). Second, the schema graph is represented as a *tree*, where shared types are *repeated* at nodes where they occur, but the *annotation* remains the same (see *e.g.,* the nodes $\texttt{TITLE}_1$ and $\texttt{TITLE}_2$ in Figure 3.3 – they both correspond to the *same type*, since their annotation is the same). Finally, recursive types can be handled similarly to *shared* types, *i.e.,* the base occurrence and the recursive occurrence are differentiated, but both correspond to the *same type* if their annotations are the same.

Any subtree in the schema tree can be regarded as a type and the node corresponding to that subtree can be annotated *without* changing the structure of the tree. We refer to this annotation as the *name* of the node and use it synonymously with annotation.

Transformations which manipulate type names, such as, inline, outline and type split/merge, can be performed on the schema tree simply by adding, deleting or renaming the type name of a single node or set of nodes.

We perform *structure changing* transformations through the process of tree pattern matching and replacement. For example, in order to perform a union distribution, we need to first find the patterns shown in Figures 3.4(a) and (c) in the *original* schema. Next, we need to *transform* these patterns into those shown in Figures 3.4(b) and (d), respectively. Similarly, Figure 3.5(a) shows the pattern to be searched for in order to

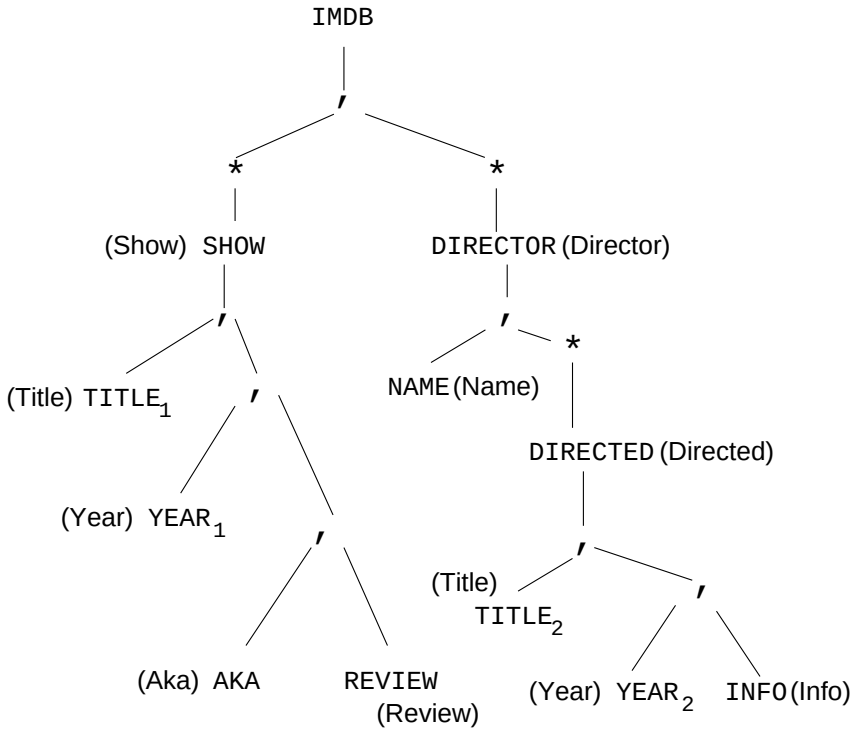


Figure 3.3: (Partial) Schema Tree for the IMDB Schema

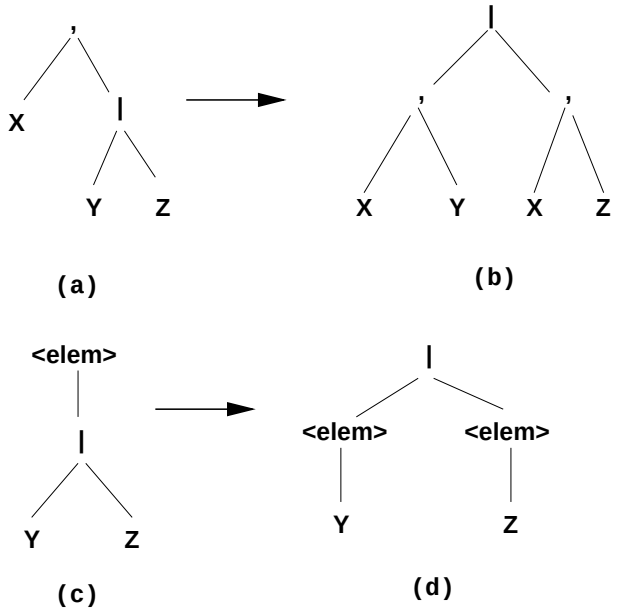


Figure 3.4: Patterns for Union Distribution and their Transformation

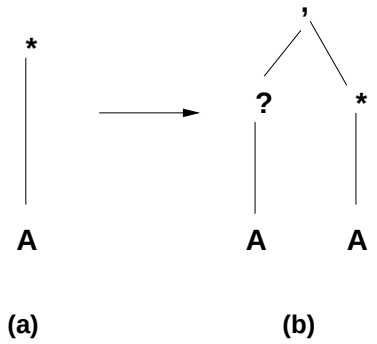


Figure 3.5: Pattern for Repetition Split and its Transformation

locate a repetition split and the pattern which should replace it.

3.7 Conclusions

In summary, this chapter introduced notations and transformations used in this thesis. The impact of the transformations on validation was discussed. While all transformations defined here do not change set of documents validated by the resulting schema, the validation algorithm would need to change based on whether the transformation violates the 1-unambiguity condition. The implementation of the transforms can be done by regarding the schema as a tree of constructors. The transforms are performed by either tree pattern matching and replacement for structure changing transforms or by manipulating the annotations of the tree nodes.

Chapter 4

Statistics Collection and Query Result Size Estimation

4.1 Introduction

In this chapter we describe StatiX – a framework for statistics collection and cardinality estimation. Its design is based on the following:

XML Schema-based statistics collection: Using the XML Schema as the basis for statistics collection enables StatiX to produce concise and accurate summaries of XML data. Moreover, the use of XML Schemas is becoming commonplace in a large number of XML applications.

Histogram summaries: A large variety of mechanisms are available for representing statistical summaries (for example, path trees [1], graph summaries [54], etc.). We have specifically selected histograms for this purpose – the use of histograms enables StatiX to maintain scalable and symmetric summaries of both the *structures* of the types as well as the *values* in the data.

Basing statistics on XML Schema types facilitates the re-use of standard XML technology, namely, *validating parsers*, for statistics gathering. Another advantage of type-based statistics is that the granularity of statistics can be tuned. That is, since StatiX collects

statistics *for each named type* in the schema, more detailed or less detailed statistics can be gathered through *schema transformations*. The schema transformations defined in Chapter 3 allow the addition of type names (through transformations such as outline, repetition split, union distribution, etc.) as well as deletion of type names (inline, repetition merge, union factorization, etc.). Thus, granularity of the statistics can be changed appropriately depending on the memory budget. An important consequence of schema-based statistics collection is that the *maximum size* of the summary depends on the size of the schema, not on the size of the data and this can be determined *beforehand* – that is, even before gathering the statistics. Also, using schema information on the occurrences of elements can help in considerably *reducing* the size of the final summary.

Using histograms to store structural summaries elegantly captures the data skew prevalent in XML documents. Histograms are attractive because they are simple to implement, have been well-studied, and proven to be effective for selectivity estimation [57, 58]. Moreover, because histograms are already widely used in relational database engines, our framework can be easily integrated with these systems.

StatiX can currently handle a significant subset of XQuery – namely, branching path expressions with value predicates. A detailed evaluation over different datasets and query sets shows that StatiX can provide extremely accurate summaries at very reasonable memory budgets.

Organization. The rest of this chapter is organized as follows. In Section 4.2 we describe the components of StatiX summaries and then, in Section 4.3, we outline the algorithm for cardinality estimation. In Section 4.4 we identify some limitations of using histograms and propose the use of schema transformations to improve the accuracy of StatiX summaries. In Section 4.5 we describe how to construct StatiX summaries. The experimental setup and performance evaluation are described in Sections 4.6 and Section 4.7, respectively. Finally, we conclude in Section 4.8.

4.2 Description of StatiX Summaries

StatiX collects statistics based on the schema given by the user. It distinguishes types which are *named* and those which are not. That is, statistics are collected only for types which are *outlined* in the schema (Chapter 3). However, a restriction imposed by StatiX is that only types with the tag constructor can be outlined. Each outlined type has two types of histograms associated with it:

Structural Histogram: Given a type T and it's parent T_p, a structural histogram $H(T)$ captures the distribution of the elements of type T with respect to its parent T_p. We also refer to the structural histogram as the *parent histogram* of type T.

Value Histogram: If T is a simple type, then, in addition to a structural histogram, a value histogram is also constructed. The value histogram captures the distribution of values of type T. Currently, equi-depth integer value histograms are constructed in StatiX.

In addition to the basic two types of histograms above, other statistics, such as the number of distinct values, number of null values, etc. are gathered. An example of an XML schema and a possible StatiX summary corresponding to this schema is shown in Figure 4.1. The schema describes a database which contains information about shows. A show can be either a movie or a TV show; has a title and year of release; and may contain zero or more reviews, and zero or more alternative titles (*i.e.,* `AKA`). The summary contains statistical information about all types defined in the schema. For each complex type, it records the type cardinality, *i.e.,* the number of occurrences of that type in the document; its id (or key) range (which can be regarded as a trivial, single bucket *key histogram*); and its parent histogram. For example, the type Review has cardinality 16; ids ranging from 1 to 16[1]; and a parent histogram corresponding to Show, which indicates that there are 8 instances of `REVIEW` under `SHOW`s with ids from 1 to 3 and 8 instances under `SHOW`s with ids from 4 to 5. Simple types, that correspond to elements with atomic

[1]In StatiX summaries, intervals are left-closed and right-open.

content, are associated with value histograms. For example, the type Year has a value histogram indicating that there are 3 occurrences of Year with values between 1990 and 1993, and 2 occurrences with values between 1994 and 2000.

```
define element IMDB {
    type Show* }

define type Show {
  element SHOW {
    element TITLE {xsd:string },
    type Year,
    element AKA { xsd:string }*,
    (element MOVIE {
      element BOXOFFICE{xsd:integer } } |
    type Tv),
    type Review* } }

define type Tv {
  element TV {
    element SEASONS{xsd:string } } }

define type Review {
  element REVIEW {
    element RATING{xsd:integer }
    element COMMENT{xsd:string } } }
```

(a)

```
define stat Show {
  cardinality { 5 }
  id_domain { 1 to 6 } }

define stat Review {
  cardinality { 16 }
  id_domain { 1 to 17 }
  parent histogram Show {
    bucket number { 2 }
    buckets {
      from 1 to 4 count 8,
      from 4 to 6 count 8 } } }

define stat Tv {
  cardinality { 2 }
  id_domain { 1 to 6 }
  parent histogram Show {
    bucket number { 1 }
    buckets {
      from 1 to 6 count 2 } } }

define stat Year {
  value_domain { 1990 to 2001 }
  number distinct {5}
  bucket number { 2 }
  buckets {
    from 1990 to 1994 count 3,
    from 1994 to 2001 count 2 } }
```

(b)

Figure 4.1: IMDB schema and the corresponding StatiX summary

4.3 Estimating Query Result Cardinality in StatiX

StatiX estimates the result cardinality of XML queries using histogram multiplication. Since path queries are expressed in terms of element (tag) names, and StatiX collects statistics for types, the tags in the query are first mapped to the corresponding types; and then the structural and value histograms corresponding to the tags in the path are multiplied. If a structural histogram is not available for a given tag, a uniform-distribution is assumed for that tag.

Input: c, $\mathcal{H}$
c is the path expression identifying the location
$\mathcal{H}$ is the set of histograms (value and structure) for all types corresponding to the elements in c

1: let $c = /t_1[b_1]/t_2[b_2]/t_3[b_3]/.../t_n[b_n]$
   $\{t_i$ is the tag (correspondingly, its type is $T_i)\}$
2: **for all** $i \in 1$ to n **do**
3: B_i = result distribution of b_i
4: $J_i = B_i \bowtie keyHist(T_i)$
5: $keyHist(T_i)$ = key distribution of T_i based on J_i
6: $parentHist(T_i)$ = compute distribution based on $keyHist(T_i)$
7: **end for**
8: **for all** $i \in 1$ to $n-1$ **do**
9: $J_i = keyHist(T_i) \bowtie parentHist(T_{i+1})$
10: $keyHist(T_{i+1})$ = distribute $freq(J_i)$ into $keyHist(T_{i+1})$
11: **end for**
   {Cardinality of the update}
12: $card$ = frequency (J_n)

Algorithm 1: Cardinality Estimation in StatiX

Algorithm 1 describes the cardinality estimation of *branching path expressions*, given a StatiX summary. The general format of such a branching path expression is $/t_1[b_1]/t_2[b_2]/.../t_n[b_n]$, where t_i is the tag and b_i is a path expression which may contain value and structural predicates. In the sequel, we use T_i to denote the *type* corresponding to the tag t_i. The cardinality estimation procedure operates in two stages: (i) compute the key distribution and parent-key distribution for each of the t_is in the presence of predicates *individually* (lines 2 through 7); (ii) use these individual distributions to compute

the overall key distribution of the complete query (lines 8 through 11).

There are three basic operations – histogram multiplication (lines 4 and 9), finding the key distribution (line 5), and finding the parent key distribution (line 6). Histogram multiplication is a well-known operation to find the join estimate given two histograms [29]. Below, we describe the other two operations in more detail.

Key distribution. Note that when two histograms are multiplied, one of the histograms is the key histogram having values which occur exactly once. However, the join distribution gives the total number of tuples in the result – that is, the values in the key histogram may occur multiple times in the result. From this join histogram, we need to determine which distribution of keys occurs in the join (line 5). The fact that keys are unique can be used to compute this distribution as follows: (i) initially, construct the key distribution K by dividing the key histogram into the same number of buckets as the join histogram and in which the frequency of each bucket is the same as its range, (ii) for corresponding buckets j_i in the join histogram and k_i in the key distribution histogram, if frequency of j_i is less than the frequency of k_i, change the frequency of k_i to that of j_i. The resulting histogram is the statistically determined distribution of keys in the join. This histogram is used to compute the parent key distribution described next.

Parent key distribution. An important observation in the case of structural histograms is that the node ids (keys) and parent ids have a strong correspondence with each other – that is, if $nodeid_1 > nodeid_2$, then $parentid(nodeid_1) >= parentid(nodeid_2)$. The parent histogram is a summarization of this correspondence, as illustrated in Figure 4.2. Using this observation, we can compute the parent key distribution as shown by the example next.

Consider the case where the parent histogram of Review (with respect to Show) is `[1-4: 8; 4-6: 8]`. The multi-bucket key histogram of Review would then be `[1-9: 8; 9-17: 8]`. Conversely, suppose Review has now been "filtered" through a value predicate (say, Reviews with Rating > 6) leading to the following key histogram for Review: `[1-9: 5; 9-17: 3]`. The corresponding distribution in the parent histogram of Review is now:

$Card_{Year} = \sigma_{<1992}$ (Year)	1.5
Key_{Show} = distribute $Card_{Year}$ into id range of Show	[1-6: 1.5)
$Card_{Review}$ = freq ($parentHist$(Review) $\bowtie$ Key_{Show})	≈ 5

Table 4.1: Cardinality Computation in StatiX

[1-4: 5; 4-6: 3]. This is because, from the key histogram, we know that 5 of the first 8 Reviews are "relevant". We know from the parent histogram that the first 8 Reviews occur under the first 3 Shows. Hence, clearly, the 5 relevant Reviews of the first 8 Reviews, now occur under the same Shows. Similarly, we know from the key histogram that 3 of the last 8 Reviews are relevant and from the parent histogram, we know that these 3 Reviews occur under Shows 4-5. Hence, the second bucket of the parent histogram now gets a count of 3. This computation is a direct consequence of the observation made in the previous paragraph. This new parent histogram is then used to compute the cardinality and join distribution of the result (lines 8 to 12).

Consider the cardinality estimation of the following query asking for all Reviews of Shows made before 1992, on data corresponding to the schema in Figure 4.1:

```
Query 1:    //SHOW[YEAR < "1992"]/REVIEW
```

Here, the mapping of element names to type names is straightforward, and in order to compute the query cardinality, we perform the computations outlined in Table 4.1. The result distribution of the branch `YEAR < ''1992''` is first calculated (line 3 in Algorithm 1). Then, in the next step, this distribution is distributed into the key histogram of the parent Show (line 5 in Algorithm 1). Next, moving to the main branch, the parent histogram of Review is multiplied with the newly computed key histogram of Show (step 9 in Algorithm 1). The frequency of this join histogram is the cardinality of the query (step 12 in Algorithm 1). Hence, we conclude that the cardinality of the query (that is, the number of Reviews) is approximately 5. Note that all the steps from the algorithm not shown in Table 4.1 have no bearing on the final result and hence are not shown explicitly.

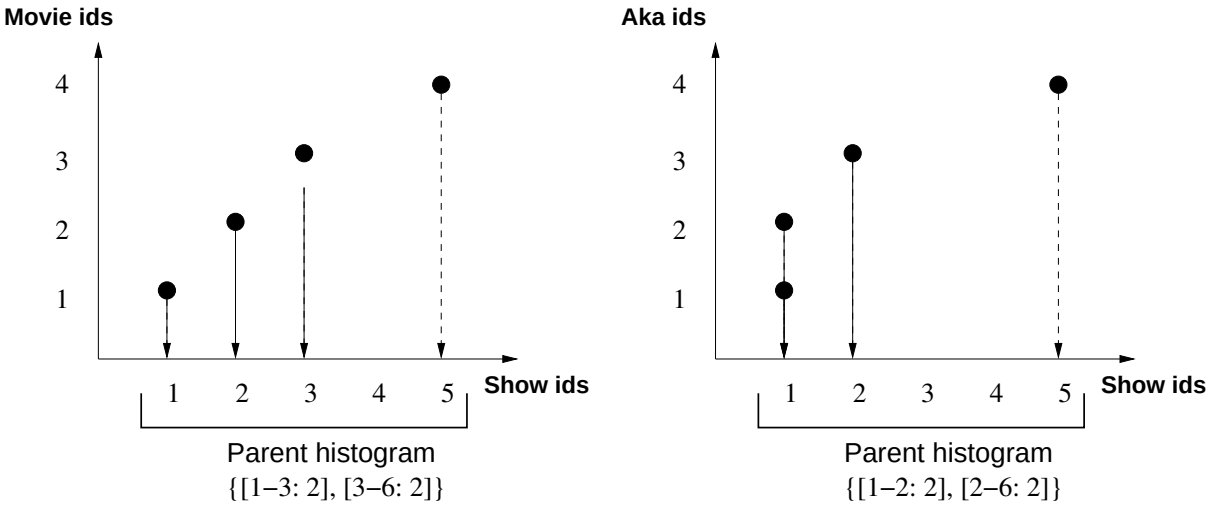


Figure 4.2: Node and Parent ids have a Correspondence

4.4 Tuning the Accuracy of StatiX Summaries

The accuracy of StatiX summaries can be tuned by: (i) increasing/decreasing the number of buckets in the histograms; and/or by (ii) adjusting the *granularity* of the statistics collection.

4.4.1 Potential Limitations of Structural Histograms

Histograms are well-known to be concise and effective in capturing skew in the underlying data [57, 58]. The addition of more buckets to the histogram results in a more accurate reflection of the underlying data distribution. In StatiX, histograms are used for two different purposes – to capture the: (i) value skew (value histograms) and, (ii) the structural skew (structural or parent histograms).

The domain of any structural or parent histogram is potentially very large. Larger the number of occurrences of a particular type, larger its id domain. For example, the number of occurrences of a type like Show may be in the thousands – if there are 10,000 Shows in the database, then there are 10,000 distinct ids. This would affect the parent histogram of its children, say, Tv and Movie. Suppose approximately 20% of the Shows are Tv shows and the other 80% are Movie shows. Then, clearly, the way in which these Tv and Movie shows are *interleaved* in the data affects the parent histograms of both types.

For example, suppose Tv and Movie shows span almost the entire range of Shows. Then, the parent histograms of both types contain the same range of 10,000, and may not be able to capture the parent-child distribution accurately. On the other extreme, if all Tv shows occured before all Movie shows in the data, then the parent histogram of Tv would have a range of 1 to 2000 and that of Movie, from 2001 to 10,000. Now, neither parent histogram has any holes and very accurately reflects the distribution. In fact, a *single bucket* in each parent histogram, is enough to capture the distribution.

In effect, for a given number of histogram buckets, n, the histogram could potentially be less effective in capturing the skew, as the domain of values grows larger and the *occurrence pattern* of the type leaves too many gaps in the id range. In order to overcome this limitation, we propose the use of schema transformations to improve the summary accuracy in the following section.

4.4.2 Transformations for Finer Granularity Statistics

Since StatiX gathers statistics based on the types in the schema, there are several transformations which can be applied to the schema to increase or decrease the number of types and consequently finer or coarser-grained statistics can be collected. Although the types defined in an XML Schema do not appear in the document, they are used during validation as annotations to document nodes. These transformations can be used to improve the accuracy of the summary, since they reduce the problem of *interleaved* elements described in the previous section.

For example, consider the schemas shown in Figures 4.3 through 4.5. All three schemas are *equivalent* to one another. That is, they validate exactly the same set of documents. The schemas were derived as follows:

Schema 1: The original schema.

Schema 2: On Schema 1, perform a union distribution of Tv and Movie. Then, perform a type split. In Schema 2, Tv shows and Movie shows are separated. This, in effect, ensures that Shows with Tv have a *different id range* from Shows with Movie. Hence,

```
define element IMDB { type Show*}
define type Show { element SHOW
  { type Title, type Aka*, type Tv | type Movie} }
define type Title { element TITLE { xsd:string } }
define type Aka { element AKA { xsd:string } }
define type Tv { element TV { xsd:string } }
define type Movie { element MOVIE { xsd:string } }
```

Figure 4.3: Schema 1

```
define element IMDB { (type Show1 | type Show2)* }
define type Show1 { element SHOW  { type Title1, type Aka1*, type Tv} }
define type Show2 { element SHOW  { type Title2, type Aka2*, type Movie} }
define type Title1 { element TITLE { xsd:string } }
define type Aka1 { element AKA { xsd:string } }
define type Title2 { element TITLE { xsd:string } }
define type Aka2 { element AKA { xsd:string } }
define type Tv { element TV { xsd:string } }
define type Movie { element MOVIE { xsd:string } }
```

Figure 4.4: Schema 2

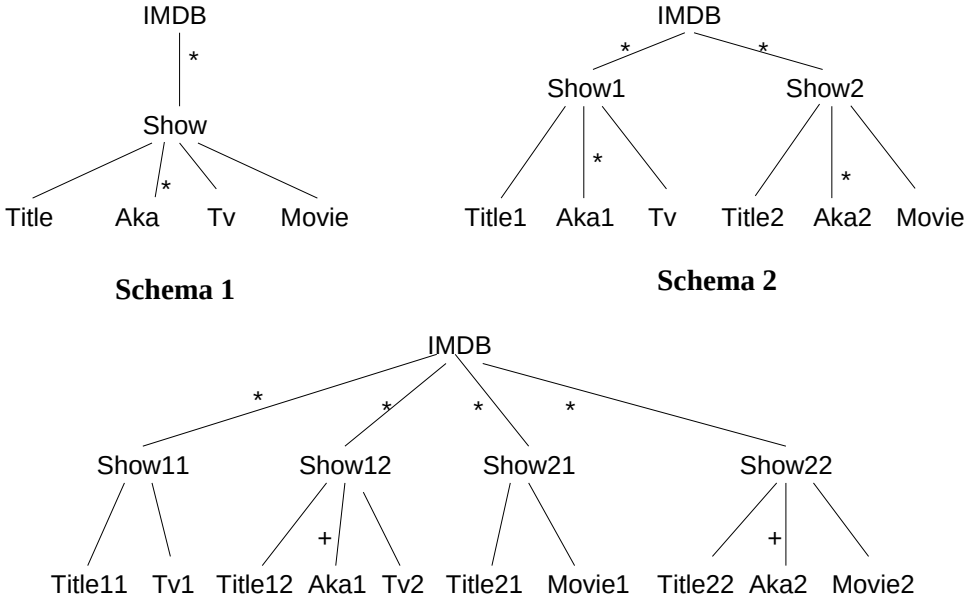
even if Tv and Movie shows are interleaved in the data, the type assignment ensures an *artificial* segregation, making the parent histograms of both Tv and Movie more accurate.

Schema 3: On Schema 2, convert the two repetitions of Aka – Aka1* and Aka2* into unions. That is, Aka1* = () | Aka1+. Subsequently, perform a union distribution to get 4 different Shows – (i) Show11 – Tv shows without Akas, (ii) Show12 – Tv shows with Akas, (iii) Show21 – Movie shows without Akas and (iv) Show22 – Movie shows with Akas. In this schema, in addition to the segregation of Tv and Movie shows, Shows *with* Akas and those *without* are also segregated, thus reducing the impact of interleaved Tv and Movie Shows with and without Akas.

A slightly modified tree representation of each of these schemas is shown in Figure 4.6. The tree representation only shows the *nesting* and *repetition* of types, but not the

```
define element IMDB { (type Show11 | type Show12 | type Show21 | type Show22)* }
define type Show11 { element SHOW   { type Title11, type Tv1} }
define type Show12 { element SHOW   { type Title12, type Aka1+, type Tv2} }
define type Show21 { element SHOW   { type Title21, type Movie1} }
define type Show22 { element SHOW   { type Title22, type Aka2+, type Movie2} }
define type Title11 { element TITLE { xsd:string } }
define type Title12 { element TITLE { xsd:string } }
define type Aka1 { element AKA { xsd:string } }
define type Title21 { element TITLE { xsd:string } }
define type Title22 { element TITLE { xsd:string } }
define type Aka2 { element AKA { xsd:string } }
define type Tv1 { element TV { xsd:string } }
define type Movie1 { element MOVIE { xsd:string } }
define type Tv2 { element TV { xsd:string } }
define type Movie2 { element MOVIE { xsd:string } }
```

Figure 4.5: Schema 3



Schema 1

Schema 2

Schema 3

Figure 4.6: Type graphs of the Three Schemas

Q.no.	Query	Most Accurate
1	/IMDB/SHOW	1,2,3
2	/IMDB/SHOW/TITLE	1,2,3
3	/IMDB/SHOW/AKA	1,2,3
4	/IMDB/SHOW[TV]/AKA	2, 3
5	/IMDB/SHOW[MOVIE]/AKA	2, 3
6	/IMDB/SHOW[AKA]/TITLE	3
7	/IMDB/SHOW[AKA && TV]/TITLE	3

Table 4.2: Queries, Schemas and Accuracy

constructors. The artificial segregation can be clearly seen. Schema 1, Schema 2 and Schema 3 are in *increasing order of granularity* – that is, not only does each schema have more types than the previous one, but the larger number of types contributes to a *more accurate summary*. Though not apparent from the example, which gives the impression that the larger granularity summary has a larger memory budget because of the increase in the number of types, the estimation accuracy of the finer granularity summary increases *even with a fixed memory budget*. This is shown in Section 4.7.

Consider the queries in Table 4.2. For each query, the most accurate summary is shown in the third column. Note that each of the 7 queries can be answered using any of the three schemas in conjunction with Algorithm 1. However, we are concerned with the accuracy of these estimates.

For example, for query 4, it is necessary to distinguish Tv shows. Clearly, Schema 1 does not distinguish between Tv and Movie shows – they are both nested under the same type Show. However, Schema 2 clearly separates Tv shows into a separate type (Show1). And so, Schema 2 is able to give an accurate estimate of the cardinality. Similarly, Schema 3 also distinguishes Tv shows – Show11 and Show12. As another example, consider query 7. Here, it is necessary to not only distinguish Tv shows, but those Tv shows which also have Akas associated with them. Clearly, Schema 3 is the only schema which makes this separation.

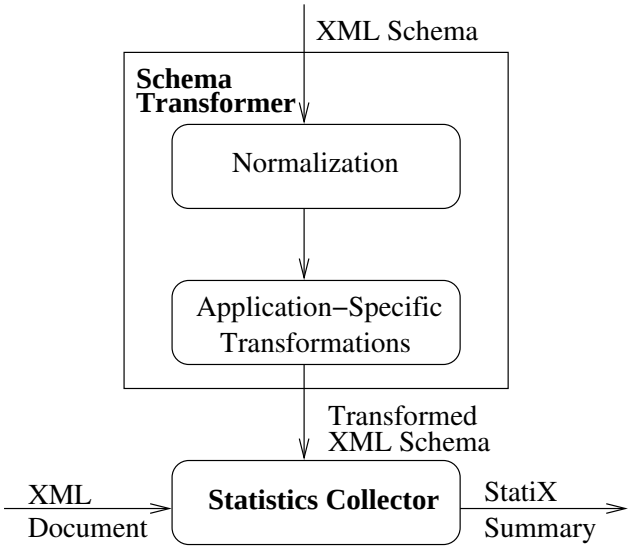


Figure 4.7: Building StatiX Summaries

Summary Compression. The XML Schema can also be used to *compress* the summary. We make the observation that the statistics of some types can be *inferred* directly from the statistics of their parents. That is, if a child type T occurs exactly once under its parent T_p, then the distribution of T under T_p is one-to-one and there is no need to store a structural histogram for T. That is, by inlining such types, we can reduce the size of the summary. As shown in the experiments section, this simple compression scheme significantly reduces the amount of space occupied by the summary because there are several such 1:1 occurrences in real-life schemas.

4.5 Construction of a StatiX Summary

The architecture of the StatiX module, shown in Figure 4.7, depicts the two main components of StatiX: (i) the **Schema Transformer**, which enables statistics collection at different levels of granularity, and, (ii) the **Statistics Collector**, that takes as input, the transformed schema and simultaneously validates the document against the schema and gathers the associated statistics. In what follows, we describe these components in detail.

4.5.1 The Statistics Collector

While in practice, the Statistics Collector comes into play only after the Schema Transformer has completed its rewritings, for ease of exposition, we will describe the role of the collector first.

The statistics collector has two functions: validation of the source XML document against its XML Schema description, and the simultaneous collection of statistics for this schema. If the transformed schema is a valid XML Schema, then a standard validating parser such as Xerces [76] or Galax [28] can be used as the statistics collector. The statistics are gathered on a *per-type* basis.

The successful validation of an XML document against a given schema results in the assignment of types (defined in the schema) to the nodes in the document [71]. StatiX leverages this information to build the statistical summaries. Intuitively, as the document is validated, StatiX keeps track of the number of occurrences of each type, and how the instances of a given type are distributed over the instances of its parent type(s).

Statistics gathering proceeds as follows. Each type defined in the schema is associated with a unique *type id*. As a document is parsed and occurrences of a given type are encountered, a new sequential *node id* is assigned to each occurrence. The concatenation of type id and node id uniquely identifies a given node in the document tree. Note that the order of occurrence of the type in the document determines the order in which node ids are assigned. For each type defined in the schema, StatiX has an associated *parent set*. Since validation is performed in a top-down fashion, and a parent is always processed before its children, for each type instance encountered, the id of the parent node is incrementally added to the parent set of the corresponding child type. This information is later summarized in a *structural histogram*.

Assigning *contiguous* ids to a given type is critical to building accurate and concise histograms – the use of non-contiguous ids will necessarily result in large gaps within buckets as well as between buckets. Moreover, the assignment of contiguous ids automatically keeps track of the *order* of the occurrences. Since equi-depth histograms result in significantly smaller estimation errors as compared to equi-width histograms [53], we have

implemented the former in StatiX.

Besides structural information, StatiX also captures value distributions at the leaf-node level using *value histograms*. While structural histograms are unique to the XML context, value histograms are commonly used in traditional relational storage systems.

Handling Ambiguous Transformations

One of the major limitations in using a standard XML validator for statistics gathering is that a standard validator cannot handle *ambiguous* schemas (as was described in Section 3.5 in Chapter 3).

In general, validating a document is nothing more than recognizing that the grammar defined by the schema recognizes the document as either belonging to (valid) or not belonging to (invalid) the language recognized by it. It follows that validating a document against a non 1-unambiguous schema involves recognizing whether or not a particular regular expression is satisfied before a particular type assignment. For example, in the case of distributing the union of Tv and Movie shows, we need to recognize the regular expression:

```
<SHOW><TITLE>...</><YEAR>...</>.....<TV><SEASONS>...</></>.....</SHOW>
```

in order to determine that the SHOW node validates to Show1 (and not Show2), TITLE node validates to Title1, etc. That is, the tag TV has to be seen before assigning a type to SHOW. In order to build a statistics collection module for such schemas, we utilized a recently proposed programming language called CDuce [10]. CDuce provides for regular expression pattern matching with specific emphasis on XML-style patterns. That is, it is possible in CDuce to express regular expressions using XML elements, attributes, types, etc. CDuce is a functional language in the style of Caml [9], and has been proposed for designing efficient applications which use XML pattern matching extensively. The statistics collection method remains unchanged. That is, the nodes in the documents are assigned types, and the parent set of a given type contains the list of ids of its parent element. Once this validation process is complete, structural and value histograms are built.

4.5.2 Schema Transformer

The Schema Transformer first performs some basic operations on the input XML Schema to ensure that statistics are generated for *all* elements and attributes. That is, *both*, a structural as well as a value histogram (for base types) are associated with each element and attribute. It then performs additional transformations, as required. And so, the function of the schema transformer is two-fold: (i) "normalize" the input XML Schema and (ii) perform appropriate transformations. Normalization involves the following:

- Give type names to all elements and attributes in the Schema. That is, outline all elements and attributes.
- Ensure that no type is shared. That is, perform all type split operations, so that no type has more than one parent histogram. This step ensured that the implementation of the cardinality estimation algorithm described before was simplified.

The second function of the schema transformer is to apply application-dependent transformations to make the resulting summary more accurate. StatiX currently does not implement any specific algorithm which does application-specific transformations on the schema. But, in general, several application characteristics, such as the query workload, memory budget available, etc. can be utilized to automate the process of generating the appropriate summary.

Apart from the outline and type split operations performed during the normalization steps, the transformer can also apply the *repetition-to-union* and the *union distribution* transformations, which *increase* the accuracy of the summary. Examples of schemas with increasing accuracy were shown in Section 4.4.

From now on, we refer to the normalized schema as the N-Schema and the summary resulting from this schema as the N-Summary. This is the *coarsest* summary possible in StatiX, while the *finest* summary is obtained when *all* repetition-to-union and union distribution operations have been performed on the N-Schema. We refer to this *decomposed* schema as the D-Schema and the corresponding summary as the D-Summary. Schema 1

in Figure 4.3 is an example of an N-Schema, while Schema 3 in Figure 4.5 is the corresponding D-Schema.[2]

4.6 Experimental Setup

We performed several experiments to measure the effectiveness and efficiency of StatiX. All experiments were run on a Pentium IV, 2.4GHz machine with 1GB of main memory, running Redhat 8.0. The query and datasets used are described below.

Data: We synthetically generated several different datasets of varying sizes conforming to the IMDB schema using ToXgene [5]. The generated data contained moderate to high skew both in structure as well as values. Since the trends across these datasets were similar, we report on the results obtained for the 5MB dataset. We also experimented with about 30MB of the DBLP dataset available from [18].

Schema: As mentioned in Section 4.4, it is possible to collect statistics at various granularities – coarser to finer. In order to show the impact of tuning StatiX summaries, we present results for summaries based on two "extreme" schemas – the N-Schema and the D-Schema, described in Section 4.5.

Query workload: The query workload was generated by sampling the BF-bisimilar graph of the data described in [54]. We generated two separate query workloads – one containing *branching path expressions without value predicates* and the other containing *branching path expressions with value predicates*. The length of the path expression varied from 2 to 6 elements, with at least one branch and a maximum of two branches. Each branch had a single predicate (either structural or value). From now on, we refer to the query workload without value predicates as *BP* and the workload with value predicates as *VP*.

[2]Note that one more repetition-to-union transform is possible at the root, but we do not perform this transform since it would result in *multiple* roots when the union distribution is subsequently applied.

4.6.1 Metrics

With the above setup, we performed experiments to measure: (i) estimation accuracy, (ii) summary size, and (iii) overheads in statistics collection . Each of these metrics are described in more detail below.

Estimation Accuracy. In order to measure the estimation accuracy of the summary, we used *average relative error* as our metric. The relative error is defined as follows:

$$RE = \frac{ABS(Card_{Est} - Card_{Act})}{Card_{Act}}$$

where $Card_{Est}$ is the *estimated* cardinality of the query and $Card_{Act}$ is the *actual* cardinality of the query. The average relative error is then defined as:

$$ARE = \frac{\Sigma_{i=1}^{n} RE^i}{n}$$

where n is the total number of queries and RE^i is the relative error of the i^{th} query.

Size. In order to be effective, the size of the summary should be as small as possible. We show through our experiments that not only is the accuracy of the summary very high, but the size is also moderate, even when finer granularity statistics are collected. We tabulate the size in terms of both the number of types in the schema, as well as the number of bytes required to store the summary. We show the sizes with and without the compression technique, outlined in Section 4.4.

Overhead. We measure the timing overheads involved in collecting statistics which involves two major phases – validation and histogram construction. We tabulate these metrics for both summaries – N and D.

The experimental setup is summarized in Table 4.3.

Data	IMDB, DBLP
Schema	N (Normalized), D (Decomposed)
Query set	BP (BPE without value predicates), VP (BPE with value predicates)
Metrics	Estimation Accuracy, Summary Size, Statistics Collection Overheads

Table 4.3: Experimental Setup

4.7 Performance Evaluation

4.7.1 Estimation Accuracy

The estimation accuracy depends on two parameters: (i) the number of buckets allocated to the structural and value histograms, and, (ii) the granularity of the schema itself. We first discuss the estimation accuracy results for each of the N- and D- Summaries separately. Subsequently, we compare the estimation accuracy across the two schemas given a fixed budget.

N-Summary

We discuss the estimation accuracy of the BP and VP query workloads for both the IMDB dataset as well as the DBLP dataset. In addition to plotting the overall estimation accuracy, we sorted the queries in descending order of their relative errors and plotted the estimation accuracy of the top 30% and the top 50% of the queries in this list – that is, queries with the worst estimates.

BP workload. Figure 4.8 shows the estimation accuracy in terms of the average relative error for the BP workload for the IMDB dataset. The first point on the X-axis corresponds to a *single* structural histogram bucket per type – effectively, only the cardinality is stored and the uniform distribution assumption is made. Several important points to be noted from the graph are as follows:

1. After a certain "cutoff" at around 60 histogram buckets, the estimation accuracy

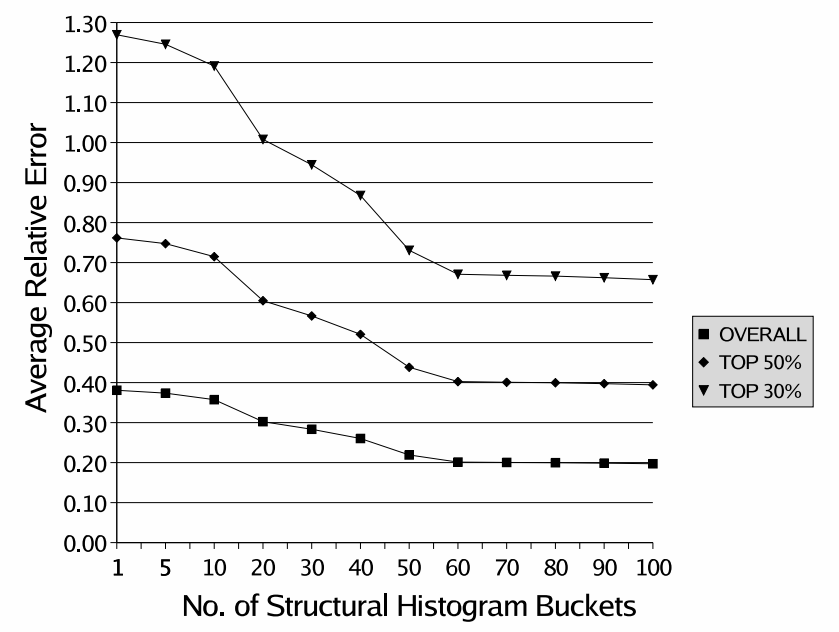


Figure 4.8: IMDB: Estimation Accuracy for BP Queries over N-Summary

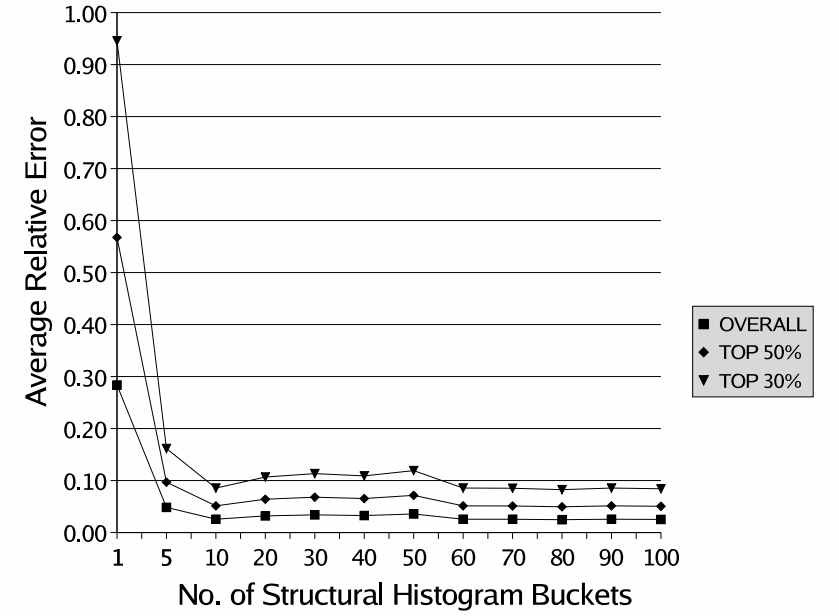


Figure 4.9: DBLP: Estimation Accuracy for BP Queries over N-Summary

does not improve substantially. As was discussed in detail in Section 4.4.1, the interleaving of elements in the dataset was the indeed reason for the lack of improvement in estimation accuracy. The results obtained imply that simply increasing the number of histogram buckets (unless a very large number of additional buckets are allocated) is not always an effective solution to the problem of bad estimates.

2. The estimation accuracy increases faster for the more inaccurate estimates when the number of histogram buckets is increased. The "TOP-30%" line falls more rapidly than the "TOP-50%" line. This indicates that increasing the number of histogram buckets does make a substantial difference to some of the queries. But after a certain cutoff, the gains are limited. Inspite of the substantial improvements, the worst top 50% and top 30% of errors is still unacceptably high, though the overall relative error is a moderate 20% (at 100 structural histogram buckets).

The reason for the poor estimation accuracy is the presence of a few queries with very large errors (about 8% of the estimates were over-estimates which were off by more than 100%). These queries had predicates which were optional, or part of unions. For example, //IMDB/SHOW[REVIEW]/TV, which asks for only Tv reviews. Note that Tv is part of a union and Review is repeated *zero* or more times per Show.

The estimation accuracy for the DBLP workload is shown in Figure 4.9. In contrast to the IMDB workload, the estimation accuracy is very high with only a small number of histogram buckets. Even for the top 30% of the worst estimates, the error is less than 12% for just 10 structural histogram buckets. And very small gains are obtained by increasing the number of buckets. This is due to the fact that the DBLP dataset does not have a lot of skew. And whatever skew exists is adequately captured by a small number of buckets. Moreover, when we examined the actual cardinality and the estimates, we found that the large errors in the dataset were found in a very small number of queries with small cardinality (comprising less than 10% of the total workload). For example, a query with cardinality 2 was estimated as 4 leading to a 100% error.

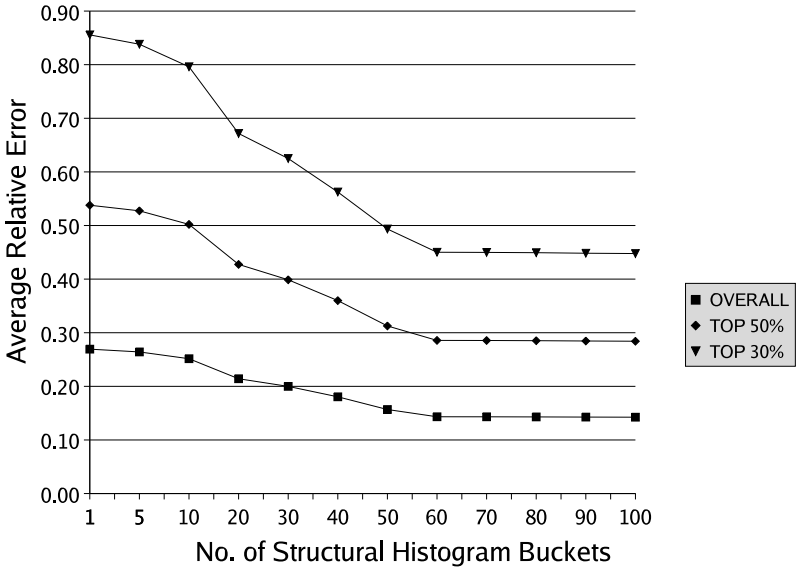


Figure 4.10: **IMDB:** Estimation Accuracy for VP Queries over N-Summary with 30 Value Histogram Buckets

VP workload. For the VP workload, we study the impact of increasing both the number of structural histogram buckets as well as the number of value histogram buckets. Figure 4.10 shows the estimation accuracy of the VP workload as the number of *structural* histogram buckets increases. The number of value histogram buckets was kept constant at 30. In addition to the overall estimation accuracy, the top 30% and the top 50% of the worst estimates are also plotted. As was indicated in the case of the BP workload, there is a cutoff beyond which the number of structural histogram buckets fails to make a significant impact.

Figure 4.11 shows the effect of increasing the number of *value histogram buckets* on the estimation accuracy. The graph shows 4 curves for 4 different settings of the number of structural histogram buckets. Note that the first point on the X-axis shows a *single* value histogram bucket – effectively, the range and cardinality of the values is stored and uniform distribution assumption is made. Only the *overall* estimation accuracy is shown.

Figure 4.12 shows the top 30% and top 50% of the worst estimates in addition to the overall accuracy for the 100 structural histogram bucket case. As before, the "TOP 30%" curve falls faster as the the number of value histogram buckets increases. Similar graphs

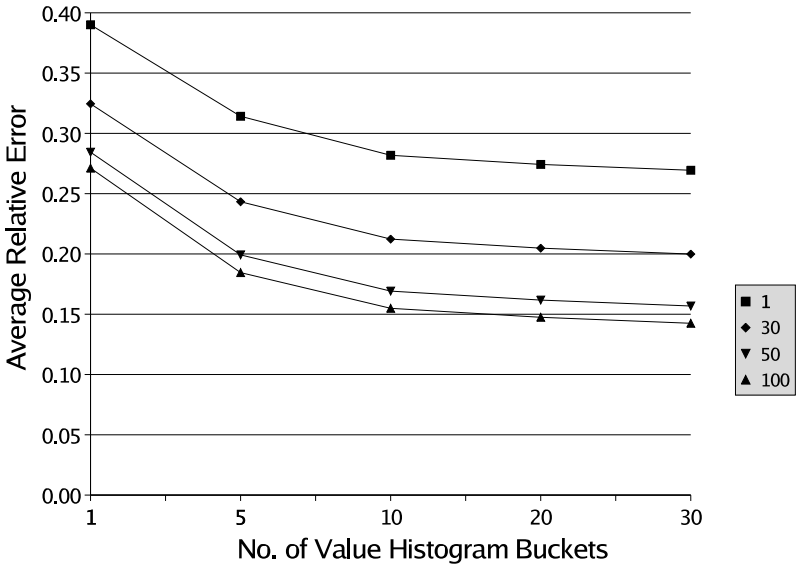


Figure 4.11: **IMDB:** Overall Estimation Accuracy for VP Queries over N-Summary with Increasing Value Histogram Buckets

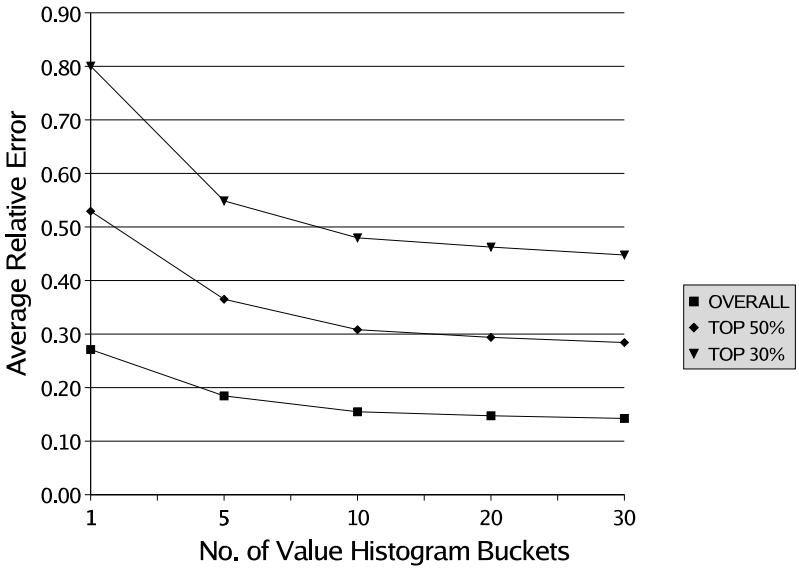


Figure 4.12: **IMDB:** Estimation Accuracy for VP Queries over N-Summary with Increasing Value Histogram Buckets and 100 Structural Histogram Buckets

were obtained for other values of the number of structural histogram buckets.

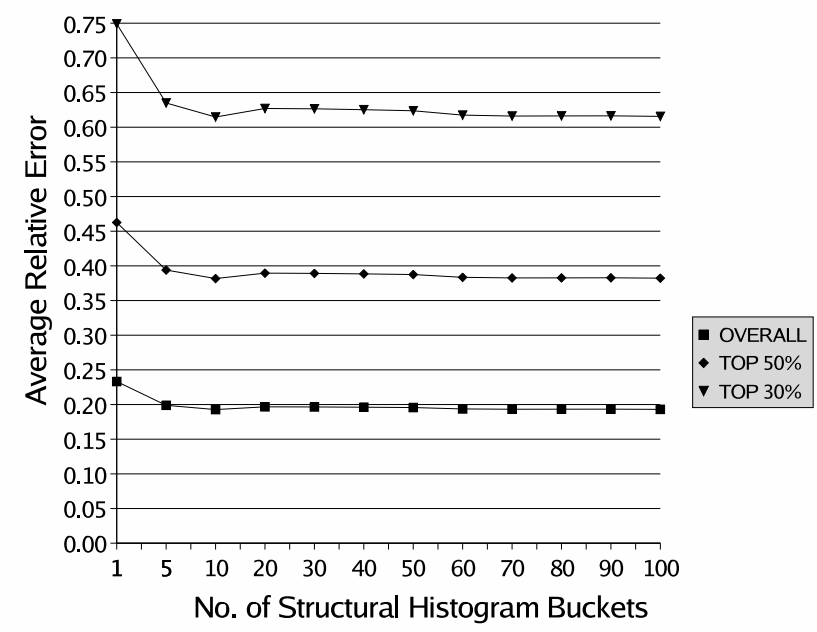


Figure 4.13: **DBLP:** Estimation Accuracy for VP Queries over N-Summary with 30 Value Histogram Buckets

Similar results were found for the DBLP dataset as well. Figure 4.13 shows the estimation accuracy when the number of value histogram buckets is kept constant at 30. Again, as indicated from the results for the BP queries, increase in the number of structural histogram buckets does not have a significant impact on the quality of the estimates.

Figure 4.14 shows the improvement in estimation accuracy as the number of value histogram buckets is increased. As seen from the graph, the improvement is quite significant – the overall relative error reduces from about 55% for 5 buckets to a little under 20% for 30 buckets, indicating that the dataset has a large amount of skew in the values (in contrast to the structure skew). The overall estimation accuracy for different settings of the number of structural histogram buckets follows very closely the same curve as that shown in Figure 4.14 and is not shown.

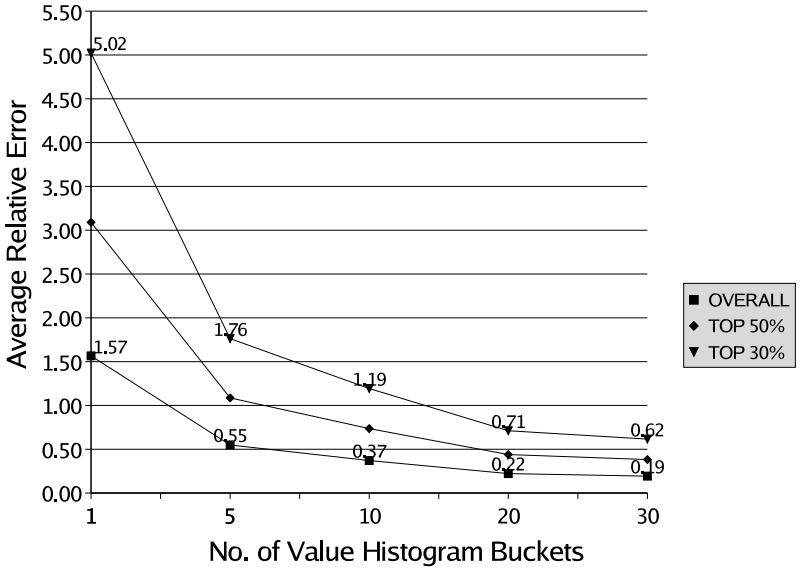


Figure 4.14: **DBLP:** Estimation Accuracy for VP Queries over N-Summary with Increasing Value Histogram Buckets and 100 Structural Histogram Buckets

D-Summary

We now move on to the estimation accuracy in the case of the D-Summary. Note that, given a fixed number of structural and value histogram buckets, the *size* of the D-Schema is *much larger* than the N-Schema. We first discuss the estimation accuracy of the D-Summary independently and then, in the next section, we compare the accuracies of the two summaries when they both have the same memory budget.

BP Workload. For the BP workload, the D-Summary of *both* IMDB as well as DBLP gave *100% estimation accuracy* – that is, the relative error of each of the queries in the workload was 0. This is not very surprising, given that the D-Summary is a very fine granularity summary.

VP Workload. We study the impact of increasing the number of structural histogram buckets as well as the number of value histogram buckets independently. Figure 4.15 shows the estimation accuracy for increasing number of structural histogram buckets when the number of value histogram buckets is kept constant at 30. As already indicated by the results of the BP workload, the increase in number of histogram buckets hardly has any

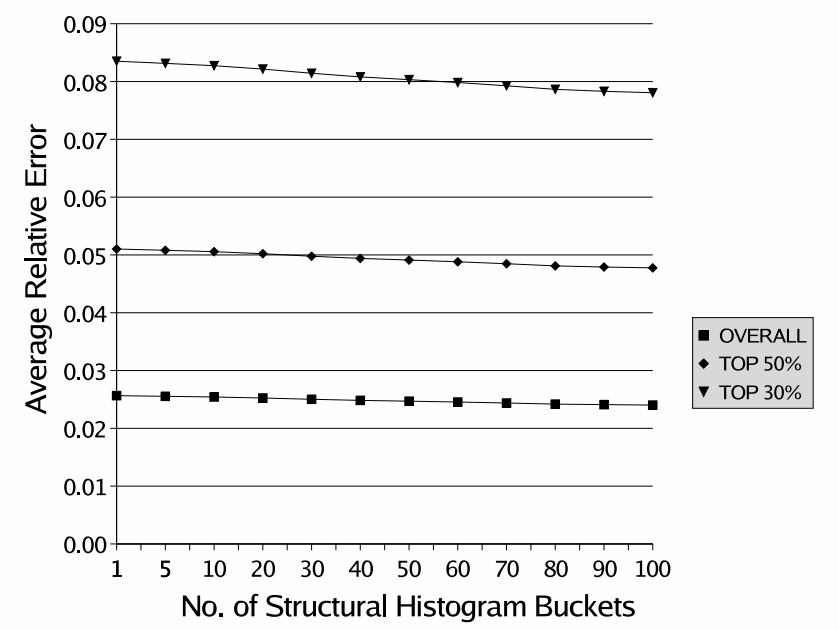


Figure 4.15: **IMDB:** Estimation Accuracy for VP Queries over D-Summary with 30 Value Histogram Buckets

impact on the estimation accuracy. Even in the case of the top 30% of the worst estimates, the improvement was a mere 0.5% from just storing the count to allocating 100 structural histogram buckets.

Figure 4.16 shows the estimation accuracy when the number of value histogram buckets increases. There are 4 different curves for 4 different settings of the number of structural histogram buckets. Note that only the *overall* accuracy is plotted. While there is significant improvement as the number of value histogram buckets increase, there is no such improvement when the number of structural histogram buckets increase (as already indicated from Figure 4.15).

Figure 4.17 shows the top 30% and top 50% of the worst estimates as well as the overall accuracy for the 100 structural histogram bucket case. In addition to improved estimation accuracy as the number of value histogram buckets increases, the "TOP 30%" curve falls faster as the the number of value histogram buckets increases. This shows that for the D-Summary, the main bottleneck is in coming up with effective value histograms. In contrast, in the N-Summary, both value as well as structural histograms have the potential to make significant improvements in the estimation accuracy.

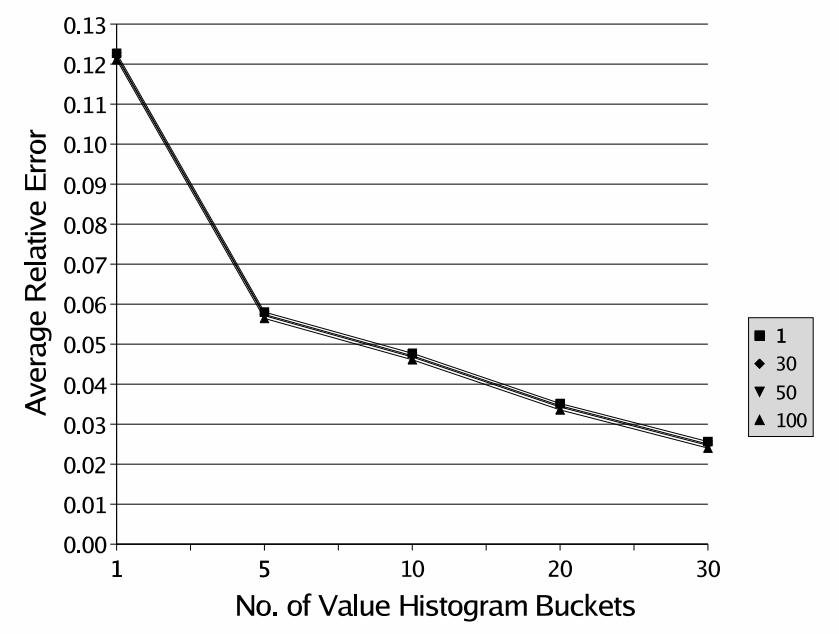


Figure 4.16: **IMDB:** Estimation Accuracy for VP Queries over D-Summary with Increasing Value Histogram Buckets

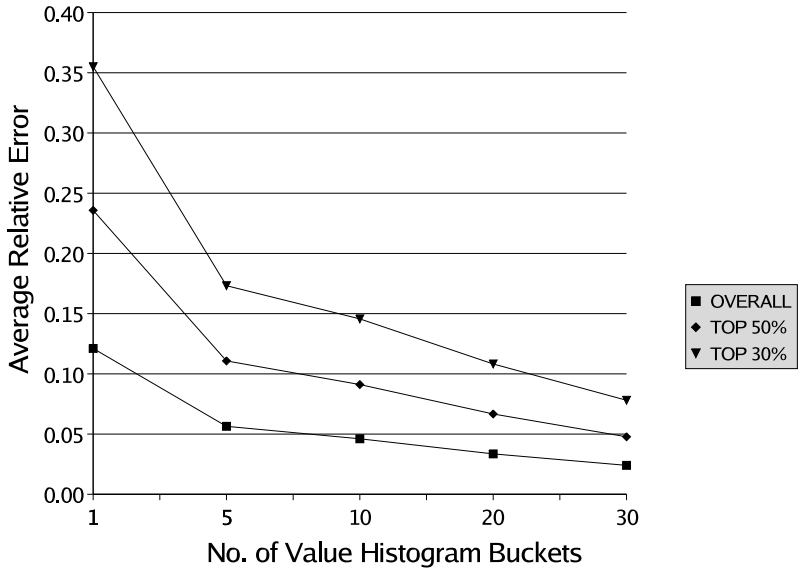


Figure 4.17: **IMDB:** Estimation Accuracy for VP Queries over D-Summary with Increasing Value Histogram Buckets and 100 Structural Histogram Buckets

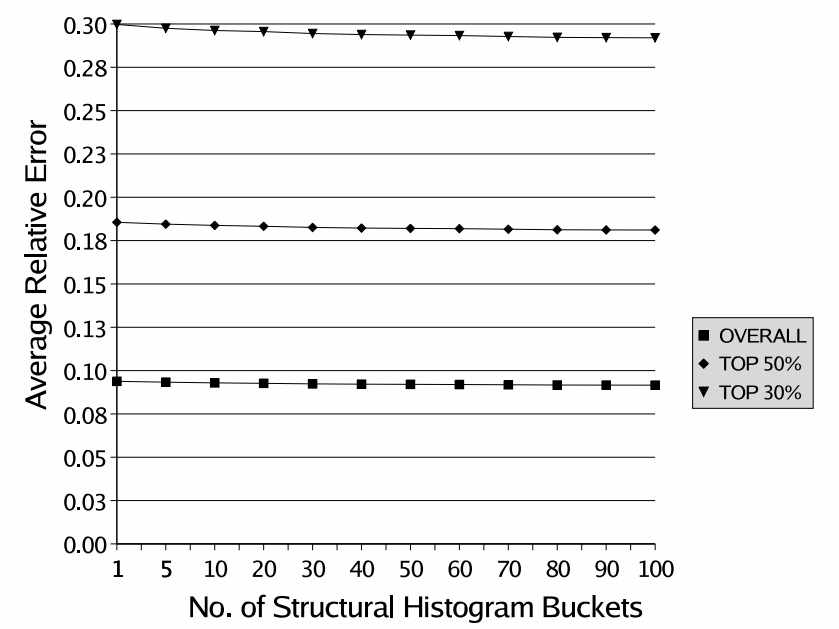


Figure 4.18: **DBLP:** Estimation Accuracy for VP Queries over D-Summary with 30 Value Histogram Buckets

Similar results were found for the DBLP dataset as well. Figure 4.18 shows the estimation accuracy when the number of value histogram buckets is kept constant at 30. Again, as indicated from the results for the BP queries, increase in the number of structural histogram buckets does not have a significant impact on the quality of the estimates.

Figure 4.19 shows the estimation accuracy when the number of value histogram buckets increases. Again, the improvement in estimation accuracy is significant. Even for the top 30% of the worst estimates, the average relative error decreases from over 150% when there is no value histogram, to less than 30% with 30 value histogram buckets. The overall estimation accuracy for different values of structural histogram follows very closely the same curve as that shown in Figure 4.19 and is not shown.

Across the board, the estimation accuracy provided by the D-Summary is extremely high, ranging from 100% accuracy for BP queries to less than 10% error for VP queries. This holds for *both* the IMDB as well as the DBLP datasets.

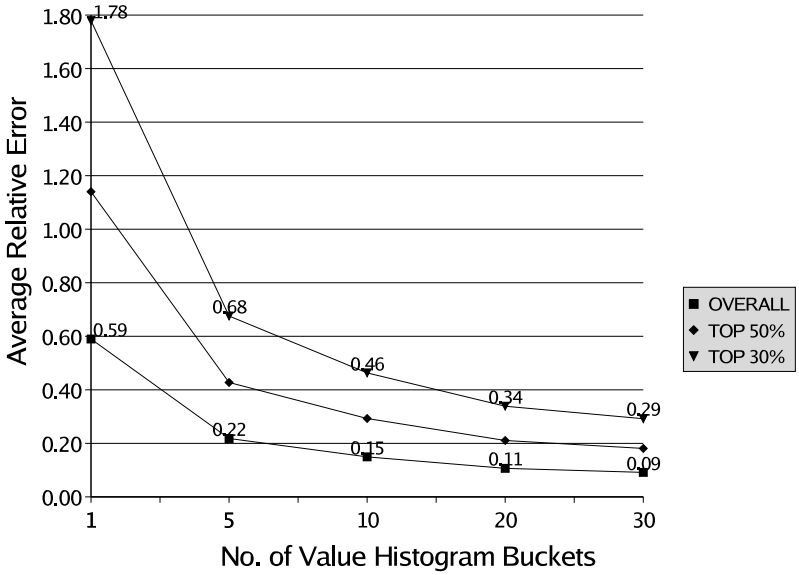


Figure 4.19: **DBLP:** Estimation Accuracy for VP Queries over D-Summary with Increasing Value Histogram Buckets and 100 Structural Histogram Buckets

4.7.2 Size of the Summary

The previous section discussed the estimation accuracy of the N- and D-summaries for both the BP and VP workload. In this section we consider the sizes of the summaries from two different viewpoints: (i) The estimation accuracy of both summaries when they are given the *same memory budget* and, (ii) The absolute sizes of the summaries with and without the compression technique.

Estimation Accuracy with Equivalent Memory Budgets

Clearly the D-Summary is a lot bigger than the N-Summary because of the larger number of types. In this section we study the behaviour of the two summaries when they are allocated the *same memory budget*.

Table 4.4 tabulates the "equivalent" number of buckets allocated for the two summaries for both the DBLP and IMDB datasets. These equivalences were derived by first counting the total number of buckets allocated to the structural histograms in the D-Summary and then dividing the total number by the number of *types* in the N-Schema – this number gives the number of buckets to be allocated to each type in the N-Schema. A similar procedure

IMDB				**DBLP**			
Structural		*Value*		*Structural*		*Value*	
D	**N**	**D**	**N**	**D**	**N**	**D**	**N**
1	15	1	15	1	10	1	10
50	550	5	60	50	135	5	25
100	1000	10	120	100	215	10	40
		20	225			20	65
		30	325			30	80
						40	100
						50	115
						60	130
						70	140

Table 4.4: Equivalent Number of Buckets

was used to calculate the equivalent number of value histogram buckets. For example, in the case of IMDB, allocating a single bucket per structural histogram in the D-Summary translates to allocating 15 buckets per structural histogram in the N-Summary.

Figures 4.20 and 4.21 show the estimation accuracy for the IMDB and DBLP summaries with equivalent number of buckets for the VP workload. The number of structural histogram buckets in both cases is the equivalent of 100 buckets in the D-Summary – that is, 1000 buckets for IMDB and 215 buckets for DBLP for the N-Summary.

Clearly, in the case of IMDB, the D-Summary is far superior to the N-Summary. This result is expected since we previously noted from Figure 4.8 that the number of structural histograms made a significant difference to the estimation accuracy. And so, the *structure component* of the dataset contributes significantly to the estimation accuracy. Since the D-Summary *improves* the structure of the summary, it is significantly more accurate than the N-Summary. This result shows that in order to increase the estimation accuracy, simply increasing the number of structural histogram buckets may be far less effective as compared to performing the schema transformations.

In contrast, for the DBLP dataset, the results are a bit mixed. Previously, from Figure 4.9, we saw that the structure component was much less important. This fact is clearly reflected in Figure 4.21 where the dominating effect is due to the number of

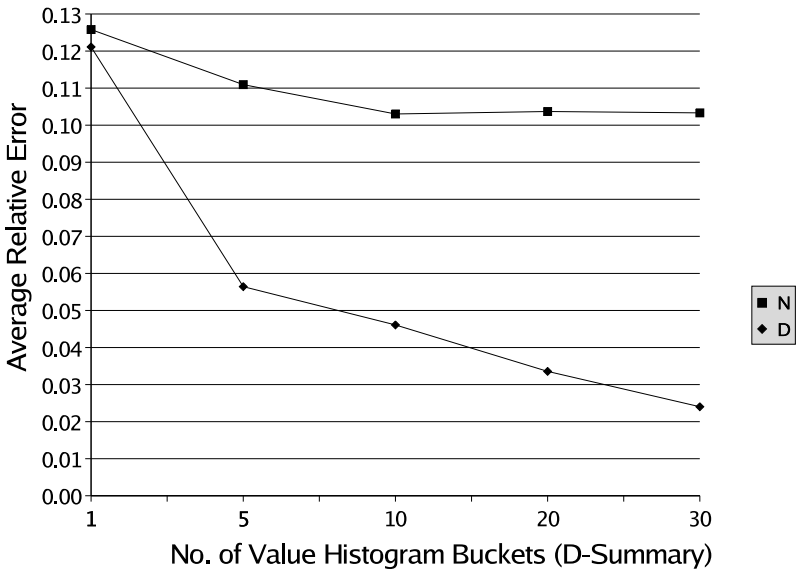


Figure 4.20: **IMDB:** Estimation Accuracy for VP Queries with Equivalent Number of Buckets

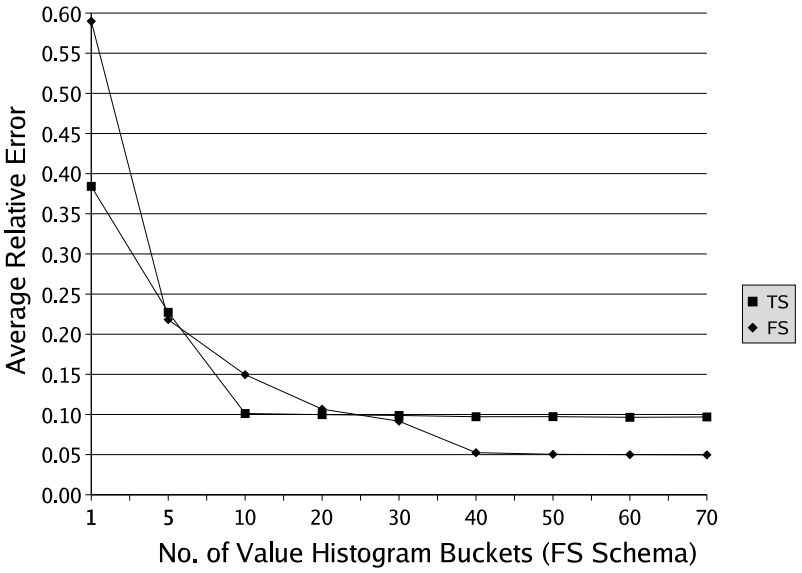


Figure 4.21: **DBLP:** Estimation Accuracy for VP Queries with Equivalent Number of Buckets

value histograms. When the D-Summary has just a single value histogram bucket, the N-Summary has 10 buckets which captures a reasonable amount of skew in the values leading to a 50% improvement in the estimation accuracy. However, as the number of value histograms is increased, the D-Summary starts to perform better than the N-Summary. The conclusion that can be drawn from these two graphs is that when there is very little structural skew, the effectiveness of performing the schema transformations is limited.

Absolute Summary Sizes

Moving away from *equivalent* memory budgets for the N- and D-summaries, we now turn our attention to their absolute sizes, given a fixed number of structural and value histogram buckets. The summary size depends on the schema as well as the data, while the *maximum* summary size depends only on the schema. Clearly, performing schema transformations increases the number of types and consequently, the number of histograms that need to be stored. However, even though the schema has several types, not all of them may be actually instantiated in the data. For example, in the case of IMDB, if the document does not contain any `MOVIE` shows, then the type Movie and all its children would not have any histograms associated with them. For the datasets used in our experiments, we tabulate both the *maximum* size of the summary as well as the *actual* size for the specific documents we used.

	No. of types (structural)	**No. of types** (value)	**Maximum Size** (100,30) bkts	**Actual Size** (100,30) bkts
IMDB				
N	32	22	45.2 KB	41.5 KB
D	409	280	577.7 KB	457.3 KB
DBLP				
N	58	50	85.5 KB	41.6 KB
D	3321	2982	4940.1 KB	186.8 KB

Table 4.5: IMDB and DBLP: Absolute Sizes of the Summaries

Table 4.5 tabulates the number of types for both the N- and D-summaries. The

number of types which contain structural histograms only are tabulated separately. Each histogram bucket is assumed to require 3 integers with 4 bytes per integer – two integers to store the bucket boundaries and one to store the count. The actual N-summary sizes are less than 50 KB for both IMDB and DBLP, while the D-summary sizes are less than 500KB. An interesting observation is that the maximum summary size is larger than the actual summary size for both N- and D-summaries. In fact, for the D-summary of DBLP, there is an order of magnitude difference between the two. This difference is due two reasons:

- The N- or D-Schema could contain types which are *not instantiated in the data*. Consider the DBLP schema which contains a lot of optional and repeated elements *in sequence*. For example, the type ARTICLE contains AUTHOR*, EDITOR*, MONTH?, PUBLISHER? and URL? in sequence. When constructing the D-schema, these repetitions are converted to unions and the unions are all distributed. This leads to 2^5 different types of ARTICLEs. That is, articles with none of these sub-elements, articles with all these sub-elements, articles with at least one author but none of the other sub-elements, articles with at least author and one editor, etc. The given data *does not* contain all these different types of articles and hence a lot of these types are not utilized in the summary. And so, the actual summary size is much less than the maximum summary size.

- Not all the histogram buckets may be utilized. For example, in the IMDB data, the Rating type can contain a total of only 10 values (1 to 10). And so, even though 30 value histogram buckets are allocated, only 10 of those buckets are utilized. Also, with a large number of types which are optional, not all the structural histogram buckets may be utilized. For example, in the DBLP data, even though there are more than 500 ARTICLEs, only 30 of them contain CDROMs.

Compressing the Number of Types. Eliminating the structural histograms for child types which have a one-to-one correspondence with their parents (that is, those types which occur exactly once under their parent) considerably reduces the number of types,

	No. of types (structural)	**No. of types after compression**	**Maximum Size** (100,30) bkts	**Maximum Size after compression** (100,30) bkts	**Savings**
IMDB					
N	32	11	45.2 KB	20.6 KB	54.4%
D	409	113	577.7 KB	230.8 KB	60.0%
DBLP					
N	58	16	85.5 KB	36.3 KB	57.5%
D	3321	1458	4940.1 KB	2756.9 KB	44.1%

Table 4.6: IMDB and DBLP: Savings with Compression

and consequently, the memory budget. Table 4.6 tabulates the savings gained by reducing the number of structural histograms. The results show that the savings are more than 40% for both the N- and D-summaries.

4.7.3 Statistics Collection Overheads

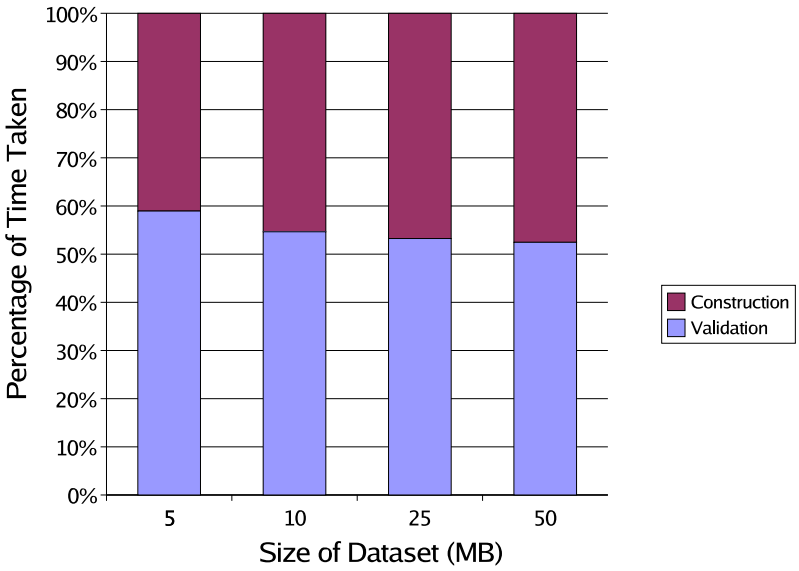


Figure 4.22: IMDB: Efficiency of Statistics Collection for the N-Schema

The efficiency of statistics gathering depends on: (i) parsing and validation of the XML data file (Validation) and, (ii) construction of the summary structure (Construction). We

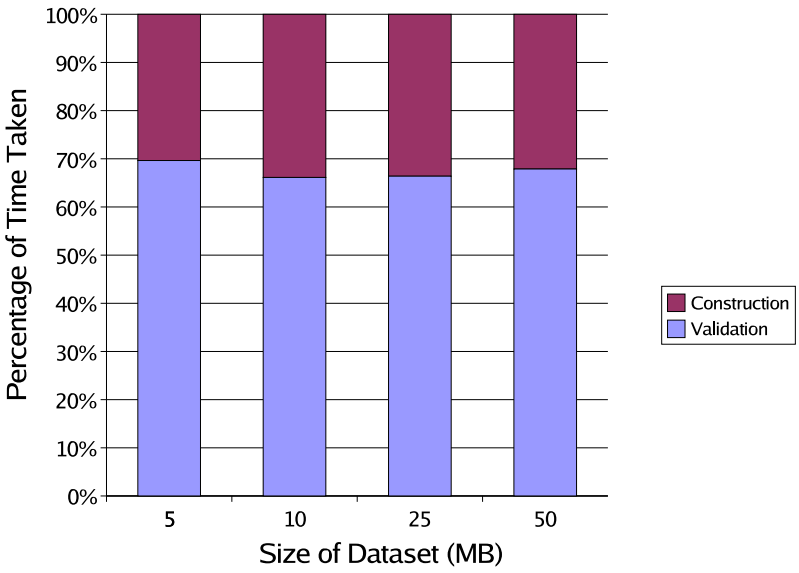


Figure 4.23: IMDB: Efficiency of Statistics Collection for the D-Schema

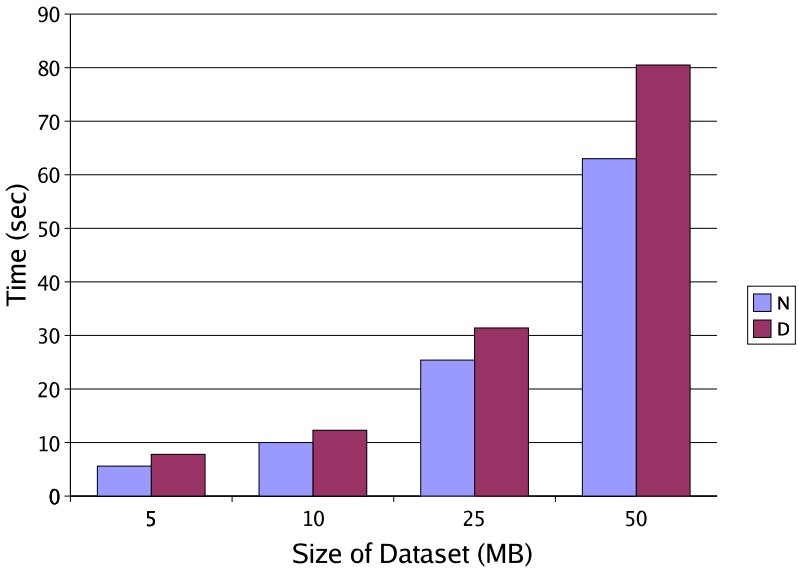


Figure 4.24: **IMDB:** Comparison of Validation Times for the N- and D-Schemas

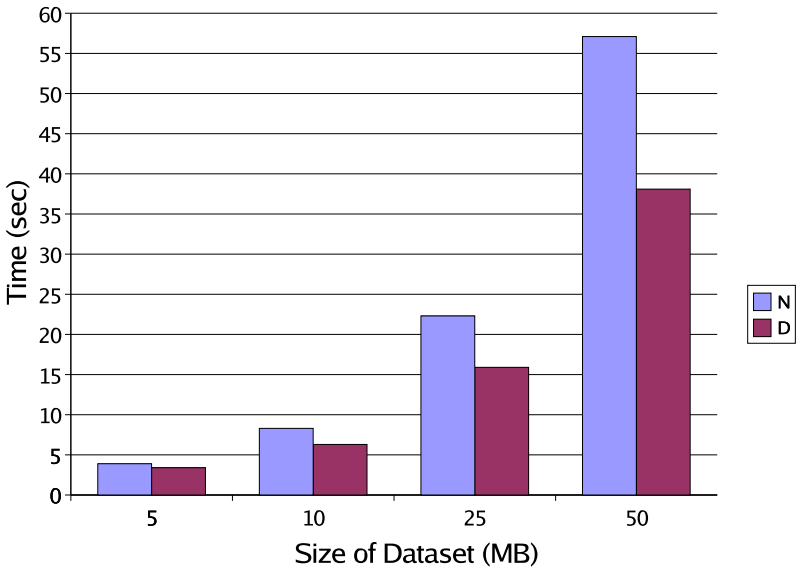


Figure 4.25: **IMDB:** Comparison of Summary Construction Times for the N- and D-Schemas

tabulate the relative time taken by each of these phases for the IMDB N-Schema and the IMDB D-Schema for different sizes of datasets. The results are shown in Figures 4.22 and 4.23. Clearly, the results indicate that the major part of the time is taken up by the parsing/validation part of the statistics gathering (over 50% for the N-Schema and over 60% for the D-Schema). An interesting result in this regard is that it takes *less* percentage of time for statistics construction in the case of D-Schema inspite of the larger number of types than in the case of the N-Schema. The absolute times for validation and construction, shown in Figures 4.24 and 4.25 respectively, indicate that it takes *longer* for validating against the D-Schema, but the time for construction is much *shorter* than for the N-Schema. This can be explained by the fact that in the case of the N-Schema, there may be less number of types, but the size of the value and parent-id lists are much longer. Since we need to sort these lists in order to construct the equi-depth histograms, it takes longer. On the other hand, for the D-Schema, the sorting takes place for much shorter lists, though the number of such lists is larger.

4.8 Conclusions

In this chapter, we introduced StatiX, a framework for statistics collection and cardinality estimation. The main features of StatiX include: (i) the use of XML Schema as the basis for statistics collection, (ii) the use of several schema transformations to improve the accuracy of the summary, (iii) use of histograms to symmetrically capture both structural and value distributions, (iv) support for cardinality estimation of branching path expression queries with value predicates, and (v) the ability to use standard XML Schema validators where possible, for statistics collection.

Our experimental evaluation showed that the D-Summary could achieve 100% accuracy for branching path expressions without value predicates, while the N-Summary could improve the estimation accuracy by increasing the number of histogram buckets. However, increasing the number of histogram buckets was effective only when there was a reasonable amount of skew in the data as in the case of the IMDB dataset. For branching path expressions *with* value predicates, the number of *value histogram buckets* made the largest difference in improving the estimation accuracy, while the number of structural histogram buckets made a limited impact (this was especially noticeable in the DBLP dataset which did not contain much structural skew).

The absolute amount of memory used for the N-summary was less than 50KB, while for the D-summary, it was less than 300 KB for DBLP and less than 500KB for IMDB. Our compression technique resulted in size reductions of 40%-60% on the maximum summary size.

Experiments on estimation accuracy with *equivalent* memory budgets for both the N- and D- summaries indicated that performing schema transformations is a more effective method of improving accuracy as compared to increasing the number of histogram buckets.

With regard to the efficiency of statistics collection, our results indicate that parsing and validation of the XML data takes up the major portion of the time as compared to the actual construction of the histograms. In fact, upto 70% of the total statistics collection time devoted was to validation.

Chapter 5

Incremental Maintenance of XML Summaries

5.1 Introduction

An increasing number of XML applications are *dynamic* and frequently update the underlying data. This gives rise to the problem of *maintaining* the statistics in the presence of these updates. Periodically recomputing the statistics from scratch is a possible solution, but suffers from two problems: (i) Recomputation is an expensive process since it involves the parsing of the *entire* document, and (ii) Improperly timed recomputations could result in stale summaries, leading to unacceptable estimation errors. In this chapter, we present new techniques to incrementally update XML statistical summaries *in parallel* with the receipt of document updates. We assume that an accurate summary of the data, created at the document loading time, is initially made available, and then, as and when updates are received, this summary is also correspondingly updated. Specifically, given an initial document D and its summary S, and a stream of updates $U = U_1, U_2, \ldots, U_m$ comprising of inserts, deletes or modifications, the goal is to efficiently and incrementally create summaries, $S^1, S^2, \ldots, S^m$, such that the accuracy of these summaries are comparable to those obtained with a recomputed-from-scratch summary $S_R^1, S_R^2, \ldots, S_R^m$. Moreover, this should be achieved within a *fixed memory budget* (that is, the incremental approach has

the same resource constraints as recomputation).

Our solution to the XML statistics maintenance problem is an algorithm called **IMAX** (Incremental MAintenance of XML statistics). IMAX is built around the StatiX framework (Chapter 4), which not only produces concise and accurate summaries for XML documents, but also has several features that make it attractive in a dynamic scenario. For example, StatiX captures order information among the elements in a document through the document schema and its numbering scheme (see Chapter 4 for details). This information makes it possible to estimate the *location* of updates – a key step in IMAX. In addition, its use of histograms permits the re-use of well-known techniques for incremental histogram maintenance.

An important extension that we make to the StatiX framework is the use of *two-dimensional* value histograms (instead of the originally proposed 1D histograms) to capture the correspondence between the node ids and their values. The use of 2D histograms is a key factor in the effectiveness of IMAX. An empirical evaluation of IMAX (with both 1D as well as 2D histograms) over a variety of XML documents and update streams demonstrates that IMAX provides, at a marginal run-time cost, accuracy comparable to the brute-force recomputation approach, even with a fixed memory budget.

Organization. The rest of this chapter is organized as follows. In Section 5.2, we highlight several issues which arise in the maintenance of XML statistics with particular reference to the StatiX framework. In Section 5.3, we describe IMAX, our solution to the statistics maintenance problem. In Section 5.4, we present an experimental evaluation of IMAX. Finally, in Section 5.5, we conclude the chapter.

5.2 Issues in Updating StatiX Summaries

Given an update query, it is important to know both *how many* updates will be applied and also *where* they will be applied. The importance of knowing the locations of the updates stems from the fact that structural histograms capture the relative distributions of children with respect to their parents. Hence, if the correct ids of the updated components

can be computed, the appropriate buckets of the histogram can be updated. In the case of XML updates there is always an implicit *location* component to the update. For example, consider the following insertion (using the syntax of [39]):

Example 1 Add a `REVIEW` element to the `SHOW` with title "The sixth sense".

```
update
insert <REVIEW>
         <RATING>Top drawer stuff!</RATING>
       </REVIEW>
into  //SHOW[TITLE="The sixth sense"]
```

□

Here, the path expression: `//SHOW[TITLE="The sixth sense"]` describes the particular Show at which the update applies. Inherently, there is an ordinal associated with this `SHOW`, which is critical in updating the summary. Moreover, the ordinal of `SHOW` determines the ordinals of the other elements in the update fragment. For example, for the above update query, in the parent histogram of Review, the count of the bucket which contains the Show id of "The sixth sense" needs to be incremented; and based on where the review is added, the parent histogram of Rating also needs to be updated. Note that if titles are unique, there is a *single* location in the document which is updated with the given `REVIEW` fragment. However, an update can also be applied to a *set* of locations. For example, the following query inserts a new `AGE` sub-element into all movies and TV shows made prior to 1930:

Example 2 Add the `AGE` element into all shows with year less than 1930.

```
update
insert <AGE> Golden Oldie ! </AGE>
into  //SHOW[YEAR < "1930"]
```

□

5.2.1 Location and Cardinality Estimation

It is possible to rely on the actual update operation to determine the number and location of updates – the database can provide this information to the estimator module. Recall, however, that the accuracy of estimation and the conciseness of summaries achieved by StatiX are largely due to *contiguous* node ids which both capture the order among elements and are effectively summarized by histograms. While such a numbering scheme is effective for StatiX, it may not be suitable for the backend database – using a contiguous node id scheme at the backend could lead to unacceptable update performance, since it may require a large number of elements to be renumbered [15, 67, 74]. Therefore, instead of relying on a translation mechanism between the contiguous node id scheme required by StatiX, and the many possible id schemes at the backend, we make update maintenance self-sufficient by estimating both the *cardinality* and *location* of the updates.

5.2.2 Updates to Structure and Value Histograms

Another important difference to note in the case of updating StatiX summaries is the nature of the histograms being updated. Previously proposed techniques for histogram maintenance (*e.g.,* [30]) were designed for *value histograms*, not structural histograms. There are important differences between a structural histogram and a value histogram. First, there is no sanctity to the values in a structural histogram – structural histograms are based on node ids, but the specific value of the node id is not relevant as long as the histogram correctly captures the parent-child distribution. For example, it does not make a difference whether a sequence of Shows is numbered from 1 to 10 or from 100 to 110, as long as the parent histograms of its children use the same values. Second, the term "insertion" in the case of value histograms and structural histograms take on different meanings. In the case of insertion into a value histogram, the count of the corresponding value is updated. However, in the case of structural histograms, a "new" value is inserted and the subsequent values renumbered. For example, if a new `REVIEW` is inserted between `REVIEW` 2 and `REVIEW` 3, the id of the new `REVIEW` is set to 3, and the ids of the subsequent reviews are incremented. Thus, the *domain of the values* in a structural

histogram continuously changes, and this change in ordinals affects the bucket boundaries of all the parent histograms for the children of type Review as well.

5.3 The IMAX Technique

In this section we introduce our techniques for maintaining statistics in an XML document in the presence of insertions and deletions of tree fragments. We restrict our attention to the class of updates where the location of the update is determined through branching path expressions in the query.

A high-level description of IMAX is provided in Algorithm 2. It consists of three main steps: location estimation; id estimation; and summary update. These steps are described in detail in the remainder of this section.

Input: *Summary* $\mathcal{S}$, *Update* $\mathcal{U} = (c, u)$
$\mathcal{S}$ *is the initial summary;* $\mathcal{U}$ *is divided into condition* c*, and update fragment* u
Output: *UpdatedSummary* $\mathcal{S}'$
1: **Estimate the location of update using** c **and** $\mathcal{S}$
2: **Estimate the ids of update fragment** u **using** $\mathcal{S}$
3: **Update** $\mathcal{S}$

Algorithm 2: IMAX Algorithm

5.3.1 Estimating the Location of the Update

Given the branching path predicate for the update location, IMAX needs to estimate the cardinalities of these updates, as well as the ids of the nodes where the updates takes place. Estimating the location of the updates is closely tied to the cardinality estimation. As previously mentioned in Section 4.2, each type can be thought of as having a trivial one-bucket *key histogram* whose end points are the range of ids of the type, and whose frequency is the cardinality of the type. As we explain below, we utilize this key histogram and the parent histogram associated with each type to perform cardinality and location estimates. A high-level description of the procedure is shown in Algorithm 3. Note that a major part of this procedure as well as the explanation below is reproduced from Chapter

4. In the interests of completeness, we repeat the salient points here.

Input: c, $\mathcal{H}$
c is the path expression identifying the location
$\mathcal{H}$ is the set of histograms (value and structure) for all types corresponding to the elements in c
Output: Cardinality and Location Ids of the Updates
1: let $c = /t_1[b_1]/t_2[b_2]/t_3[b_3]/.../t_n[b_n]$
{t_i is the tag (correspondingly, its type is T_i)}
2: **for all** $i \in 1$ to n **do**
3: B_i = result distribution of b_i
4: $J_i = B_i \bowtie keyHist(T_i)$
5: $keyHist(T_i)$ = key distribution of T_i based on J_i
6: $parentHist(T_i)$ = compute distribution based on $keyHist(T_i)$
7: **end for**
8: **for all** $i \in 1$ to $n-1$ **do**
9: $J_i = keyHist(T_i) \bowtie parentHist(T_{i+1})$
10: $keyHist(T_{i+1})$ = distribute $freq(J_i)$ into $keyHist(T_{i+1})$
11: **end for**
{Cardinality of the update}
12: $card$ = frequency (J_n)
{We now compute the location ids}
13: $locations$ = randomly choose $card$ ids from the buckets of $keyHist(T_n)$ in proportion to their frequency

Algorithm 3: Location and Cardinality Estimation for the Updates

This procedure operates in three stages: (i) compute the key distribution and parent-key distribution for each of the t_is in the presence of predicates *individually* (lines 2 through 7); (ii) use these individual distributions to compute the overall key distribution of the complete query (lines 8 through 11); and finally (iii) estimate the cardinality and the location of the updates (lines 12,13).

There are three basic operations – histogram multiplication (lines 4 and 9), finding the key distribution (line 5), and finding the parent key distribution (line 6). Histogram multiplication is a well-known operation to find the join estimate given two histograms. For a description of how the key and parent key distributions are computed, please refer to Section 4.3 in Chapter 4.

Choosing the ids

By performing the steps in Algorithm 3, we get the key distribution of the *result* of the query (that is, the key distribution corresponding to t_n, shown in line 1). Computing the actual location ids is now a matter of choosing the ids from this key histogram. The ids are chosen from the buckets of the key histogram *in proportion to their counts*. For example, suppose the final key distribution of Show from the previous update is: `[1-12: 1; 12-25: 1]`. We randomly choose 1 Show id from 1 to 11 and 1 id from 12 to 24 – these choices comprise the statistically determined Show ids where the updates of the `REVIEW` fragment takes place.

Improved Location Estimation

A potential limitation in the current location estimation process is the use of single dimensional histograms for values. The problem stems from the fact that *no correspondence* between the occurrence of a value and the id of the node at which it occurs, is stored, as in the case of structural histograms. Consequently, we have to make the independence assumption when computing the distribution of the nodes containing particular values – that is, distribute the estimated cardinality into the parent histogram in proportion to the bucket counts. For example, consider the type Year with values ranging from 1900 to 1960. Suppose the key histogram of year is as follows (note that the key histogram has been arbitrarily made into a two-bucket histogram).

`[1-12: 12; 12-30: 18]`

Now, let the value histogram of Year have the skew as shown below:

`[1900-1912:7; 1912-1924:7; 1924-1931:6; 1931-1945:6; 1945-1960:4]`

Suppose we were to estimate the location of the following location condition:

//`SHOW`[`YEAR` <= “1930”]

There are 20 `YEAR`s with value less than or equal to 1930 and their key distribution would be evenly distributed across the range of key ids of Year (and consequently, into the parent histogram of Year which is to be multiplied by the key histogram of Show). But, if all the relevant Year ids were in the range of, say, [12-30), then the location estimate has

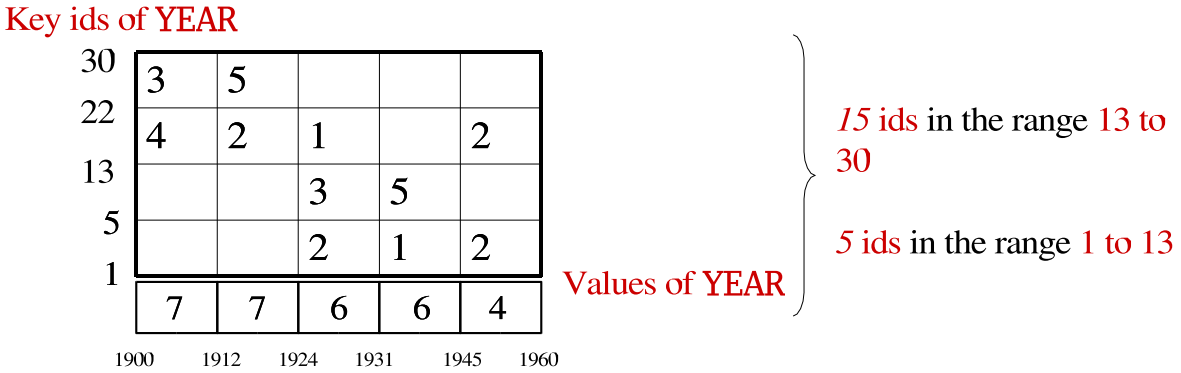


Figure 5.1: 2D Histogram to Capture Correlation Between Year values and Year Ids

a large error – that is, it estimates 8 of the ids from the wrong range.

In order to overcome this limitation, we propose the use of 2D histograms to explicitly capture the correspondence between values and the corresponding node ids. For the previous example, suppose we constructed the histogram shown in Figure 5.1[1]. Then the location estimation process would accurately estimate that there are 5 ids in the range 1 to 13 and 15 ids in the range 13 to 30.

Since 2D histograms require more space, the budget for value histograms must be increased to improve the accuracy. However, as we show in Section 5.4, the advantages of using 2D histograms are substantial. We use the algorithm proposed in [44] to build equi-depth 2D histograms by choosing one axis at a time. We chose the *key dimension* as the first dimension – the key dimension is *contiguous* and hence will lead to histogram buckets which are well packed in that dimension.

5.3.2 Estimating the Ids of the Update Fragment

Once the locations of the update is determined, we next need to estimate the ids of the elements in the update fragment. In the case of insertions, the update fragment is explicitly given in the query and for each insertion, the *number* of elements being inserted is known, while the *ids* of these elements have to be estimated. But, in the case of

[1]Note that the 2D histogram could have a more complex bucket structure than the simple square buckets shown in Figure 5.1.

Input: $parentHist_{child}$, id_{parent}
The parent histogram of the child element and the id of the parent are the inputs
Output: id_{child}
1: $id_{child} = 0$
2: $B_k \in parentHist_{child}$ such that $id_{parent} \in B_k$
3: **for all** $i \in 1$ to $k-1$ **do**
4: $\quad id_{child} +=$ frequency of B_i
5: **end for**
6: $id_{child} += \lfloor freq(B_k)/range(B_k) * (id_{parent} - lowerbound(B_k)) \rfloor + 1$

Algorithm 4: Estimating Ids

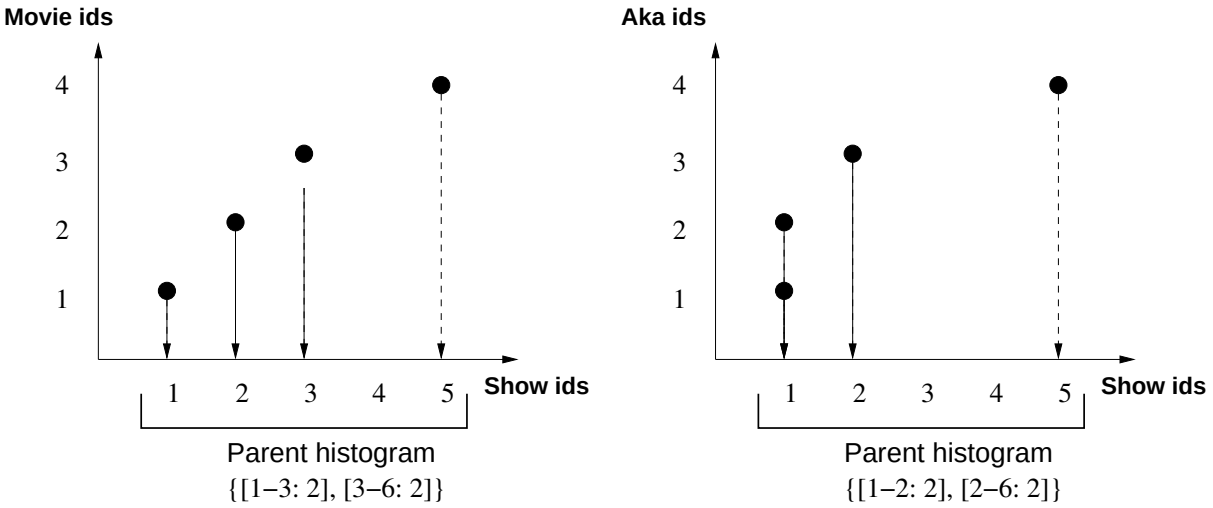


Figure 5.2: Node and Parent ids have a Correspondence

deletions, only the root of the subtree to be deleted is given, so the number as well as the ids of the deleted elements in the subtree need to be estimated.

In order to estimate the ids of the update fragment, we use the parent histogram which summarizes the correspondence between parent and child ids (Figure 5.2).

Estimating ids for insertions

Algorithm 4 describes how the parent histogram is used to estimate the id of a child fragment. The algorithm outputs a single child id. If there are multiple children in the update with the same tag, then a set of *contiguous ids* are assigned beginning from the estimated id of the first child (as determined by Algorithm 4). For Example 1, let the key distribution of Show be computed as:

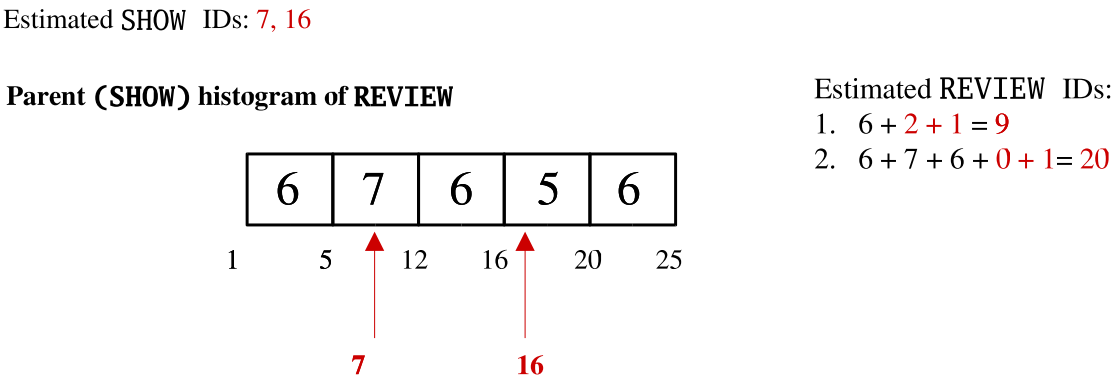


Figure 5.3: Computing the Ids of `REVIEW`

```
[1-12:  1; 12-25:  1]
```

Suppose the actual ids chosen were 7 and 16, then, the insertion ids for `REVIEW` would be computed as shown in Figure 5.3.

Estimating ids for deletions

In the case of deletions, only the root node of the subtree to be deleted is given. The elements in this subtree have to be first determined from the schema. Since the id of the root node of the deletion is known, Algorithm 4 can be used to estimate the id of the child. In addition, the frequency of the child can be estimated from B_k (line 2 in Algorithm 4) by dividing the frequency of B_k by the range of B_k.

5.3.3 Updating the Summary

The relevant parent histograms in the summary need to be updated by either inserting new ids or deleting them. This includes not only the parent histograms of the types in the update fragment, but also the children of these types which may not be present in the update fragment. As mentioned in Section 5.2, when a structural histogram is updated, not only does the count of the bucket increase, but the *subsequent* ordinals may have to be renumbered. An example of such an insertion is shown in Figure 5.4.

However, a large number of insertions or deletions to the histogram may make it inaccurate. For example, if new documents are appended continuously, then clearly, only

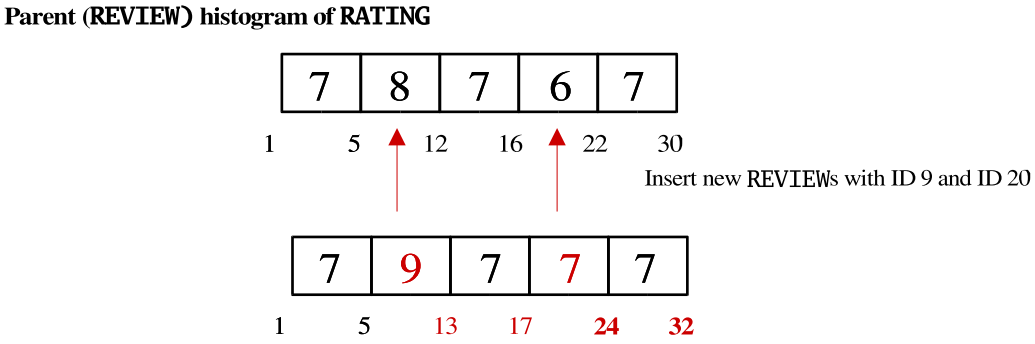


Figure 5.4: Inserting Ids into the Parent Histogram of `RATING`

the last bucket of a histogram is updated each time with new ids. Therefore, while the last bucket keeps accumulating counts eventually making it inaccurate, the remaining buckets retain their original counts. One strategy to approximately maintain the equi-depth histogram is to periodically *redistribute* the bucket counts by splitting a bucket once its count reaches a threshold occupancy T into two new buckets, and then simultaneously merging a pair of buckets whose combined count is less than T [30]. If more than one such pair exists, then the pair whose combined frequency is the least is chosen. If such a pair of *split-merge* operations cannot be performed, then the histogram is *recomputed* from the base data. Note that this procedure for determining whether to recompute the histogram from base data is very *conservative*, since the criterion for recomputation is whether or not there is a split-merge pair available. Other techniques, such as, testing whether the *current* histogram is equi-depth, before performing a recomputation, could potentially *reduce* the number of recomputations. In this thesis, we evaluate the conservative approach.

Algorithm 5 highlights the main steps in inserting a new value into a parent histogram. The input to the algorithm is the pair (id, f). Note that the id in this case is the id of the *parent*, while the histogram being updated is the parent histogram of the *child*. The pair (id, f) indicates the number of times, f, the given child occurs under the given parent with id id. Steps 3 to 7 perform a *shift* operation to indicate the insertion of a new id – this is equivalent to renumbering the previous ordinals of the elements due to the insertion of a new one. Steps 8 to 15 determine whether only a *reorganization* will suffice or whether a complete *recomputation* of the histogram from the base data needs to take

Input: $Histogram : H$, $Update : (id, f)$, $Threshold : T$
H is the histogram to be updated
(id, f) is the update consisting of new (id, frequency) pair
T is threshold occupancy at which a bucket is split
Output: $UpdatedHistogram : H'$

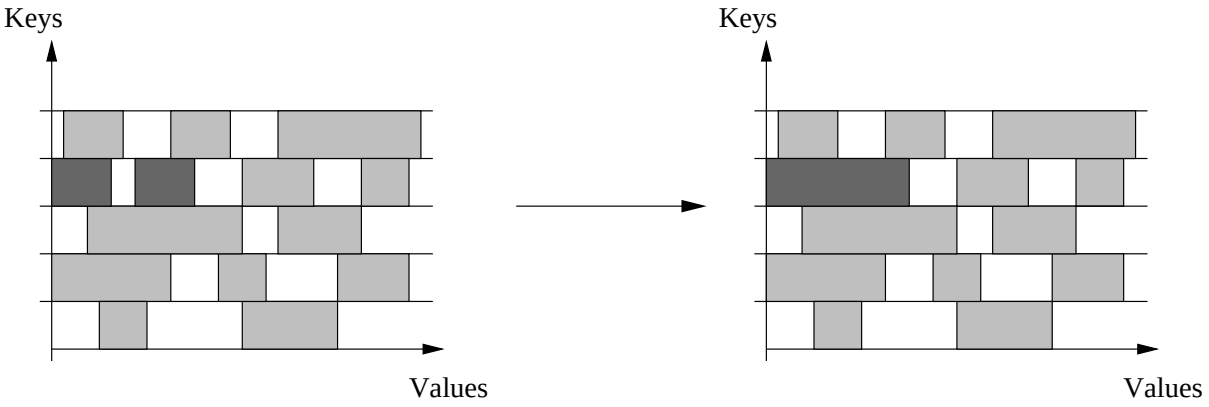
1: $B_k \in H$ such that $id \in B_k$
{Update the frequency of the bucket}
2: $B_k.frequency += f$
{Update bucket's upper limit to reflect *insertion* of new id}
3: $B_k.hi = B_k.hi + 1$
{n is the number of buckets in H}
{Update the boundaries of remaining buckets}
4: **for all** $i \in k+1$ to n **do**
5: $B_i.lo = B_i.lo + 1$
6: $B_i.hi = B_i.hi + 1$
7: **end for**
8: **if** $B_k.frequency >= T$ **then**
9: found = find B_i, B_{i+1} in H such that $B_i.frequency + B_{i+1}.frequency < T$
10: **if** found **then**
11: **REORGANIZE** H merging B_i, B_{i+1} and splitting B_k
12: **else**
13: **RECOMPUTE** H from base data
14: **end if**
15: **end if**

Algorithm 5: Insertion of a new id into a parent histogram

place.

For deletions, instead of ids being "inserted", the ids need to be deleted. Similar issues also arise for deletions – that is, a single bucket may have a very small count compared to the others. The strategies outlined for insertions can be easily modified to handle deletions as well.

Maintaining 2D Histograms. A split-merge strategy with a threshold T is used to maintain the 2D histograms as well. However, unlike the strategy for 1D histograms, merge pairs are always chosen such that the bucket boundaries in the key axis match and the split happens only along the values axis – this ensures that the buckets remain rectangular even after merging. Figure 5.5(a) shows a 2D histogram with the key axis being chosen first during construction. Two buckets in Figure 5.5(a) are selected for



(a) A 2D Histogram (b) Merged buckets

Figure 5.5: 2D Histograms - Construction and Merge

merging (darkened rectangles) – both buckets have the same boundaries on the key axis. Figure 5.5(b) shows the histogram after the buckets have been merged.

5.4 Experimental Evaluation

5.4.1 Experimental Setup

We carried out a detailed evaluation of the IMAX approach on synthetically generated IMDB data and also on a subset of DBLP data available from [18]. All experiments were performed on a Compaq ES45 dual-processor machine with 1.25 GHz and 16 GB memory. For ease of presentation, we classify the types of insertions into: (i) Append only, and (ii) Random insertions.

Memory Budget

The memory budget for the summary depends on the number of types in the schema and the number of buckets allocated for structural histograms and value histograms. All experiments in this section were performed with a minimum of 5 buckets for each structural histogram and 100 buckets for each value histogram – translating to about 5KB of memory, and a maximum of 30 structural histogram buckets and 500 value histogram

buckets – translating to about 23 KB of memory. Note that the value histograms are 2D value histograms requiring 5 integers per buckets (4 integers to identify the boundaries and 1 to store the count). Also, *schema fragments* were used in the experiments, which were *smaller* than the full Schema and encompassed only the types required for the updates.

Threshold Factor

The reorganization threshold of histogram H_i is set as $T_i = t * f_i$ where f_i is the equi-depth bucket frequency of histogram H_i, and t is a user-specified *threshold factor*. In our experiments, the threshold factor was set to 2.5.

Metrics

Our primary performance metric is to compare how close the IMAX incrementally-generated summary is with respect to the recomputed-from-scratch summary. For each affected histogram, this is quantitatively captured by μ_{mse} defined as follows:

$$\mu_{mse}(IMAX) = \frac{\sum_{i=1}^{N}(Est_{Recomputed} - Est_{IMAX})^2}{totalCardinality}$$

where $i = 1$ to N covers the total range of values in the histogram, $Est_{Recomputed}$ is the estimate of value i from the histogram computed from scratch, and Est_{IMAX} is the estimate computed from IMAX. The $totalCardinality$ refers to the overall occupancy of the histogram.

To quantitatively establish that there is indeed a significant difference between the updated document and the original document, we also compute μ_{mse} between the currently computed-from-scratch summary and the original summary (*i.e.,* before any updates were received), as shown below:

$$\mu_{mse}(ORIGINAL) = \frac{\sum_{i=1}^{N}(Est_{Recomputed} - Est_{ORIGINAL})^2}{totalCardinality}$$

While the above metrics measure the *accuracy* of IMAX in the face of significant updates, our next metric aims to measure its *efficiency*. This is done by tracking the number of recomputations incurred by IMAX during its maintenance process. This metric,

called $RECOMP$, is defined as the number of recomputations divided by the total number of insertions into the histograms, that is, $RECOMP = \frac{r}{I}$ where r is the number of recomputations and I is the total number of histogram insertions. $RECOMP$ can be calculated on a per-type basis or over all types in the insertions.

5.4.2 Append-Only Updates

Append-only updates occur in warehouse scenarios, where new documents are continuously being added. The main complexity in append-only updates is in the reorganization of the histograms since appends occur at the root of the document. For IMDB, we appended new Shows to the document, while for DBLP, we appended new ARTICLES.

Results. For the IMDB dataset, the μ_{mse} values for two types: Review and Aka are shown in Figure 5.6. Note that the histograms correspond to the *parent histograms* of these types. In this graph, the number associated with each algorithm in the legend (for example, 10 in Review(10,IMAX)) refers to the number of structural histogram buckets. Note that the number of value histogram buckets is not an issue here, since the location condition does not involve a value predicate. For the updates of value histograms, the μ_{mse} values are shown for type Year in Figure 5.7. Here the legend denotes the number of 2D value histogram buckets.

The first point to note in Figures 5.6 and 5.7 is that the μ_{mse} values (which are shown on a *log-scale*) for IMAX are very low, especially when compared with the μ_{mse} values for the original parent histogram – in fact, there is close to *two orders of magnitude difference* in their quality. This clearly indicates that (a) there is a substantial change between the original document and the updated document, and (b) IMAX is able to track these changes rather well for both the structural and the value histogram cases.

Next, the efficiency aspect is captured in the RECOMP numbers shown in Table 5.1. It shows that only a very small fraction of recomputations are required to support the IMAX incremental maintenance strategy.[2]

[2]Recomputation refers to the recomputation of the specific histogram, as mentioned in Section 5.3.3.

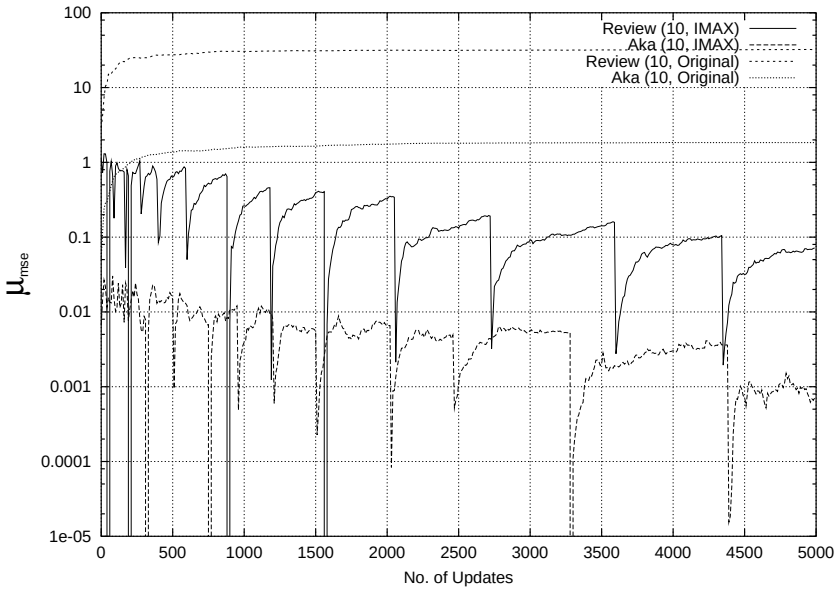


Figure 5.6: IMDB: μ_{mse} values for types Review and Aka

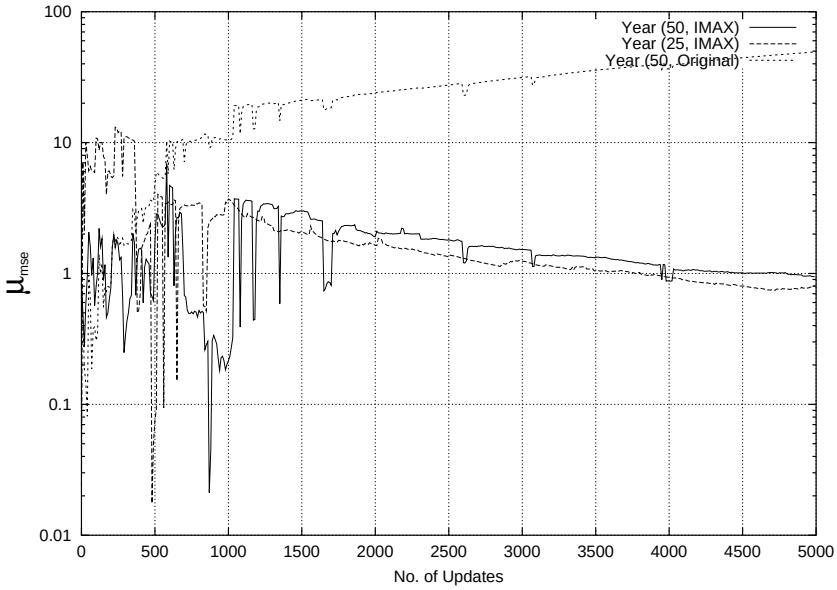


Figure 5.7: IMDB: μ_{mse} values for type Year

Type	No. of Insertions	$RECOMP$
Show	5000	0
Review	170123	0.008%
Aka	9798	0.12%
Tv	2461	0.28%
Movie	2539	0.27%
Year	5000	0.02%
TOTAL	189921	0.01%

Table 5.1: IMDB: $RECOMP$ with Appends

Type	No. of Insertions	$RECOMP$
ARTICLE	10000	0
AUTHOR	16174	0.04%
URL	9989	0.08%
TOTAL	109359	0.06%

Table 5.2: DBLP: $RECOMP$ with Appends

Similar results for the DBLP dataset are shown in Figure 5.8 and Table 5.2 for the μ_{mse} and $RECOMP$ metrics, respectively. Note that in Table 5.2, only a subset of types updated have been enumerated, while the last line totals all updated types.

5.4.3 Random Insertions

Turning our attention to random insertions, the most important component here is the location estimation. If a single update query results in updates in multiple locations, then the cardinality estimation also comes into play. We divided insertions into two categories: (i) Unique insertions, where a single update query results in an insertion at a unique location in the document, and, (ii) Multiple insertions, where a single update query results in insertions at multiple locations in the document.

For IMDB, we generated an Actor database consisting of information about actors. Each `ACTOR` subtree consists of a `NAME` sub-element, and multiple `PLAYED` sub-elements. Each `PLAYED` element may contain multiple `EPISODE` sub-elements. The update query

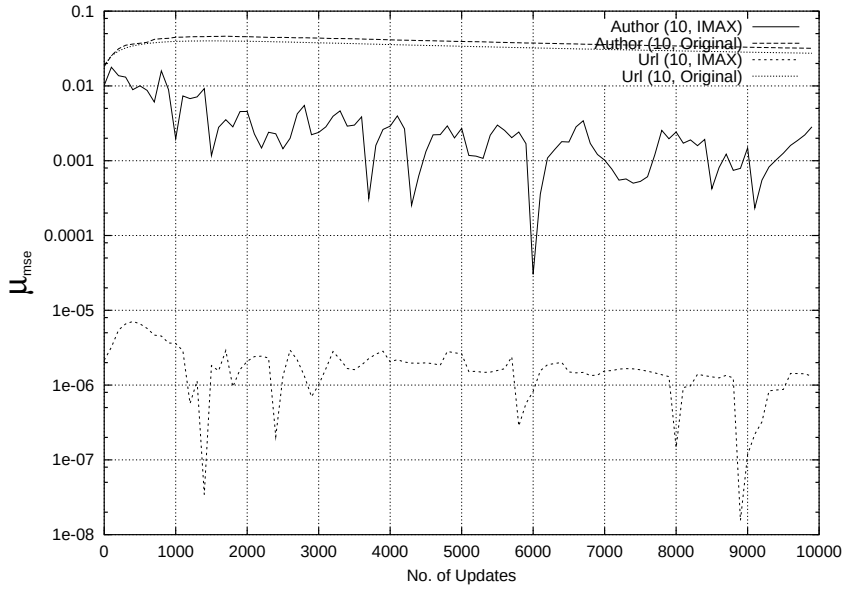


Figure 5.8: DBLP: μ_{mse} values for types Author and Url

reflects the addition of new information regarding the actor's acting history. The insertions were of the form:

```
update insert
  <PLAYED>
     <EPISODE>...</>
     <EPISODE>...</>
     ....
  </>
into /ACTOR[NAME="x"]
```

The number of Actors in the database was 1000 – that is 1000 unique values for the value predicate involving Name. Note that this query has *multiple* levels of insertions where the estimated id of Actor (from Algorithm 3) is used not only to update the parent histogram of Played, but also to estimate the id of Played (from Algorithm 4). This id in turn is used to determine the ids of the multiple Episodes.

For the DBLP dataset, we chose a set of journal articles from 134 different journals. Each journal had articles published in that journal in a separate subtree. The insertions

we chose reflects the addition of new articles into a database segregated on the basis of journal names. Each `article` had multiple `author` elements along with several other relevant information such as `url`, `publisher`, `year`, etc.

The insertions were of the form:

```
update insert
   <article>
      <author>..</>
      <author>..</>
      ...
      <year>..</>
      <url>..</>
      ...
   </>
into /dblp/articles[journal="x"]
```

Additional Measures

Apart from the μ_{mse} and $RECOMP$ metrics defined earlier, we utilize two additional supporting measures here to help explain the results:

Location Estimation Accuracy: This metric measures the effectiveness of the location estimation technique. It compares the *estimated* location against the *actual* location. The location estimation is deemed to be correct if both the estimated as well as the actual location both fall into the same histogram bucket. The location estimation accuracy is defined to be: $LEA = \frac{L_{correct}}{L_{total}}$ where $L_{correct}$ is the number of correctly estimated locations and L_{total} is the total number of locations.

μ_{count}: μ_{count} considers each histogram bucket and computes the *deviation* of the frequency of the bucket from the actual frequency normalized to the average bucket count. This metric helps in highlighting where the incorrect location estimations are being distributed.

The metric [30] is defined as:

$$\mu_{count} = \frac{\beta}{N}\sqrt{\frac{1}{\beta}\sum_{i=1}^{\beta}(f_{B_i} - B_i.count)^2}$$

where N denotes the number of values, β denotes the number of buckets, f_{B_i} denotes the actual count of bucket B_i, and $B_i.count$ denotes the current count of bucket B_i.

Results. The location estimation accuracy for the IMDB and the DBLP datasets under random insertions are shown in Figures 5.9 and 5.10, respectively, as a function of the number of value histogram buckets. Each graph shows the location estimation accuracy in two cases: (i) when the structural histogram contains only 5 buckets and, (ii) when it contains 30 buckets. Further, both the 1D and 2D versions of IMAX are presented in the graphs and we see that using 2D histograms clearly gives superior estimation accuracy as compared to using 1D histograms. Note that in order to compare *only* the location estimations, 2D histograms were used for cardinality estimation in both cases. The equivalent cardinality estimation for the 1D case would contain only the square root of the number of buckets in the value histogram. And so, the X-axis in the graph denotes the total number of 2D histogram buckets utilized per type, while the equivalent number in the 1D case would contain only the square root number of buckets. This is the tradeoff between the space utilized and the accuracy. Note however, that *increasing* the number of 1D histogram buckets has *no impact* on the *location* estimation accuracy. This is because, since *no correlation* is stored between the values and their corresponding node ids, the location estimates are always chosen *randomly* from the entire range of node ids.

The μ_{mse} metric for the type Played is shown in Figure 5.11 for both the original summary, as well as with the 1D and 2D versions of IMAX. Note that, again, there is over *two orders of magnitude* difference in accuracy between the original summary and both versions of IMAX.

An interesting observation in Figure 5.11 is that the 2D version of IMAX provides only marginal accuracy gains over the 1D version. This is inspite of the fact that the 2D version is far superior in terms of location estimation as compared to the 1D version (Figure 5.9). The reason is that the insertions are approximately *uniformly distributed* over the whole document. So, what may not be the correctly estimated location for one

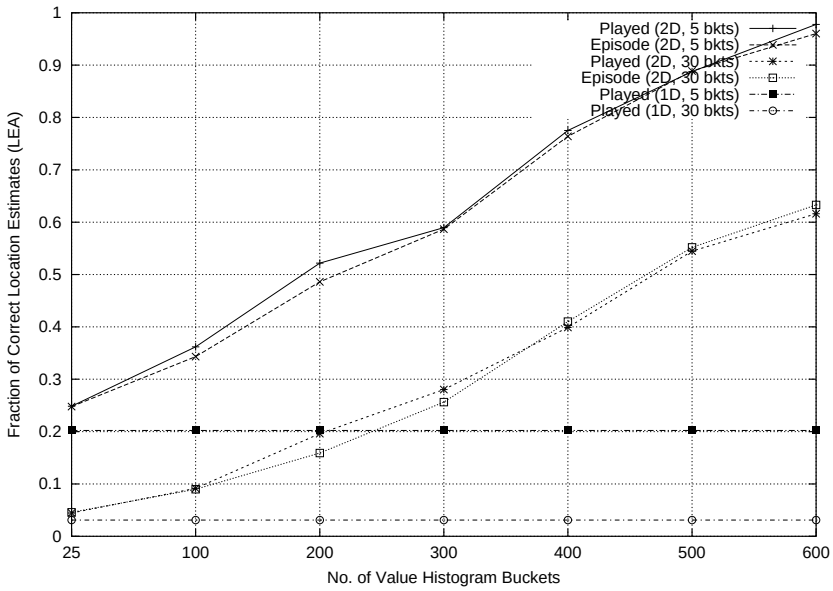


Figure 5.9: **IMDB**: *LEA* for Random Insertions with 1D and 2D Value Histograms

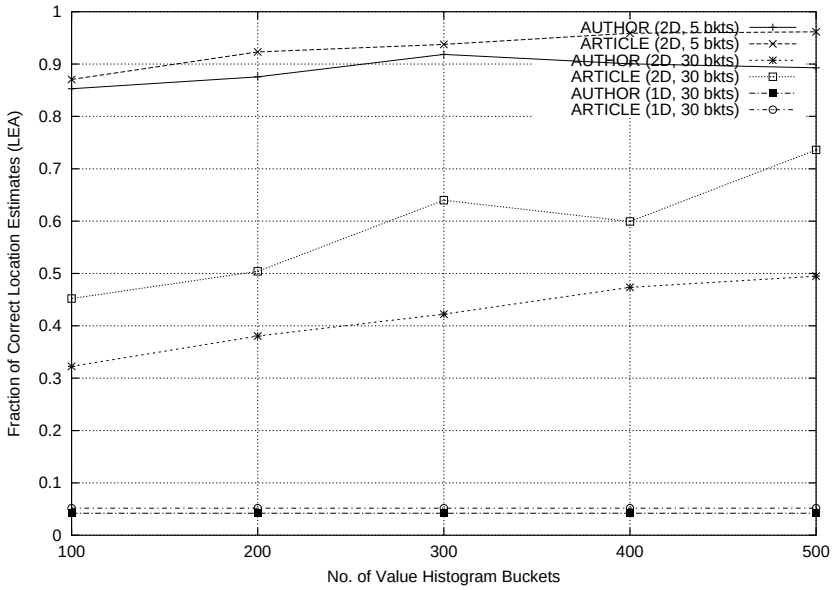


Figure 5.10: **DBLP**: *LEA* for Random Insertions with 1D and 2D Value Histograms

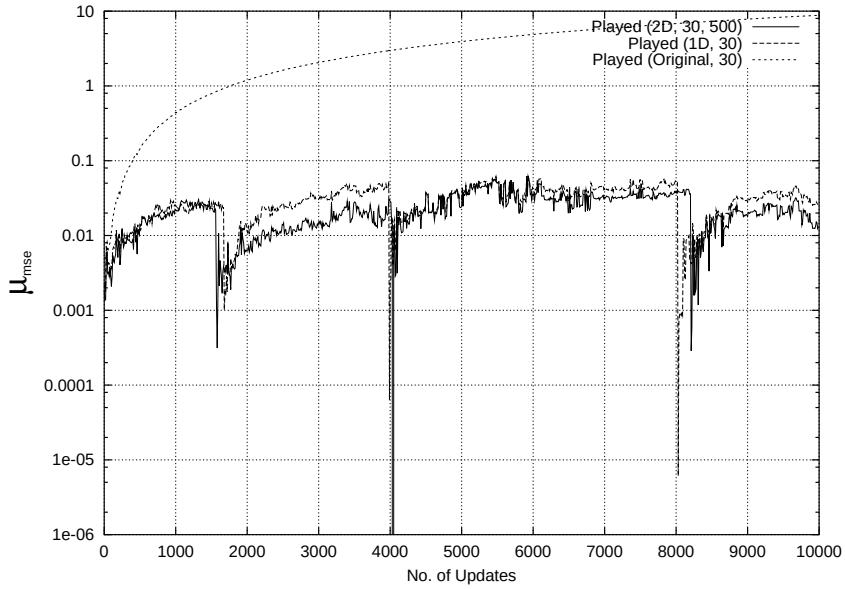


Figure 5.11: IMDB: μ_{mse} values for type Played for Random Insertions

insert may very well turn out to be the correct location for some other insert, effectively *canceling out* the effect of several wrong estimations. This is clear from Figure 5.12 which plots the μ_{count} metric for Played. The μ_{count} values of both the 1D and 2D cases are close together here.

However, if we consider insertions where the locations of the insertions are *skewed*, the benefits of using 2D histograms become immediately apparent. Such insertions are possible when, say, more recently added actors need to be updated more frequently than others. Figure 5.13 shows the μ_{count} for such skewed insertions, and we observe nearly an order of magnitude difference in the μ_{count} values of the 1D and 2D versions. This demonstrates the benefits of using 2D histograms. The μ_{mse} metric for the 2D version is shown in Figure 5.14. There is a significant improvement in the μ_{mse} values of at least an order of magnitude over the 1D version.

Similar behaviour is seen for type Episode which is nested under Played. When the insertions are random, the difference between the μ_{mse} values for the 1D and 2D cases is not significant, while in the case of skewed insertions, there is a big difference between the 1D and 2D cases. This behaviour is shown in Figures 5.15 and 5.16. Note that there is some interleaving of the lines since there are more recomputations which take place for

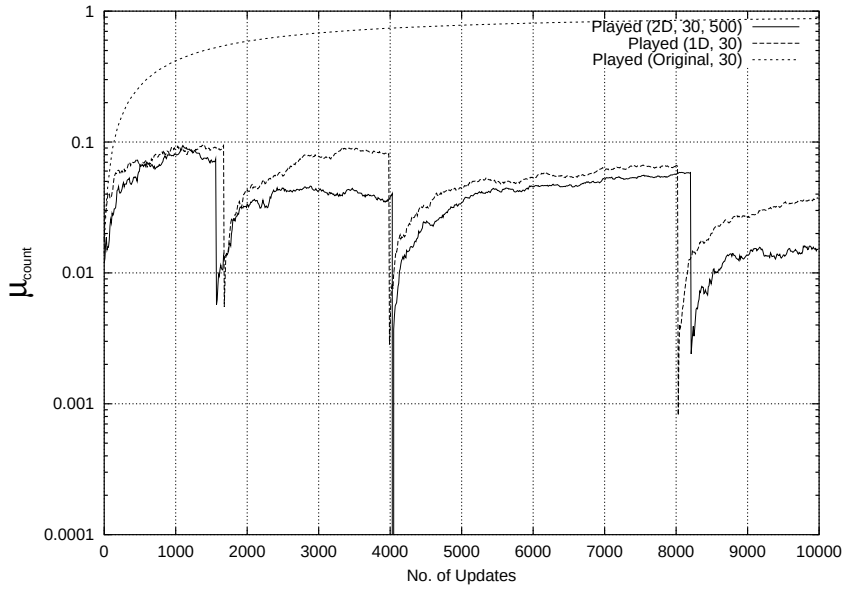


Figure 5.12: IMDB: μ_{count} values for type Played for Random Insertions

Type	**No. of Insertions (Random)**	*RECOMP* **(Random)**	**No. of Insertions (Skewed)**	*RECOMP* **(Skewed)**
Played	10000	0.03%	2000	0.05%
Episode	104569	0.006%	20937	0.02%
TOTAL	124569	0.01%	24937	0.02%

Table 5.3: IMDB: *RECOMP* with Random and Skewed Insertions

Episode, but the general trend of the difference between the μ_{mse} values for the 1D and 2D cases can be seen clearly.

Moving on to the efficiency aspect of IMAX under random insertions, the number of recomputations for both DBLP and IMDB, with and without skewed insertions, are shown in Tables 5.3 and 5.4, respectively. The tables provide the specific measures for only a subset of the types, but the totals in the last line are across all types.

Clearly, the number of recomputations required is a very small fraction of the total number of insertions made in the document. Note that the number of recomputations can be further reduced by increasing the reorganization threshold – trading off on the accuracy of the histograms.

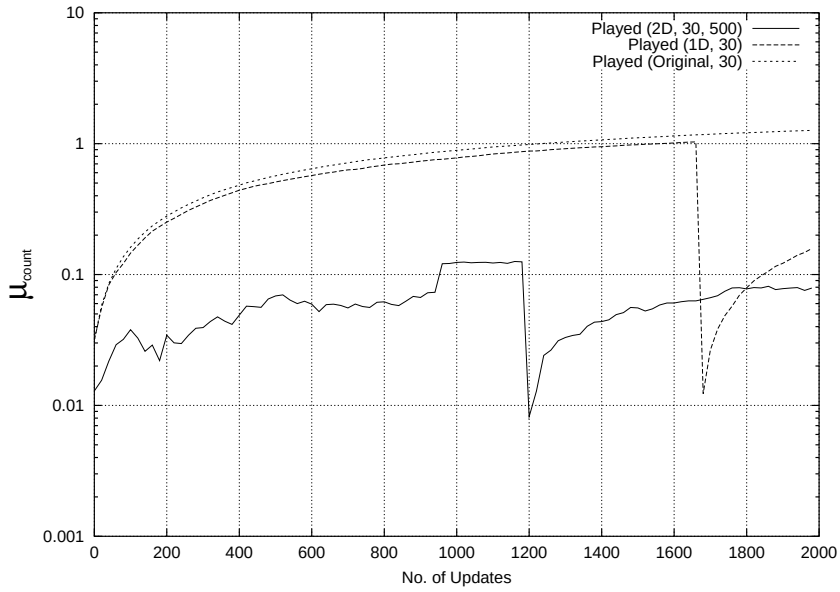


Figure 5.13: IMDB: μ_{count} values for type Played for Skewed Insertions

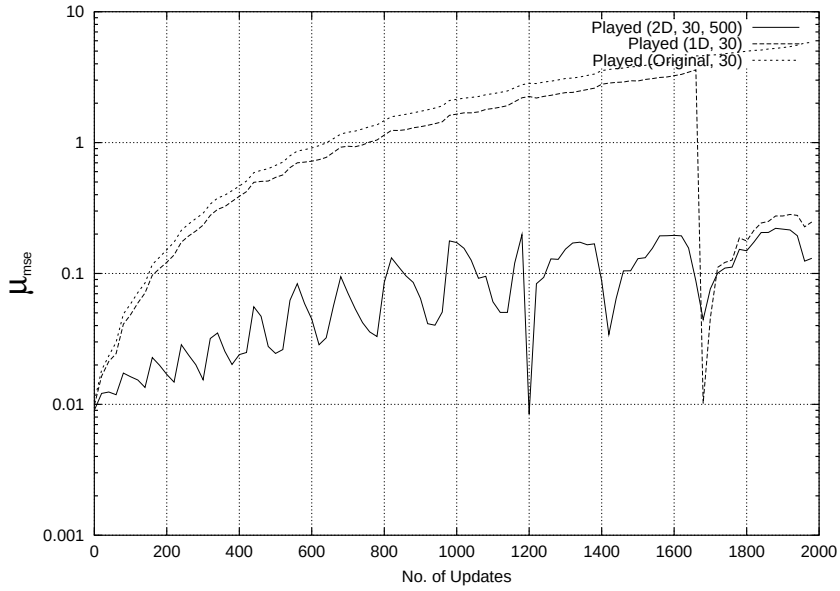


Figure 5.14: IMDB: μ_{mse} values for type Played for Skewed Insertions

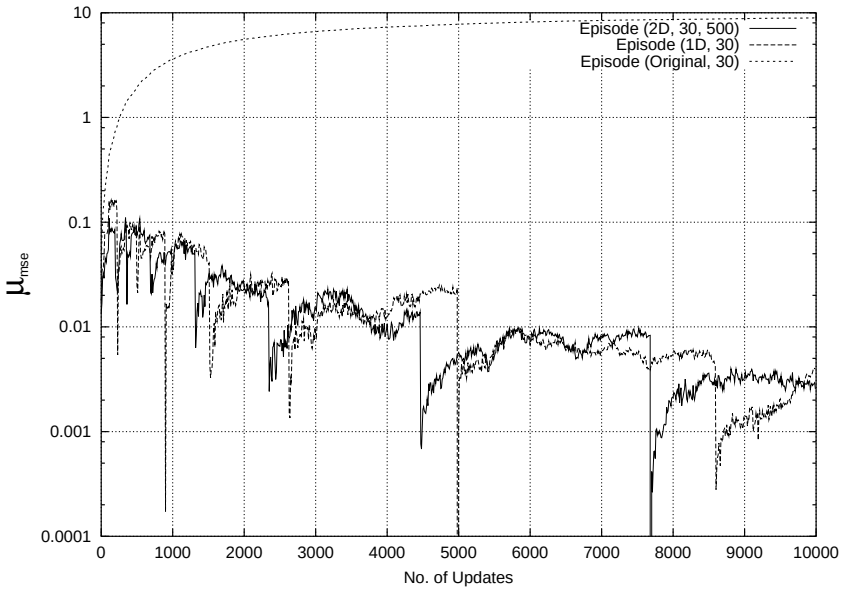


Figure 5.15: IMDB: μ_{mse} values for type Episode for Random Insertions

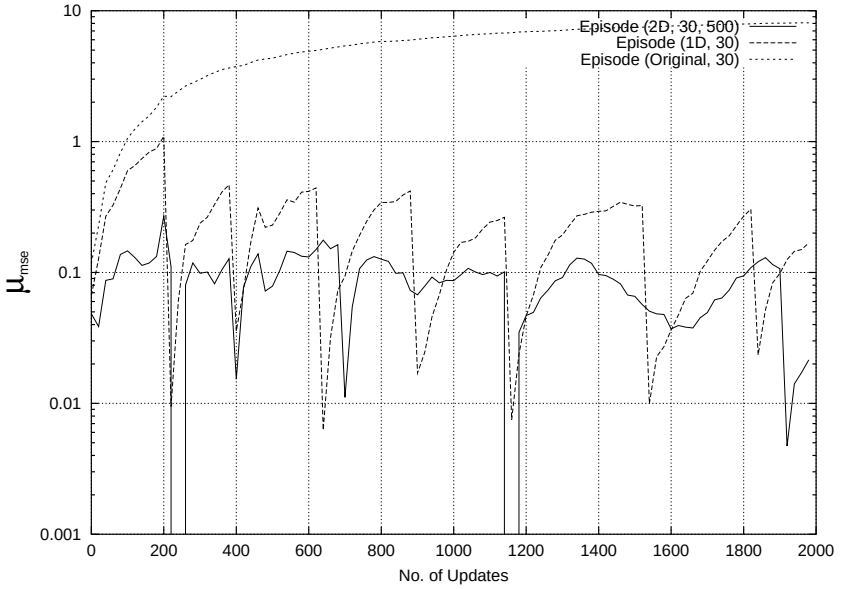


Figure 5.16: IMDB: μ_{mse} values for type Episode for Skewed Insertions

Type	**No. of Insertions (Random)**	*RECOMP* **(Random)**	**No. of Insertions (Skewed)**	*RECOMP* **(Skewed)**
ARTICLE	8000	0.02%	2000	0.05%
AUTHOR	14624	0.1%	3843	0.28%
TOTAL	88414	0.14%	22606	0.36%

Table 5.4: DBLP: RECOMP with Random and Skewed Insertions

Multiple Insertions. We consider here single update queries which spawn multiple insertions. For example, adding a comment "Arnold Rocks" for all films starring Arnold Schwarzenegger, or adding information templates for all shows satisfying certain criteria. We experimented with multiple-insertion updates on the article database of DBLP. The update involved adding a LINK for a given author denoting his/her URL. Such an update would require multiple insertions of the tag `link` depending on the number of articles authored by the author since the tag should be added to each such occurrence. The DBLP document contained a total of 1165 authors, each with at least 10 articles spread over more than 17000 articles. We performed insertions of the following form:

```
update insert
  <link> .. </> into
/dblp/article[author="x"]
```

As with the unique insertions, two sets of insertions were performed: a set of skewed insertions with around 20% of authors; and another set of insertions involving all authors. The μ_{mse} metric for both cases are shown in Figures 5.17 and 5.18, respectively. The utility of 2D histograms is limited in the case of uniformly distributed insertions, but provides considerable advantage when the insertions are skewed. This conclusion is supported by the corresponding μ_{count} metrics shown in Figures 5.19[3] and 5.20.

The efficiency of this set of insertions was quite good since no recomputations were required in the case of skewed insertions, while a single recomputation was performed

[3]While Figures 5.17 and 5.19 may look very similar, they are not the same graph.

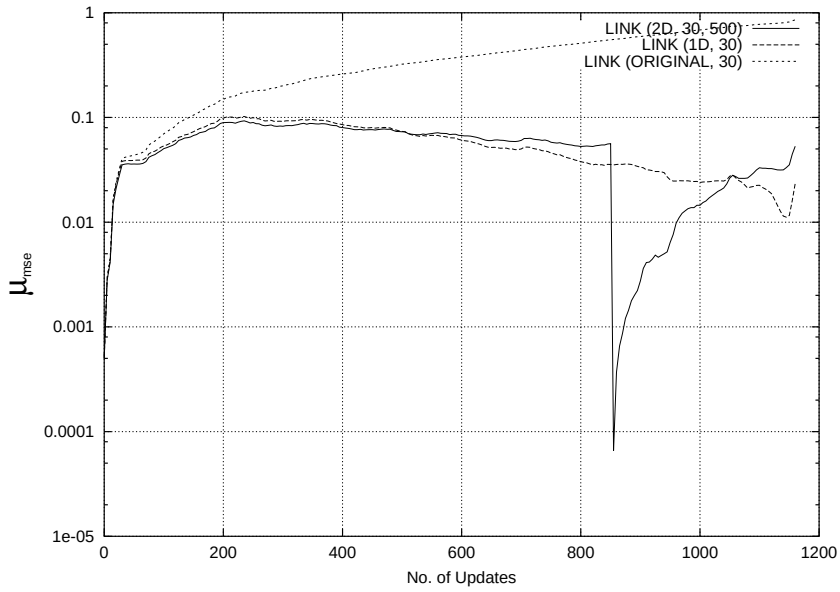


Figure 5.17: **DBLP**: μ_{mse} values for type LINK for Random Multiple Insertions

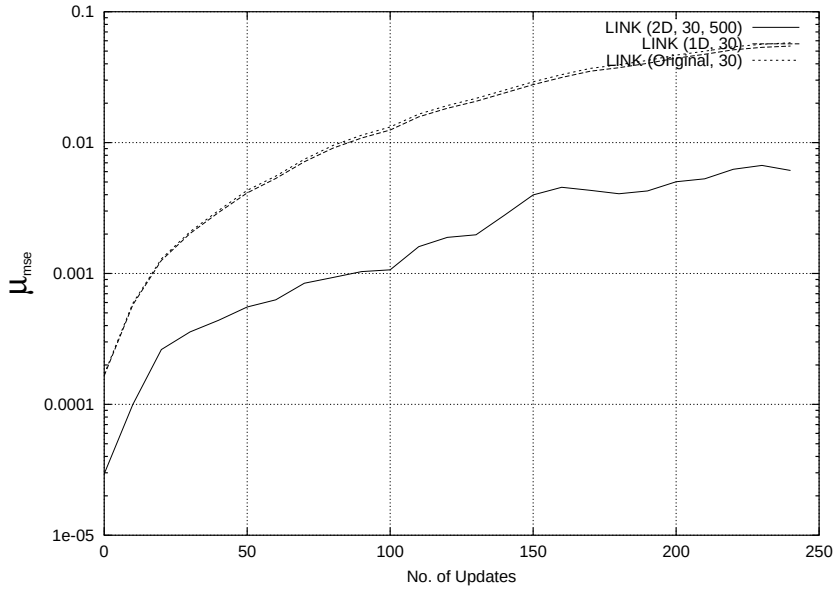


Figure 5.18: DBLP: μ_{mse} values for type LINK for Skewed Multiple Insertions

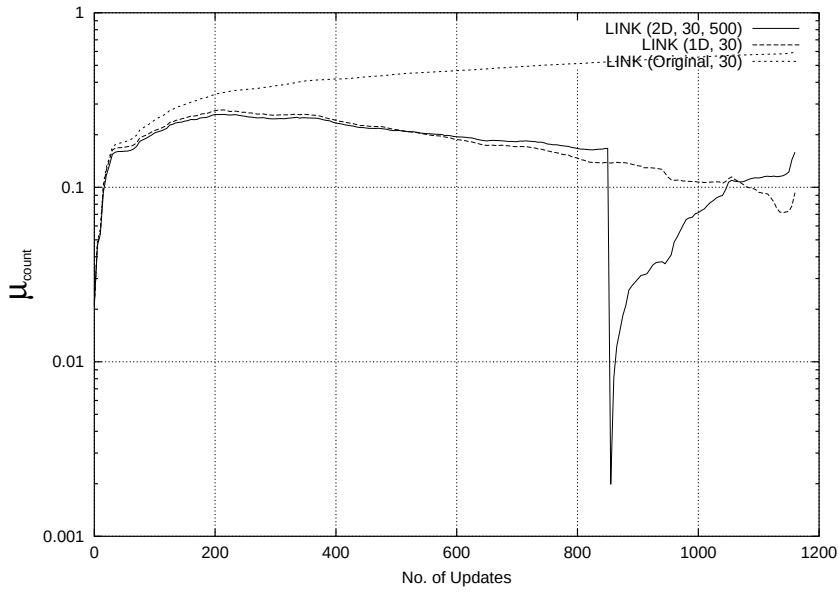


Figure 5.19: **DBLP**: μ_{count} values for type LINK for Random Multiple Insertions

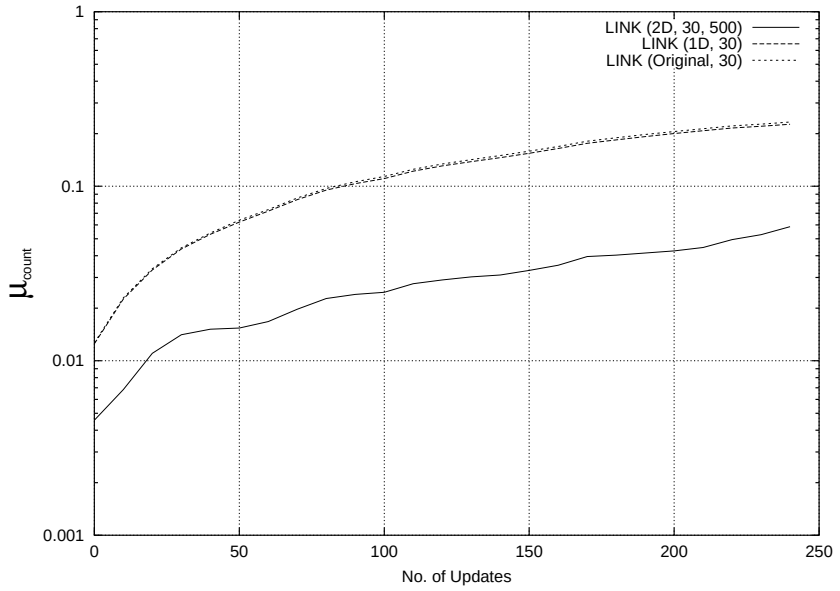


Figure 5.20: **DBLP**: μ_{count} values for type LINK for Skewed Multiple Insertions

	Append (IMDB)	Insert (IMDB)	Skewed Inserts (IMDB)	Multiple Inserts (DBLP)	Skewed Multiple Inserts (DBLP)
IMAX	97	77	122	190	86
Recompute	8167	1437	669	5403	19181

Table 5.5: Average Time per Update (in ms)

when 2D histograms were used in the case of random insertions.

5.4.4 Estimation Accuracy and Timing

The previous sub-sections dealt with the histogram accuracy (for a subset of histograms) and the number of recomputations required for various datasets. The results indicated that IMAX is very accurate when it comes to tracking the updates with a very small number of recomputations. In order to get a "global" picture of the accuracy and efficiency of IMAX, we present numbers on the estimation accuracy and timing. Note that all results are for the 30 structural histogram buckets and 500 value histogram buckets case.

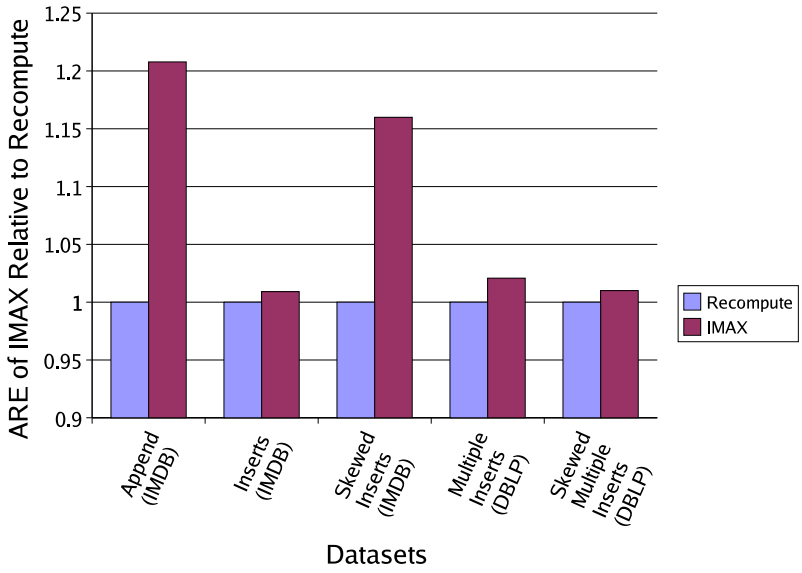


Figure 5.21: Error Relative to Recomputed Summary for IMDB and DBLP Datasets

Timing. Table 5.5 tabulates the *average time* per update for the different datasets. We see here that IMAX is almost always at least an *order of magnitude faster* than the recompute-from-scratch approach even when the occasional histogram recomputations required are taken into account. There is not too much of variation in the timings of IMAX across the datasets, since IMAX depends *primarily* on the size of the schema and the number of recomputations is only secondary in importance (hence the slight variations in timing), while the recomputation approach depends on the *size of the data*. Consequently, the larger the size of the dataset, the longer it takes for the recomputation. For example, for the DBLP Skewed Multiple Insertions case, the final size of the document was around 19MB, while for the IMDB Inserts case, the final size was around 4MB.

Estimation Accuracy. We then generated a query workload of around 300 queries with both branching path expressions without value predicates (around 15% of the workload), as well as path expressions with at least one and a maximum of two value predicates for each of the datasets. For each query workload, we computed the average relative error (ARE) in estimation using the IMAX summary as well as the recomputed-from-scratch summary. Figure 5.21 shows the ARE of IMAX *relative* to that of the recomputed-from-scratch summary. The results indicate that the quality of the IMAX summary is almost as good as that of the recomputed summary.

5.5 Conclusions

In this chapter, we introduced IMAX, an algorithm for maintaining StatiX summaries in the presence of updates to the base data. We proposed solutions for two important issues which arise in the context of XML statistics maintenance: (i) estimating the *location* of the update and (ii) estimating the ids of the update fragment. Both issues are important because of the *ordered* data model of XML. We proposed the use of 2D histograms in place of 1D histograms to substantially improve the location estimation accuracy. IMAX utilizes histogram maintenance techniques from prior literature (modified suitably for 2D histograms) in order to reduce the number of recomputations required to maintain the

quality of the summary.

Our experimental evaluation spanned three different kinds of inserts: (i) Appends, (ii) Inserts and (iii) Multiple Inserts. Using 2D histograms for location estimation resulted in an *order of magnitude* improvement in the updated histogram accuracy in the case of skewed inserts, while there was no significant difference between the 1D and 2D cases in the case of random inserts. This was due to the fact that the wrong estimates could cancel each other out since the insertions uniformly spanned all the histogram buckets.

The relative estimation accuracy of the updated summary with respect to the recomputed-from-scratch summary was within 25% in the worst case and was within 1%-2% for most of the experiments. IMAX was shown to be highly efficient when compared to the recompute-from-scratch approach – IMAX was usually at least an order of magnitude faster.

In summary, IMAX provides an effective and efficient means of statistics maintenance in the presence of updates to the data.

Chapter 6

A Cost-based XML-to-Relational Storage System

6.1 Introduction

In this chapter, we study the impact of schema transformations and the query workload on *search strategies* for finding efficient XML-to-relational mappings. Specifically, we develop a framework for generating XML-to-relational mappings, which incorporates a comprehensive set of schema transformations and is capable of supporting different mapping schemes such as ordered XML and schemaless content. Our framework, named FleXMap (Flexible XML Mappings), represents an XML Schema through type constructors (see Chapter 3) and uses this representation to define several schema transformations from the existing literature. We also propose variations of these transformations that lead to a more efficient search process, as well as new transformations that derive additional useful configurations.

FleXMap is built on top of the LegoDB prototype [8] and improves on it in several ways. Using FleXMap, a *larger* space of relational configurations, including those considered by LegoDB, is explored, leading to a final storage design that is much more efficient than those derived by LegoDB.Here, we describe a series of greedy algorithms that we have experimented with, and show how the choice of transformations impacts the search space

of configurations. The algorithms differ in the number and type of transformations they utilize. Intuitively, the size of the search space examined increases as the number/type of transformations considered in the algorithms increase. Our empirical results demonstrate that, in addition to deriving better quality configurations, algorithms that search a larger space of configurations can sometimes converge faster. Further, we propose optimizations that significantly speed up the search process with very little loss in the quality of the selected relational configuration.

An important aspect of cost-based XML-to-relational mapping is evaluating the cost of the input workload for each of the derived configurations. In order to compute precise cost estimates, it is important that accurate statistics are available as transformations are applied. Clearly, it is not practical to scan the base data for each new relational configuration that is derived during the iterative search process. As discussed later in this chapter, we address this issue by gathering statistics at the appropriate granularity before the search starts; and *deriving* accurate statistics during the search process.

Organization. The rest of the chapter is organized as follows. In Section 6.2 we show how to derive relational configurations from schema trees. In Section 6.3 the propagation of statistics when transformations are applied as well as query translation are discussed. In Section 6.4 we develop a series of greedy search algorithms and in Section 6.5 these algorithms are evaluated. In Section 6.6 we propose optimizations to speed up the search process and then conclude in Section 6.7.

6.2 From Schema Trees to Relational Configurations

6.2.1 Basic Mapping

Given a schema tree with annotated nodes, a relational configuration is derived as follows:

- If N is an annotation in the schema tree, then there is a relational table T_N corresponding to it. This table contains a *key* column and a *parent_id* column which

points to the key column of the table corresponding to the *closest named ancestor* of the current node, if it exists.

- If the subtree of the node annotated by N is a <simple type>, then T_N additionally contains a column corresponding to that type to store its values.
- If N is the annotation of a node, then T_N contains as many additional columns as the number of non-annotated children of N that are of type <simple type>.

One of the main motivations of transforming the XML schema in the XML domain rather than in the relational domain is that it is possible to exploit some of the syntactic information inherent in the XML Schema constructs. For example, it is possible to know from the XML Schema that a certain type is repeated multiple times and hence it would be more efficient to store it in a separate table. Similarly, if a type is part of a union in the schema, then we already know that there are potentially many null values in the column corresponding to that type. We make use of this knowledge in framing the following two additional rules for deriving a relational configuration from the given schema.

- Repeated types are stored in a separate table. The alternatives would be to: (i) store each occurrence of the repetition as separate columns in its parent table leading to an artificial upper bound on the number of repeats, or, (ii) store all occurrences of the repetition in the same column by duplicating the values in the rest of the tuple leading to wasted space (as well as increased complexity for updates).
- Types which are part of a union are stored in a separate table. This rule avoids nulls in the parent table.

The relational configuration corresponding to the schema tree in Figure 6.1 for the Director subtree is shown in Figure 6.2. Note that the above rules form just one possible set of rules. It is possible to frame a different set of rules – for example, we could allow components of a union to be inlined into the parent, but pay the cost of null values.

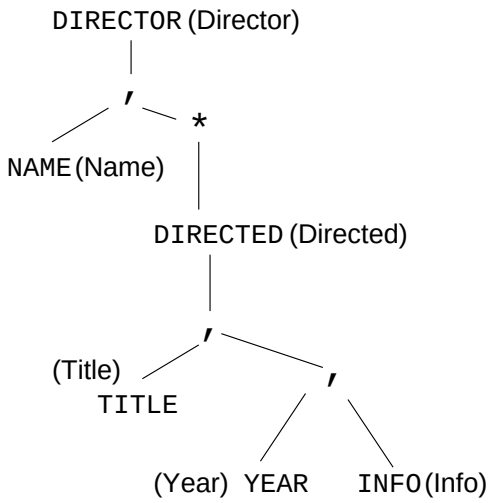


Figure 6.1: (Partial) Schema Tree for the IMDB Schema

```
Table Director [director_key INT, parent_id INT]
Table Name  [Name_key INT, NAME VARCHAR (100),
             parent_director_id INT]
Table Directed [Directed_key INT, parent_director_id]
Table Title [Title_key INT, TITLE VARCHAR (100),
             parent_directed_id INT]
Table Year [Year_key INT, YEAR, parent_directed_id INT]
Table Info [Info_key INT, INFO VARCHAR (100),
            parent_directed_id INT]
```

Figure 6.2: Relational Schema for the (partial) Schema Tree

6.2.2 Supporting Additional Features of XML Schema

The current implementation of FleXMap supports the basic features of XML Schema, but can be extended to support other features by simply adding the appropriate rules. For example:

Ordered XML: In order to support ordered XML, one or more additional columns to store the ordinal of a column could be incorporated into each of the relational tables [68]. Hence the basic mapping would include the addition of another column for the ordinal.

Mixed Content: A simple way to handle mixed content is to have one column for all the text content and additional columns (or tables) for each of the tagged contents. Another option is to treat the whole content as a CLOB and assign a single column to it.

It is also possible to support different storage schemes to a limited extent. For example, by introducing an "ANYTYPE" constructor, we can define a rule which maps annotated nodes of that type to a ternary relation (edge table) [25].

6.2.3 Schema Transformations and Relational Configurations

The previous section listed the rules for the translation of a given XML schema tree into a relational configuration. In this section, we give examples to show how several schema transformations defined in Chapter 3 (such as inline/outline, type split/merge, union distribution/factorization, repetition split/merge) help in deriving a variety of different relational configurations.

Inline and Outline

Clearly, inlining a node which is a simple type corresponds to including a separate column for it in the table of its parent (assuming that the parent is outlined). Conversely, outlining a type implies that a separate table is created for it.

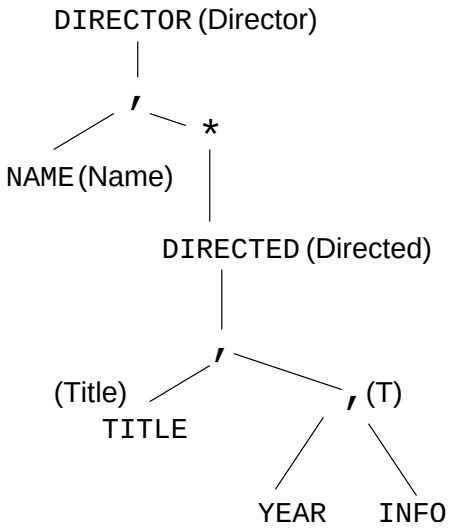


Figure 6.3: A Subset of Annotations

```
Table Director [Director_key INT, parent_id INT]
Table Name [Name_key INT, NAME VARCHAR (100),
            parent_Director_id INT]
Table Directed [Directed_key INT, parent_Director_id INT]
Table Title [Title_key INT, TITLE VARCHAR (100),
             parent_Directed_id INT]
Table T [T_key INT, YEAR INT, INFO VARCHAR (100),
         parent_Directed_id INT]
```

Figure 6.4: Relational Schema with Annotation "T"

Inline and outline may also be used to group elements together. Consider Figure 6.3 in which introducing the annotation T and removing annotations $Year$ and $Info$ results in the new relational schema shown in Figure 6.4. This configuration *groups* Year and Info together in a single table.

Since any node in the schema tree can be inlined/outlined, it is possible to generate 2^n number of different configurations, where n is the number of nodes in the schema tree, with just the inline and outline operations. However, grouping of elements may not be a useful transformation since much of its functionality can be subsumed by relational schema design tools which support vertical partitioning as one of the schema optimizations (*e.g.,* Oracle's Designer 2000).

Type Split/Merge

We refer to a type as *shared* when it has distinct annotated parents. In the example shown below, the type Title is shared by the types Show and Directed. Consequently, the table corresponding to Title would contain a parent_id column which contains key values from both Directed as well as Show – hence the parent_id column is not a foreign key. By splitting and renaming the type Title to STitle and DTitle, a relational configuration is derived where a separate table is created for each type of title.

define type Show { *element* `SHOW` {*type* Title, (*type* Tv | *type* Movie) }}
define type Director { *element* `DIRECTOR` {*type* Title, *type* Directed }}
define type Title { *element* `TITLE` {*xsd:string* }}

Type split Title →

define type Show { *element* `SHOW` {*type* STitle, (*type* Tv | *type* Movie) }}
define type Director { *element* `DIRECTOR` {*type* DTitle, *type* Directed }}
define type STitle { *element* `TITLE` {*xsd:string* }}
define type DTitle { *element* `TITLE` {*xsd:string* }}

Translated to →

```
Table Show     [Show_key INT, IMDB_parent_id INT]
Table Director [Director_key INT, IMDB_parent_id INT]
Table DTitle   [DTitle_key INT, DTitle_TITLE VARCHAR (100),
                Directed_parent_id INT]
Table STitle   [STitle_key INT, STitle_TITLE VARCHAR (100),
                Show_parent_id INT]
...
```

6.2.4 Structural Transformations

We now describe how structure changing transformations (including those defined in Chapter 3) can be utilized to derive useful relational configurations.

Commutativity and Associativity

Two basic structure-altering operations that we consider are: *commutativity* and *associativity*. Associativity is used to *group* different types into the same relational table. Consider, for example, the type Directed shown in Figure 6.5. The first tree in this figure yields a relational schema in which the Year and Info of Directed are stored in a single table called Year_Info. We can change this grouping by applying associativity as shown in the second tree and obtaining a relational schema in which Title and Year appear in a single table called Title_Year.

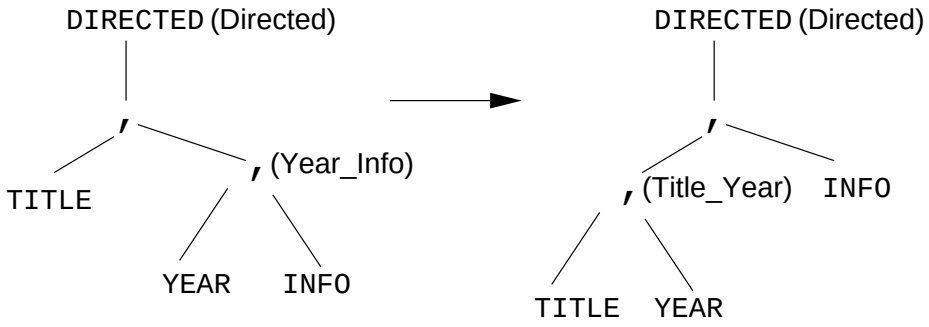


Figure 6.5: Applying Associativity

Commutativity by itself does not give rise to different relational mappings[1], but when

[1]Note that commuting the children of a node no longer retains the original order of the XML schema.

combined with associativity may generate mappings different from those considered in the existing literature. For example, in Figure 6.5, by first commuting Year and Info and then applying associativity, we get a configuration in which Title and Info are stored in the same relation.

As with inline and outline, much of the functionality of these two operations could be taken over by a relational schema optimizing tool which does vertical partitioning.

Union Distribution/Factorization

We utilize union distribution in order to separate the components of a union into different tables. The following example shows how to use a combination of union distribution, outline and type split to derive a useful relational configuration:

define type Show { *element* `SHOW` {*type* Title, (*type* TV | *type* Movie) }}

Distribute Union →

define type TVShow { *element* `SHOW` {*type* TVTitle, *type* Tv }}
define type MovieShow { *element* `SHOW` {*type* MovieTitle, *type* Movie }}

Inline →

define type TVShow {
  element `SHOW` {*element* `TITLE` { *xsd:string* }, *element* `TV` { *xsd:string* } }}
define type MovieShow {
  element `SHOW` {*element* `TITLE` { *xsd:string* }, *element* `MOVIE` { *xsd:string* }}

Translated to →

```
Table TVShow [TVShow_key INT, TVShow_TITLE VARCHAR (100),
              TVShow_TV VARCHAR (100), IMDB_parent_id INT]
```

```
Table MovieShow [MovieShow_key INT, MovieShow_TITLE VARCHAR (100),
                 MovieShow_MOVIE VARCHAR (100), IMDB_parent_id INT]
```

The information about TV shows and movie shows, which previously may have been stored in a single table called `Show`, is split into two separate tables – this is equivalent to horizontally partitioning the `Show` table, *i.e.,* one partition is created for TV shows and another for movies.

Given a single table (that is, just the Show table with, say, two columns for Tv and Movie), it would not be possible for any relational tool to do the horizontal partitioning. But, by performing this transformation in the XML domain, we are able to generate this configuration.

Repetition Split/Merge

According to the rules in Section 6.2, a repeated type is always stored in a separate table. However, it is possible to inline some of these values by a transformation which *splits* the repetition. For example:

define type Show {*element* SHOW {*type* Title, *type* Aka*} }

Split Repetition →

define type Show {*element* SHOW {*type* Title, *type* Aka1?, *type* Aka2*} }

Inline →

define type Show {*element* SHOW
{*element* TITLE { *xsd:string* }, *element* AKA { *xsd:string* }?, *type* Aka2*} }

Translated To →

```
Table Show [Show_key INT, Show_TITLE VARCHAR (100),
            Show_AKA1 VARCHAR (100), IMDB_parent_id INT]
Table Aka2 [Aka_key INT, AKA VARCHAR (100), Show_parent_id INT]
```

By *splitting* the repetition Aka*, the new type Aka1 may be inlined into Show. Aka2* may be split over and over again. In order to prevent an infinite number of splits, the number of splits must be fixed during the search. Similar to union distribution and factorization, repetition split and repetition merge also result in relational configurations which are derivable only by looking at the XML schema.

Many other transforms such as *simplifying unions* [65] (a lossy transform[2] which enables the inlining of one or more of the components of the union), etc. can be defined similarly.

6.3 Evaluating Configurations

It is important that during the search process, precise cost estimates are computed for the query workload under each of the derived configurations – this, in turn, requires accurate statistics. Since it is not practical to scan the base data afresh for each new relational configuration derived, it is crucial that these statistics be accurately *propagated* as transformations are applied.

6.3.1 Collection and Propagation of Statistics

For ease of exposition, we describe the collection and propagation of statistics at the XML Schema level, and later show how to translate these into relational statistics.

An important observation about the transformations defined in Chapter 3 is that merge operations (type merge, union factorization, repetition merge, inline) preserve the accuracy of the statistics, while split operations (type split, union distribution, repetition split, outline) do not. Intuitively, if two types T_1 and T_2 are merged into T, precise

[2]That is, the set of documents validated after the transform is applied is *different* from the set validated before.

statistics for T can be derived by summing/unioning the statistics of T_1 and T_2. However, when a type T is split into T_1 and T_2, in general it is not possible to determine precisely the statistics for the new types. However, in some special cases, *e.g.,* for the outline transform, it may be possible to accurately *infer* the statistics of the new type from the statistics of the parent type – this is true if the currently outlined type occurs exactly once under its parent.

Consequently, in order to preserve the accuracy of the statistics, before the search procedure starts, *all* possible split operations are applied to the user XML schema. Statistics are then collected for this *fully decomposed* schema. Subsequently, during the search process, only merge operations are considered.

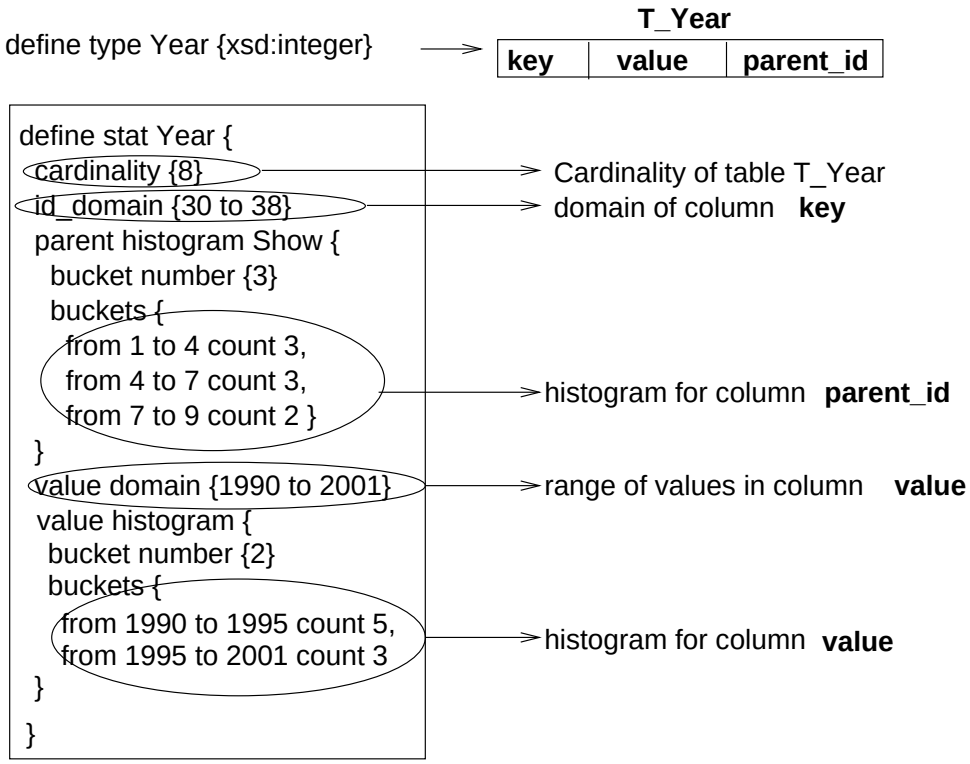


Figure 6.6: Statistics Translation

In our prototype implementation, we use StatiX (Chapter 4) to collect statistics. These statistics are then translated into relational statistics. All components of the StatiX summary, such as the cardinality, parent histogram, value histogram, etc. have a counterpart in the relational domain. Hence the statistics for each type in the StatiX summary can be

translated to the table or column statistics for that type in the relational configuration. The translation procedure is illustrated through the example in Figure 6.6.

The derived relational statistics are used as input to a relational optimizer (FleXMap uses the optimizer described in [61]), which in turn computes cost estimates for the (translated) query workload (described in Section 6.3.2) under the current relational configuration.

6.3.2 Query Translation

Input: Q, $\mathcal{S}$, $\mathcal{R}$
Q is the XQuery query, $\mathcal{S}$ is the schema, $\mathcal{R}$ is the relational configuration
1: $Q' =$ *normalize* the XQuery Q
2: $B =$ set of *bind* variables in Q'
3: $T =$ set of consistent types for the variables in B using $\mathcal{S}$
4: $SQL_{select} =$ create **SELECT** clause using $T.return$ and $\mathcal{R}$
5: $SQL_{Pathjoin} =$ create joins required for the path traversals using $T.fors$ and $\mathcal{R}$
6: $SQL_{where} =$ create **WHERE** clause using $SQL_{Pathjoin}$, $T.wheres$ and $\mathcal{R}$
7: $SQL_{from} =$ create **FROM** clause using SQL_{select}, SQL_{wheres} and $\mathcal{R}$

Algorithm 6: Query Translation

FleXMap currently supports XQuery queries which contain path expressions, value selections and value joins. These queries translate to unnested "SELECT-FROM-WHERE" SQL queries. A high-level algorithm of the query translation process is shown in Algorithm 6. We describe the working of the algorithm through a detailed example. Consider the XML Schema in Figure 6.7 and the corresponding relational configuration. The mapping between the type names in the XML Schema and the table/column names in the relational configuration are also shown. Suppose we wish to query the names of directors of all shows produced in the year 1996. Such a query would be expressed as follows:

```
for     $i  in /IMDB/SHOW
        $j  in /IMDB/DIRECTOR
where    $i/YEAR = '1996'
  and $i/TITLE = $j/TITLE
return   $j/NAME
```

Step 1 of Algorithm 6 *normalizes* the query. This is simply a process of converting the given XQuery into a more simplified data structure for easier translation. The normalized XQuery is divided into four parts: (i) the *root*, (ii) a set of $fors$ which assign bind variables to each step in the traversals, (iii) a set of *wheres* which specify the conditions in the "where" clause of the XQuery and (iv) a set of *returns* which specifies the return values. For the above example query, the normalization process yields the following:

```
root =   IMDB
fors =   [v0, IMDB, SHOW]
         [v1, IMDB, DIRECTOR]
         [v2, v0, TITLE]
         [v3, v1, TITLE]
         [v4, v1, NAME]
         [v5, v0, YEAR]
wheres = [XSelect (v5, '1996', '=')]
         [XJoin (v2,v3)]
return = [v4]
```

The *bind* variables in the normalized query are: $\{v0,v1,v2,v3,v4,v5\}$ (Step 2 in Algorithm 6). Each of these bind variables can be assigned typenames by looking at the schema (Step 3). For the schema in Figure 6.7, there is a *single* set of typenames which are consistent, and that is: $\{v0 = \mathsf{E\_IMDB}, v1 = \mathsf{Director}, v2 = \mathsf{ShowTitle}, v3 = \mathsf{DirectorTitle}, v4 = \mathsf{DirectorName}, v5 = \mathsf{ShowYear}\}$. Note that it is possible to have more than one set of consistent type assignments. For example, if Show was distributed into Show1 and Show2,

Types in the Schema

E_IMDB → **Outlined**
Show → **Outlined**
Director → **Outlined**
Tv → **Outlined**
Movie → **Outlined**
DirectorName → **Inlined into** Director
DrectorTitle → **Inlined into** Director
ShowTitle → **Inlined into** Show
ShowYear → **Inlined into** Show

define type E_IMDB { *element* `IMDB` { *type* Show*, *type* Director* }
define type Show { *element* `SHOW` {
element `TITLE` { *xsd:string* }, *element* `YEAR` { *xsd:integer* },
(*type* Tv | *type* Movie) }}
define type Director { *element* `DIRECTOR`
{*element* `NAME` { *xsd:string* }, *element* `TITLE` { *xsd:string* } }}
define type Tv { *element* `TV` {*xsd:string* } }
define type Movie { *element* `MOVIE` {*xsd:string* } }

The Relational configuration

```
Table E_IMDB   [IMDB_Key INT, doc_name VARCHAR (100)]
Table Show     [Show_Key INT, SHOW_TITLE VARCHAR (100),
                SHOW_YEAR INT, IMDB_parent_id INT]
Table Director [Director_Key INT, DIRECTOR_NAME VARCHAR (100),
                DIRECTOR_TITLE VARCHAR (100),
                IMDB_parent_id INT]
Table Tv       [Tv_Key INT, Tv_TV VARCHAR (100), Show_parent_id INT]
Table Movie    [Movie_Key INT, Movie_MOVIE VARCHAR (100),
                Show_parent_id INT]
```

Mapping between type name and table/column

E_IMDB → `E_IMDB, IMDB_Key`
Show → `Show, Show_Key`
Director → `Director, Director_Key`
Tv → `Tv, Tv_Key`
Movie → `Movie, Movie_Key`
DirectorName → `Director, DIRECTOR_NAME`
DirectorTitle → `Director, DIRECTOR_TITLE`
ShowTitle → `Show, SHOW_TITLE`
ShowYear → `Show, SHOW_YEAR`

Figure 6.7: The IMDB Schema and its relational configuration

then there would be *two* sets of consistent type assignments, one with Show1 and one with Show2.

Once the type assignments are made, we now construct the SELECT, FROM and WHERE clauses of the SQL query using the mapping between the type names and the table/column names. We first construct the SELECT clause (Step 4) by looking at the *return*s in the normalized XQuery. In this case, there is a single *return*, $v4$ which corresponds to DirectorName whose table/column is `Director, DIRECTOR_NAME`. Hence SQL.SELECT = Director.DIRECTOR_NAME.

Next, we construct the WHERE clause of the SQL query (Steps 5 and 6). There are two kinds of conditions to take care of, the joins or selection predicates in the *wheres* of the normalized XQuery and the *path joins* corresponding to the traversals in the XQuery. In order to construct the path joins, we simply have to consider each of the location steps and if the two types corresponding to the location step are in different tables, then we connect the two with a join of the *parent_id* and *key* columns. For our example, there are only two path joins: one between Show and E_IMDB and another between Director and E_IMDB. All other types in the query are inlined into their parents and hence do not need to have a path join. So the SQL.pathjoins = {`Director.IMDB_parent_id = E_IMDB.IMDB_Key, Show.IMDB_parent_id = E_IMDB.IMDB_Key`}. Then we construct the joins corresponding to the *wheres* in the normalized XQuery. The $XSelect$ predicate is translated into {`Show.SHOW_YEAR = 1996`} and the $XJoin$ predicate is translated into {`Show.SHOW_TITLE = Director.DIRECTOR_TITLE`}. Combining both the path joins and the regular joins, we get SQL.WHERE = {`Show.SHOW_TITLE = Director.DIRECTOR_TITLE, Show.SHOW_YEAR = 1996, Director.Director_parent_id = E_IMDB.IMDB_Key, Show.Show_parent_id = E_IMDB.IMDB_Key` }

Combining all the tables in the SELECT and the WHERE clauses, we construct the FROM clause (Step 7). SQL.FROM = {`E_IMDB, Show, Director`}. The XQuery query now corresponds to the following SQL query:

```
SELECT    Director.DIRECTOR_NAME
FROM      Director, Show, E_IMDB
WHERE     Show.SHOW_TITLE = Director.DIRECTOR_TITLE
     AND  Show.SHOW_YEAR = '1996'
     AND  Show.Show_parent_id = E_IMDB.IMDB_Key
     AND  Director.Director_parent_id = E_IMDB.IMDB_Key
```

6.4 Search Algorithms

Clearly, the search space of relational configurations is huge. Considering just inline and outline of elements as the allowed transformations, the number of possible relational configurations is exponential in the number of elements in the schema. Utilizing a greedy algorithm cuts this space down from $O(2^n)$ to $O(n^2)$. However, adding the other transformations such as union distribution bloats up the search space further. But, by defining more powerful transformations which *subsume* other transformations, we can ensure that even this larger search space can be searched efficiently. This is discussed in Section 6.4.3.

We next describe three greedy algorithms that we have implemented using our framework. They differ in the choice of transformations that are selected and applied at each iteration of the search.

First, consider Algorithm 7, which describes a *simple* greedy algorithm — similar to the algorithm described in [8]. It takes as input a query workload and the initial schema (with statistics). At each iteration, the transform that results in the best quality relational configuration (that is, the configuration with minimum cost) is chosen and applied to the schema (lines 5 through 19). The conversion from the transformed schema to the relational configuration (line 11) follows the rules set out in Section 6.2. The algorithm terminates when no transform can be found which improves the quality of the configuration.

Though this algorithm is simple, it is also very flexible. This flexibility is achieved by varying the strategies to select applicable transformations at each iteration (function $\mathsf{a}pplicableTransforms$ in line 8). In the experiments described in [8], only inline and outline were considered as the applicable transformations and the utility of the other

```
1: Input: queryWkld, S {Query workload and Initial Schema}
2: prevMinCost ← INFINITY
3: rel_schema ← convertToRelConfig(S, queryWkld)
4: minCost ← COST(rel_schema)
5: while minCost < prevMinCost do
6:    S' ← S {Make a copy of the schema}
7:    prevMinCost ← minCost
8:    transforms ← applicableTransforms(S')
9:    for all T in transforms do
10:      S'' ← Apply T to S' {S' is preserved without change}
11:      rel_schema ← convertToRelConfig(S'', queryWkld)
12:      Cost ← COST(rel_schema)
13:      if Cost < minCost then
14:         minCost ← Cost
15:         minTransform ← T
16:      end if
17:   end for
18:   S ← Apply minTransform to S {The min. cost transform is applied}
19: end while
```

Algorithm 7: Greedy Algorithm

transformations (*e.g.,* union distribution and repetition split) were shown independently. Below, we describe variations to the basic greedy algorithm that allow for a richer set of transformations.

As discussed in Section 6.3, it is important to perform all splits and then the merges on the schema to preserve the accuracy of statistics. It is worth pointing out that fixing this order is also important to avoid re-generating the same relational configuration in different iterations of the search. In the rest of the chapter, we assume that the starting schema for all search algorithms is the *fully decomposed schema* and only merge operations are applied during the greedy iterations.

6.4.1 InlineGreedy

The first variation we consider is *InlineGreedy*, which only allows inline transformations. Note that InlineGreedy differs from the algorithm experimentally evaluated in [8], which we term *InlineUser*, in the choice of starting schema: InlineGreedy starts with the fully decomposed schema whereas InlineUser starts with the original user schema.

6.4.2 ShallowGreedy: Adding Transforms

The *ShallowGreedy* algorithm defines the function a*pplicableTransforms* in Algorithm 7, to return *all* the applicable merge transforms. Because it follows the transformation dependencies that result from the notion of syntactic equality (see Definition 3.1), it only performs single-level or *shallow* merges.

6.4.3 DeepGreedy: *Deep* merges

The notion of syntactic equality, however, can be too restrictive for effective exploration of the search space. For example consider the following (partial) IMDB schema:

define type Show {*type* Show1 | *type* Show2}
define type Show1 { *element* SHOW { *type* Title1, *type* Year1, *type* TV }}
define type Show2 { *element* SHOW { *type* Title2, *type* Year2, *type* Movie }}

Unless a type merge of Title1 and Title2 and a type merge of Year1 and Year2 take place, we cannot factorize the union of Show1 | Show2. However, in a run of ShallowGreedy, these two type merges by themselves may not reduce the cost, but taken in conjunction with the union merge would make a substantial impact. If that is the case, ShallowGreedy is handicapped by the fact that a union merge will never be applied since the two type merges will not be chosen by the algorithm. In order to overcome this problem, we design a new algorithm called *DeepGreedy*, which we describe below.

Before we proceed to describe the DeepGreedy algorithm, we first introduce the notions of *Valid Transforms* and *Logical Equivalence*. The set of *valid transformations* for a given schema tree S is a subset of all the applicable transformations in S.

Definition 6.1 *Logical Equivalence: Two types T_1 and T_2 are* logically equivalent *under a set V of valid transforms, denoted by $T_1 \sim_V T_2$, if they can be made syntactically equal after applying a sequence of valid transforms from V.*

The following example illustrates this concept. Let $V = \{Inline\}$;

$t_1 := E(TITLE, S(string, -), Title_1)$, and $t_2 := E(TITLE, S(string, -), Title_2)$. Note that t_1 and t_2 are not syntactically equal since their annotations do not match. However, they are *logically equivalent*: by *inlining* them (*i.e.,* removing the annotations $Title_1$ and $Title_2$), they can be made syntactically equal. Thus, we say that t_1 and t_2 are logically equivalent under the set $\{Inline\}$.

Now, consider two types T_i and T_j where $T_i := E(l, t_1, a_1)$ and $T_j := E(l, t_2, a_2)$ with t_1 and t_2 as defined above. Under syntactic equality, T_i and T_j would not be identified as candidates for type merge. However, if we relax the criteria to logical equivalence with (say) $V = \{TypeMerge\}$, then it is possible to identify the *potential* type merge of T_i and T_j. Thus, several transforms which may never be considered by ShallowGreedy can be identified as candidates, provided the necessary operations can be fired to *enable* the transform. That is, if T_i and T_j are identified as a potential type merge, then to perform this type merge, t_1 and t_2 are *recursively* type merged in order to enable the type merge of T_i and T_j. Extending the above concept, we can enlarge the set of valid transforms V to contain all the merge transforms which can be fired recursively to *enable* other transforms.

DeepGreedy allows the same transforms as ShallowGreedy, *except* that potential transforms are identified not by syntactic equality, but by logical equivalence, with the set of valid transforms containing all the merge operations (including inline). This allows DeepGreedy to perform *deep* merges. Note that additional variations of the search algorithms are possible, *e.g.,* by restricting the set of valid transforms.

6.5 Performance Evaluation

In this section we present a performance evaluation of the three algorithms proposed in this chapter: InlineGreedy, ShallowGreedy and DeepGreedy. The purpose of this evaluation is twofold: (i) to analyze the relative performance of the algorithms on different kinds of query workloads, and, (ii) to establish the competitiveness of the proposed algorithms. We performed experiments on both the synthetically generated IMDB dataset as well as a subset of the DBLP dataset available from [18]. We used a Pentium IV, 2.4GHz machine with 1GB of main memory, running Redhat 8.0, for all experiments.

6.5.1 Query Workloads

We evaluated each of the algorithms on several query workloads based on: (1) the efficiency of the derived relational configuration, and (2) the efficiency of the search algorithm. Note that the latter is the same as the number of distinct configurations seen by the algorithm, and also the number of distinct optimizer invocations since each iteration involves constructing a new configuration and evaluating its quality using the optimizer.

From the discussion of the proposed algorithms, note that the behavior of each algorithm on a given query depends upon whether the query benefits more from merge transformations or from split transformations. If the query benefits more from split, then neither DeepGreedy nor ShallowGreedy is expected to perform much better than InlineGreedy.

As such, we considered the following two kinds of queries: **S-Queries** which are expected to derive benefit from split transformations (Type Split, Union Distribution and Repetition Split), and **M-Queries** which are expected to derive benefit from merge operations (Type Merge, Union Factorization and Repetition Merge).

S-Queries typically involve simple lookup. For example:

```
SQ1:  for $i in /IMDB/SHOW
      where $i/TV/CHANNEL = 9
      return $i/TITLE

SQ2:  for $i in /IMDB/DIRECTOR
      where $i/DIRECTED/YEAR = 1994
      return $i/NAME
```

Query SQ1 is specific about the Title that it wants. Hence it would benefit from a type split of Title. Moreover, it also specifies that TV Titles only are to be returned, not merely Show Titles. Hence a union distribution would be useful to isolate only TV Titles. Similarly, query SQ2 would benefit from isolating Director Names from Actor Names, and Directed Year from all other Years. Such splits would help make the corresponding tables smaller and hence lookup queries such as the above faster. The performance of the proposed algorithms on S-query workloads is analysed in Section 6.5.2.

On the other hand, M-queries typically query for subtrees in the schema which are *high up* in the schema tree. When a split operation is performed on a type in the schema, it propagates downwards towards the descendants. For example, a union distribution of Show results in a type split of Review, which in turn leads to the type split of Review's children. Hence queries which ask subtrees near the top of the schema tree would benefit from merge transforms. Similarly predicates which are "high up" in the tree would also benefit from merges. For example:

```
MQ1: for $i in /IMDB/SHOW, $j in $i/REVIEW
     return $i/TITLE, $i/YEAR, $i/AKA,
            $j/GRADE, $j/SOURCE,
            $j/COMMENTS

MQ2: for $i in /IMDB/ACTOR, $j in /IMDB/SHOW
     where $i/PLAYED/TITLE = $j/TITLE
     return $j/TITLE, $j/YEAR, $j/AKA,
          $j/REVIEW/SOURCE, $j/REVIEW/GRADE,
          $j/REVIEW/COMMENTS, $i/NAME
```

Query MQ1 asks for full details of a Show without distinguishing between TV Shows and Movie Shows. Since all attributes of Show which are *common* for TV as well as Movie Shows are requested, this query is likely to benefit from a union factorization and repetition merge. For example, a union factorization enables types like Title and Year to be inlined into the same table (the table corresponding to Show). Thus the query may benefit from reduced fragmentation. Similarly, query MQ2 would benefit from a union factorization of Show as well as a repetition merge of Played (this is because the query does not distinguish between the Titles of the first Played and the remaining Played). In both the above queries, return values as well as predicates benefit from merge transformations.

Based on the two classes of queries described above, we constructed the following workloads. Note that each workload consists of a set of queries as well as the associated weights. Unless stated otherwise, all queries in a workload are assigned equal weights and the weights sum up to 1.

1. **IMDB-S**: contains 5 S-queries on the IMDB dataset.
2. **DBLP-S**: contains 5 S-queries on the DBLP dataset.
3. **IMDB-M**: contains 8 M-queries on the IMDB dataset.
4. **DBLP-M**: contains 5 M-queries on the DBLP dataset.

The performance of the proposed algorithms on S-query workloads and M-query workloads is studied in Sections 6.5.2 and 6.5.3, respectively.

There are many queries which cannot be conclusively classified as either an S-query or an M-query. For example, an interesting variation of S-Queries is when the query contains return values that do not benefit from splits, but has predicates which do. Similarly, for M-Queries, adding highly selective predicates may reduce the utility of merge transforms. For example, adding the highly selective predicate $YEAR > 1990$ (Year ranges from 1900 to 2000) to query MQ1 would significantly reduce the number of tuples.

Such queries thus benefit from split transformations as well as merge transformations. However, in case the two types of transformations conflict, we need to analyze the balance between the two. But considering arbitrary queries is unlikely to give much insight because the impact of split transformations versus merge transformations would be different for different queries. Thus, we chose to work instead on a workload containing a *mix* of S- and M-queries, where the impact of split transformations versus the merge transformations is controlled using a parameter. The performance of the proposed algorithms on such workloads, as a function of the control parameter is studied in Section 6.5.4. Finally, in Section 6.5.5 we demonstrate the competitiveness of the configurations derived using the proposed algorithms against a set of baselines.

6.5.2 Performance on S-Query Workloads

We present results for the 2 workloads – IMDB-S and DBLP-S. The cost differences, shown in Figure 6.8, of the derived configurations for the three algorithms, DeepGreedy, ShallowGreedy and InlineGreedy, were within 1% of each other for the IMDB-S and

DBLP-S workloads. Note that *all relative costs* shown in the graphs are with respect to DeepGreedy, assuming that DeepGreedy has a cost of 1. This behaviour was expected since not too many merge transforms were required to come up with an efficient relational configuration. However, note that in both cases, DeepGreedy was still marginally better than either ShallowGreedy or InlineGreedy.

Figure 6.8: Cost of Workloads containing S-Queries

The relative number of configurations examined by each of the three algorithms DeepGreedy, ShallowGreedy and InlineGreedy, are shown in Figure 6.9 (again, DeepGreedy is taken as the baseline). In terms of number of schemas examined, DeepGreedy examined the largest number of configurations and InlineGreedy the least for the IMDB-S workload, while DeepGreedy was the most efficient in the case of the DBLP-S workload. This difference in behaviour can be explained by the nature of the two schemas. IMDB is much more deeply nested than DBLP resulting in a lot more options for merge transforms as compared to inlines. DBLP is a relatively "flat" schema with only one or two levels of nesting and hence has a relatively small number of merge transforms to consider. In the case of the DBLP-S split workload, DeepGreedy performed only a few merge transforms (specifically, repetition merge) which considerably reduced the number of subsequent inlines as well, while InlineGreedy had a lot more inlines to consider to improve the quality. Hence

Figure 6.9: No. of configurations Examined for Workloads Containing S-Queries

DeepGreedy was more efficient than InlineGreedy in the case of the DBLP schema. However, an interesting observation is with respect to ShallowGreedy on the DBLP schema – it turned to be the most expensive algorithm. This was because of its *inability* to perform the same merge transforms as DeepGreedy. Hence, it not only had to inspect all the inlines (which were reduced in the case of DeepGreedy), but also continue to consider several "wasteful" merge transforms which would not benefit the query workload.

6.5.3 Performance on M-Query Workloads

Figure 6.10 shows the relative costs of the 3 algorithms for the 2 workloads, IMDB-M and DBLP-M. In the case of IMDB-M, DeepGreedy performs extremely well compared to ShallowGreedy and InlineGreedy since DeepGreedy is capable of performing deep merges which benefit the merge queries. But, the difference in the quality of configurations was not very significant in the case of the DBLP-M workload, which can again be explained due to the flatness of the DBLP schema and the lack of opportunities to perform the merge transforms. However, note that InlineGreedy outputs configurations which are of significantly lower quality than those output by DeepGreedy, indicating that whatever merge transforms were considered by DeepGreedy were highly beneficial to the workload.

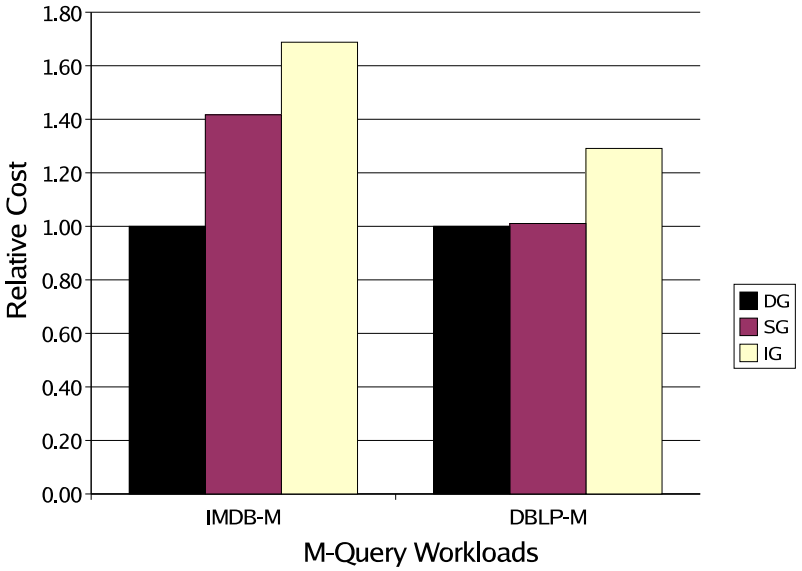


Figure 6.10: Cost of Workloads Containing M-Queries

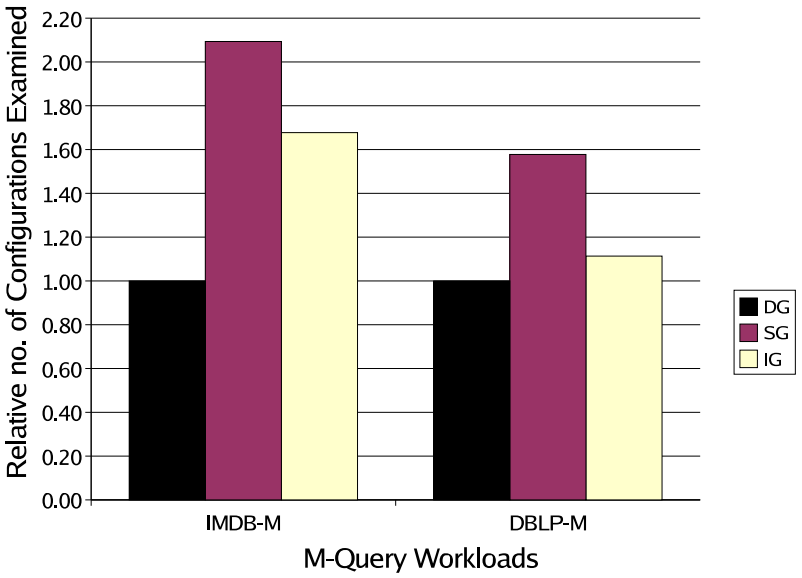


Figure 6.11: No. of configurations Examined for Workloads Containing M-Queries

In terms of the number of configurations examined, DeepGreedy performed the best as compared to ShallowGreedy and InlineGreedy for both the workloads. This might seem counter-intuitive – since we would expect that DeepGreedy, which is capable of examining a superset of transformations as compared to ShallowGreedy and InlineGreedy, would take longer to converge. However, this did not turn out to be the case, since DeepGreedy picked up the cost saving recursive merges fairly early on in its run. Consequently, this reduced the number of lower level merge and inline candidates in the subsequent iterations. This enabled DeepGreedy to converge faster. By the same token, we would expect ShallowGreedy to examine less number of configurations than InlineGreedy, but that was not the case. This is because ShallowGreedy was not able to perform any major cost saving merges since the "enabling" merges were never chosen individually. Hence, the same set of merge transforms were being examined in every iteration without any benefit, while InlineGreedy was not burdened with these candidate merges. But note that even though InlineGreedy converged faster, it was mainly due to the lack of useful inlines as reflected by the cost difference between InlineGreedy and ShallowGreedy.

6.5.4 Performance on Mixed Workloads

We now consider "mixed" workloads for both IMDB and DBLP – IMDB-MS (4 M-Queries and 7 S-Queries) and DBLP-MS (5 M-Queries and 5 S-Queries).

In order to control the dominance of S-queries vs. M-queries in the workload, we use a control parameter $k \in [0, 1]$ and give weight $(1-k)/7$ to each of the 7 S-queries and weight $(k)/4$ to each of the 4 M-queries for the IMDB-MS workload and a weight of $(1-k)/5$ and $k/5$ to each of the queries S- and M-queries in the DBLP workload, respectively.

We ran workload IMDB-MS with 3 different values of $k = \{0.1, 0.5, 0.9\}$. The cost of the derived configurations for IMDB-MS are shown in Figure 6.12. Expectedly, when S-Queries dominate, InlineGreedy performs quite competitively with DeepGreedy (with the cost of the configuration output by InlineGreedy being within just 15% of that output by DeepGreedy). But, as the influence of S-Queries reduce, the difference in costs increases substantially.

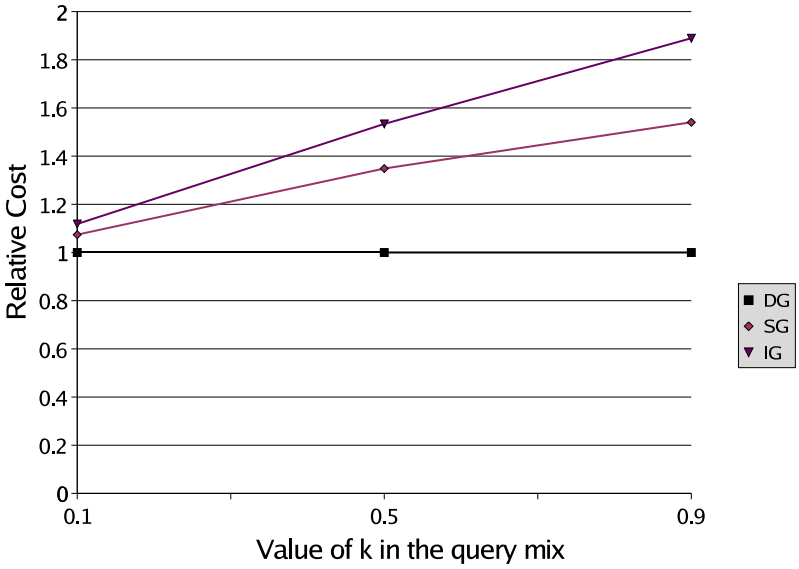


Figure 6.12: IMDB: Cost of Workloads Containing both M- and S-Queries

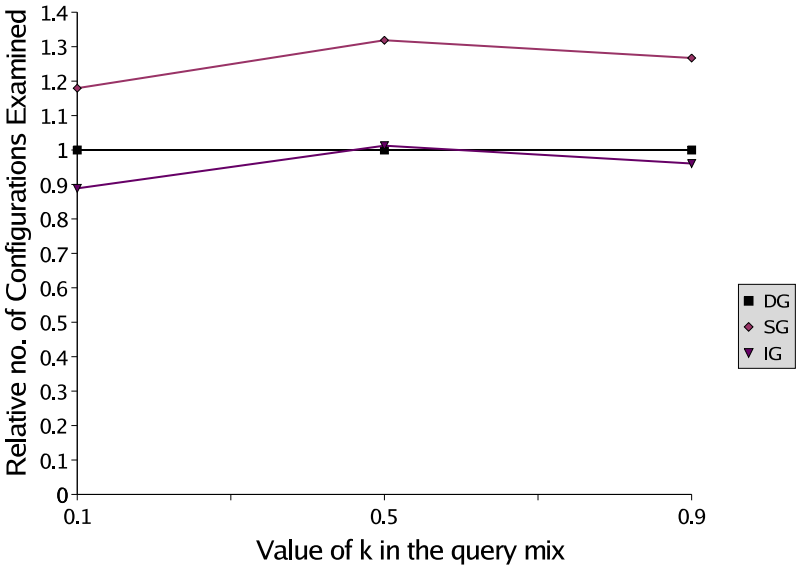


Figure 6.13: **IMDB:** No. of Configurations Examined for Workloads Containing M- and S-Queries

The number of configurations examined by all three algorithms are shown in Figure 6.13. DeepGreedy examines more configurations than InlineGreedy when S-Queries dominate, but the gap is almost closed for the other cases.

Note that both ShallowGreedy and InlineGreedy examine more configurations for k = 0.5 than in the other two cases. This is due to the fact that when S-Queries dominate (k = 0.1), cost-saving inlines are chosen earlier while when M-queries dominate (k = 0.9), both algorithms soon run out of cost-saving transformations to apply. Hence for both these cases, the algorithms converge faster.

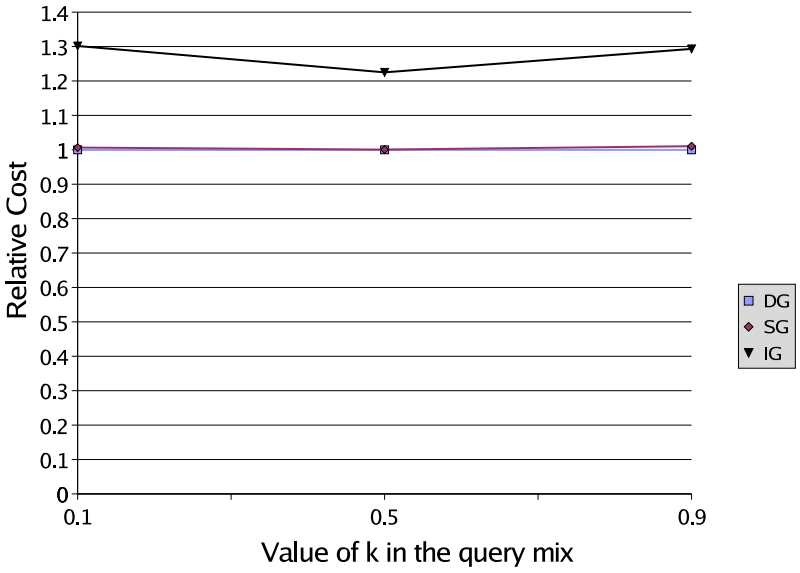


Figure 6.14: DBLP: Cost of Workloads Containing both M- and S-Queries

The relative costs of the configurations derived by DeepGreedy, ShallowGreedy and InlineGreedy for the DBLP-MS workload with k ={0.1, 0.5, 0.9} are shown in Figure 6.14. In contrast to the IMDB-MS workload, InlineGreedy is not at all competitive with DeepGreedy when S-Queries dominate. This is inspite of the fact that Figure 6.8 shows that InlineGreedy can be very competitive with respect to DeepGreedy. A closer look at the query workload revealed that in the case of the DBLP-MS workload, even though the weight of the S-Queries was higher (with a combined weight of 0.9), the *absolute* costs of a couple of M-Queries dominated the workload. That is, impact of the M-Queries had a larger influence on the final cost than that of the S-Queries. And so, InlineGreedy was not

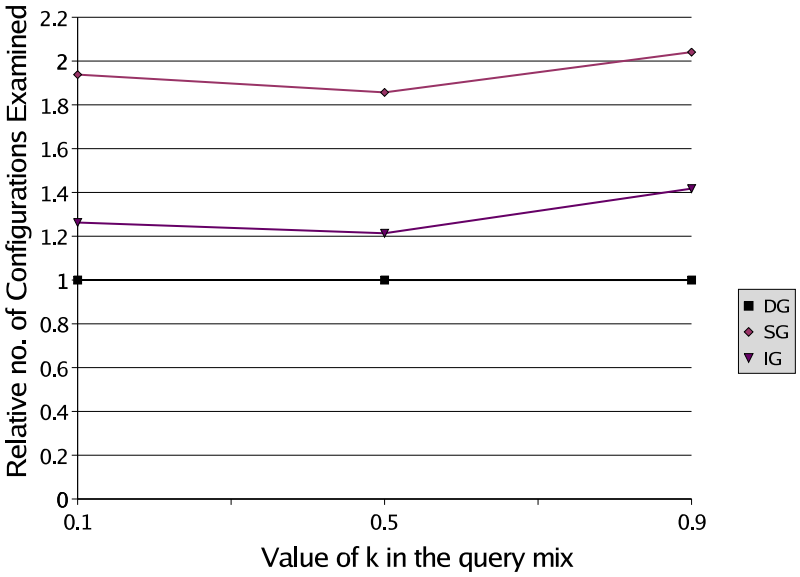


Figure 6.15: **DBLP:** No. of Configurations Examined for Workloads Containing M- and S-Queries

competitive with DeepGreedy. However, ShallowGreedy, which was highly competitive with DeepGreedy in both the M- and S-Queries cases had configuration costs within 1% of DeepGreedy.

The relative number of configurations for the DBLP-MS workload is shown in Figure 6.15. The trend of ShallowGreedy looking at a much larger number of configurations continued. The reason was the same as before – there are a smaller number of merges to consider because of the flat DBLP schema, but the merges that are considered by DeepGreedy considerably reduce the number of subsequent inlines. However, ShallowGreedy is unable to perform all the merges that DeepGreedy performs and hence the number of configurations examined by it remains large.

6.5.5 Comparison with Baselines

From the above sections, it is clear that except when the workload is dominated by S-queries, DeepGreedy should be our algorithm of choice among the algorithms proposed in this chapter. In this section we compare the quality of the relational configurations derived using DeepGreedy with the following baselines, the first four of which are non-cost-based:

1. **Fully Decomposed, All Outlined (FDAO):** Fully decompose the schema and outline all its types.

2. **Fully Decomposed, All Inlined (FDAI):** Fully decompose the schema and inline as many types as possible.

3. **Fully Merged, All Outlined (FMAO):** Retain the original schema and outline all its types.

4. **Fully Merged, All Inlined (FMAI):** Inline as many types as possible in the original schema.

5. **InlineUser (IU):** This is the same algorithm evaluated in [8].

6. **Optimal (OPT):** A lower bound on the optimal configuration for the workload given a specific set of transformations. Since DeepGreedy gives configurations of the best quality among the 3 algorithms evaluated, the algorithm to compute the lower bound consisted of transforms available to DeepGreedy. We evaluated this lower bound by considering each query in the workload individually and running an *exhaustive* search algorithm on the subset of types relevant to the query. Note that such a search is possible only if the number of types involved is very small since the number of possible relational configurations increases exponentially with the number of types. The exhaustive search algorithm typically examined several orders of magnitude more configurations than DeepGreedy.

We present results for two workloads, IMDB-MS and DBLP-MS which had a query mix of S- and M-Queries corresponding to $k = 0.5$.

The relative cost for each baseline is shown in Figure 6.16. As expected, none of the non-cost-based baselines are competitive with DeepGreedy. Moreover, InlineUser also compares unfavorably with DeepGreedy. Though InlineUser is good when there are not many shared types, it performs poorly if the schema has a few types which are shared or repeated or part of unions since there will not be too many types left to inline. This is specifically the case with the DBLP schema since most of the types like Article,

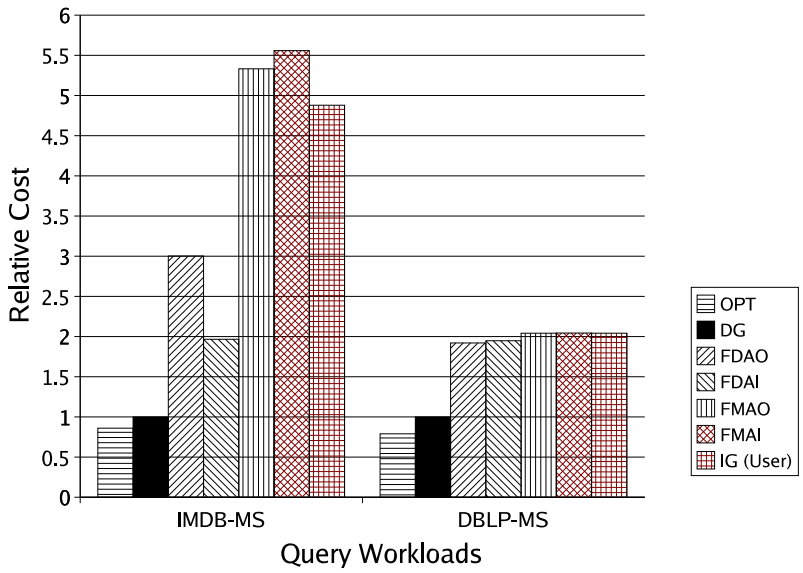


Figure 6.16: Comparison of DeepGreedy with the Baselines and Inline (User)

Inproceedings, etc. share many common attributes.The figures for the *lower bound on the optimal configuration* also show that DeepGreedy is within around 15% of the optimal for the IMDB schema and within 20% of the optimal for the DBLP schema.

6.6 Optimizations

There are several different optimizations that can be done to speed up the search algorithms. We propose a few of them here and outline their advantages and drawbacks.

6.6.1 Grouping Transformations Together

Recall that in DeepGreedy, in a given iteration, *all* applicable transformations are evaluated and the best transformation is chosen. In the next iteration, all the *remaining* applicable transformations are evaluated and the best one chosen (note that, when a transformation is applied, that transformation may *remove* the possibility of a few other transformations – for example, if a union factorization is performed, then, the type merges which would have been valid transforms *before* the union factorization are now no longer applicable). We found that in the runs of our algorithms, it was often the case that, in a

Input: $queryWkld$, $\mathcal{S}$
$queryWkld$ is the Query workload and $\mathcal{S}$ is the Initial Schema

```
 1: prevMinCost ← INFINITY
 2: rel_schema ← convertToRelConfig(S, queryWkld)
 3: minCost ← COST(rel_schema)
 4: while minCost < prevMinCost do
 5:    prevMinCost ← minCost
 6:    transforms ← applicableTransforms(S)
 7:    sortedTransforms = SORT(transforms)
 8:    for all T in sortedTransforms do
 9:       if applicable(T) then
10:          S' ← Apply T to S
             {S is preserved without change}
11:          rel_schema ← convertToRelConfig(S', queryWkld)
12:          Cost ← COST(rel_schema)
13:          if Cost < minCost then
14:             minCost ← Cost
15:             S ← S' {Retain the merge}
16:          else
17:             Goto step 5
18:          end if
19:       else
20:          Goto step 5
21:       end if
22:    end for
23: end while
```

Algorithm 8: GroupGreedy Algorithm

given iteration in which n transforms were applicable, if transformations T_1 to T_n were the best n transformations in this order (that is, T_1 gave the maximum decrease in cost and T_n gave the minimum decrease), other transformations up to T_i, for some $i <= n$, were chosen in subsequent iterations. This being the case, grouping transformations T_1 to T_i together has the potential to save several iterations. The number of transformations which can be grouped together depends on two factors: (i) whether the next transformation applied *reduces* the cost of the derived configuration further, and (ii) whether the next transformation is applicable after the current transformation is applied. Using these observations, we developed a variation of Algorithm 7, called *GroupGreedy* (Algorithm 8).

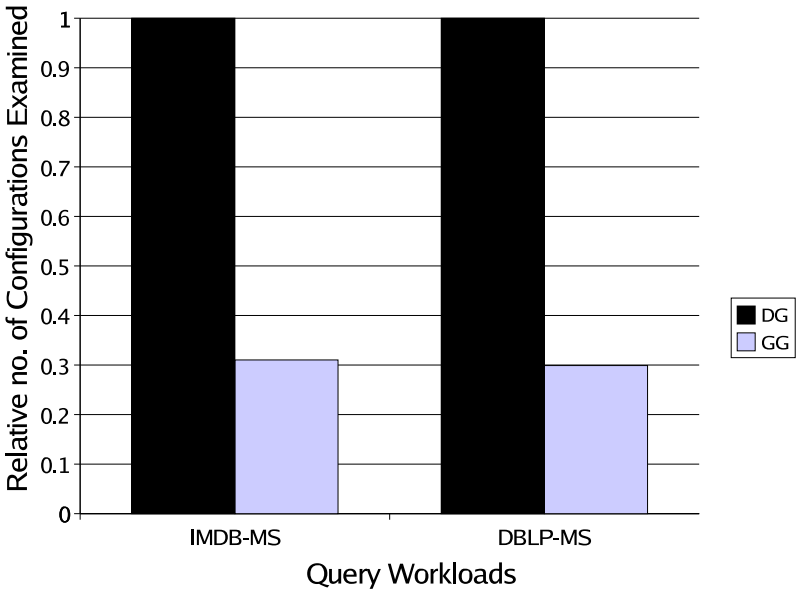


Figure 6.17: No. of Configurations Examined by DeepGreedy and GroupGreedy

We tried this optimization for DeepGreedy on the IMDB-MS and DBLP-MS workloads with $k = 0.5$. The cost of the final configuration derived by GroupGreedy (GG) was within 1% of DeepGreedy while examining approximately only about one third the number of configurations examined by DeepGreedy, as shown in Figure 6.17.

6.6.2 Early Termination

Another plausible optimization is to stop the algorithm once the decrease in the estimated cost goes below a small δ. This would save several iterations which are costly to perform, but do not give substantial decrease in cost. This optimization would be possible if the *magnitude of decrease* in cost is *monotonic*. However, during the course of our experiments, we came across several workloads which did not exhibit this behavior. The progress of DeepGreedy on such a workload, W, is shown in Figure 6.18.

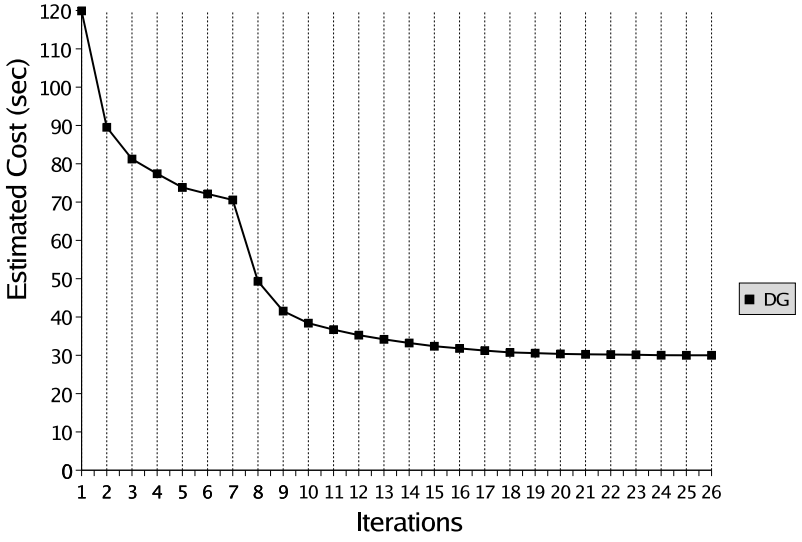


Figure 6.18: Progress of DeepGreedy on Workload W

Clearly, with an unfortunate value of δ, the algorithm would terminate at iteration 7 and miss the big cost decrease at iteration 8. Thus, while this optimization would result in improved execution times, the derived schema may not be as efficient as it could have been, if the algorithm were allowed to run to completion.

6.6.3 Applying Only Profitable Transforms

Though it is possible to decompose the schema fully by performing *all* union distributions, repetition splits and type splits, many of them may not be useful and so would simply increase the number of types in the schema. This is especially true for M-Queries where

many of the splits ultimately proved to be useless. Thus it would help if an apriori analysis of the query workload and the statistics can be used to cut down on the number of split transforms. This would reduce the number of combinations of merge transforms during search, and result in faster execution times.
As an example of such a heuristic, consider a repetition split in the following snippet:

define type Show {*element* SHOW {*type* Title, *type* Review*}}

It may be beneficial to split Review* into Review1? and Review2* only if a majority of the shows have exactly one review. Splitting Review in this way allows the inlining of the first review thus giving rise to potential cost benefits. However, if it is known that most shows have at least 10 reviews, then it is unlikely that inlining just one Review into Show would yield benefits. Nor would splitting Review help if most shows had no reviews, but a small number of shows had several reviews.

An interesting direction of future work would be to come up with heuristics based on the statistics available for the XML Schema to decide whether or not to perform a particular split operation.

6.6.4 Reducing the Search Space by Query Analysis

In order to compute our metric for the lower bound on the optimal, an exhaustive search was performed on single queries. This was made possible because the number of types relevant to the query was within reasonable limits. The same principle can be applied for the greedy algorithms as well. If the queries in the workload are concentrated to one particular part of the schema, then only those types need to be taken into consideration for the search. Or if the queries in the workload can be partitioned such that each partition has a disjoint set of types relevant to it, then the search can be performed separately for each partition.

6.7 Conclusions

In this chapter we introduced FleXMap, a cost-based system to derive relational storage schemas for XML. FleXMap is built on top of LegoDB and significantly extends this framework. In particular, we made the following contributions: (i) we studied the applicability of the transformations under different semantics such as syntactic equality and logical equivalence, the consequence of which was to derive more effective search algorithms such as DeepGreedy, (ii) proposed a mechanism for propagating statistics when a transformation is applied – this led to the conclusion that the schema has to be *fully decomposed* before the search can take place, (iii) evaluated the relative effectiveness of the algorithms DG, SG and IG under various types of query workloads as well as their effectiveness as compared to some previous approaches, and (iv) proposed optimizations to reduce the number of iterations in the greedy algorithms leading to an efficient new algorithm called GroupGreedy.

Our experimental evaluation showed that DeepGreedy achieves the best relational configurations overall, but the efficiency of the algorithm in finding that configuration depends on the query workload. If the query workload consists mainly of S-Queries, then InlineGreedy can achieve configuration quality which is close to that obtained by DeepGreedy, but by examining fewer number of configurations – this was true in the case of the IMDB dataset. In the case of predominantly M-Query workloads, not only did DeepGreedy achieve the best relational configuration, but did so by examining much fewer configurations than either ShallowGreedy or InlineGreedy. This was because the cost-saving merges were made earlier in the search, thus reducing the number of configurations which needed to be examined in subsequent iterations of the algorithm.

On an absolute scale, we compared the relational configuration output by DeepGreedy against a lower bound on the optimal configuration and showed that DeepGreedy is within 15%-20% of this lower bound. In comparison to previous results (heuristic as well as cost-based), we showed that DeepGreedy is far superior in the quality of configurations it outputs.

We also proposed optimizations to speed up the search process. In particular, the

GroupGreedy algorithm outputs configurations with cost within 1% of the configuration output by DeepGreedy, but performs much more efficiently than DeepGreedy – the number of configurations examined by GG is only about 20%-30% of the number of configurations examined by DG.

Chapter 7

Conclusions and Future Work

In this thesis, we presented a set of three tools, namely, (i) **StatiX**, (ii) **IMAX** and, (iii) **FleXMap**, to address the problems of XML statistics production, statistics maintenance and document storage, respectively. The basis for our solutions was the presence of an XML Schema and schema transformations.

We started off with StatiX, a framework for XML statistics collection and cardinality estimation. StatiX uses the XML Schema as the basis for statistics collection and summarizes the collected statistics in histograms. These histograms capture the skew of both the structure and values in the XML data. Several schema transformations can be used to make the summary more accurate. StatiX can currently support cardinality estimation for branching path expressions with value predicates. From an implementation view-point, statistics gathering for unambiguous schemas can piggy-back on validation, facilitating the reuse of standard XML validators. Experimental evaluation of StatiX on different data and query sets showed that highly accurate summaries can be built by applying schema transformations, while the size of the summary remains moderate. The summary size can be further reduced by a compression technique which eliminates some of the unnecessary structural histograms.

We introduced IMAX, a system which extends the schema-based statistics framework of the StatiX approach to incrementally handle updates to XML repositories. The novel challenges in the design of IMAX included developing techniques for accurately estimating

both the locations and the sizes of updates, as well as for the maintenance of structural histograms. To accurately estimate the location of updates, we extended the StatiX model with 2D histograms that capture the correspondence between the value of an element and its id. Our experiments to evaluate the utility of IMAX covered a variety of updates and datasets, and indicated that the accuracy of estimation from the updated statistics is very close to that obtained from the expensive brute-force option of re-computing the statistics from scratch. Further, these benefits can be obtained quite efficiently, requiring only rare recomputations of the summaries from the base data. In summary, IMAX makes sustained and efficient query processing feasible even in real-world XML environments whose contents are dynamically changing, which may become the norm in the coming years.

Moving on to data storage, we described a framework for exploring the space of XML-to-relational mappings and showed how schema transformations can be used to derive relational configurations. These transformations encompass physical database design strategies such as vertical and horizontal partitioning, through the use of inline/outline and union distribution, respectively. Since the framework searches the space of configurations in the XML world, it can be used with any other backend as long as the rules for translating the canonical schema are specified. We designed and implemented three greedy algorithms and studied how the quality of the final configuration is influenced by the transformations used and the query workload. We also proposed optimizations to speed up the time taken by the search algorithm with little loss in the quality of the final relational configuration. Experimental results show that our new algorithms provide significantly improved relational schemas as compared to those derived by previous approaches in the literature.

7.1 Future Work

There are several directions to extend the work described in this thesis. Cardinality estimation, which is currently limited to branching path expressions with value predicates can be extended to encompass other constructs such as the “for” and “let” bindings in

XQuery. A direct consequence of extending the cardinality estimation is its impact on statistics maintenance where more complex location components can be utilized in the update queries.

Currently, StatiX provides a framework for statistics collection. This framework can be effectively utilized to develop algorithms to build summaries based on a given memory budget. Such an algorithm can take into account several different factors: (i) the necessity of building structural histograms on certain types – we showed one minimization technique which removes the need for building structural histograms on several types, (ii) the statistics associated with a given type – it is possible that a particular type is *uniformly* distributed under its parent and does not require a structural histogram, and, conversely, another type may require the allocation of a large number of buckets, and, (iii) the query workload – a sample query workload can help in identifying *useful* transformations relevant to the workload.

An interesting direction of future work for statistics maintenance is the use of a *backup* summary. Currently, we have described the scenario where a single summary is being maintained. However, it may be useful for a more detailed, and consequently, larger summary to reside on disk. When updates are applied to the base data, both the smaller "primary" summary as well as the larger "backup" summary are updated. The primary summary is *recomputed* from the backup summary when its accuracy degrades. This technique would avoid recomputations from the base data and improve the efficiency of statistics maintenance.

Moving on to data storage, an important problem as yet unaddressed in the XML storage literature – especially cost-based storage – is the case when the application's query workload and/or statistics undergoes a significant change. So far, storage design has been considered to be a one-time operation. However, it may be necessary to update the storage as and when the application characteristics undergo changes. If changes to the backend configuration (in terms of the backend schema) are expensive to implement (as is the case in the schema evolution of relational schemas), then alternatives such as the appropriate set of views and indexes to be built have to be considered. The novelty of

performing schema evolution for XML-on-relational is that the cost-benefit tradeoff needs to be taken into account in the *XML domain* through schema transformations (otherwise, several XML-specific optimizations in both relational schema generation as well as query translation may go unnoticed).

In closing, this thesis presented a toolkit for effectively supporting the highly popular XML world-view on the underlying storage and processing engines.

References

[1] A. Aboulnaga, A.R. Alameldeen, and J.Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.

[2] Amazon.com. `http://www.amazon.com`.

[3] D. Barbosa, J. Freire, and A. Mendelzon. Information preservation in XML-to-relational mappings. In *Proceedings of the XML Database Symposium (XSym)*, 2004.

[4] D. Barbosa, J. Freire, and A. Mendelzon. Designing information-preserving mapping schemes for XML. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2005.

[5] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: An extensible template-based data generator for XML. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2002.

[6] P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. LegoDB: Customizing relational storage for XML documents. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002.

[7] P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Bridging the XML-relational divide with LegoDB: A demonstration. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2003.

[8] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations:

A cost-based approach to XML storage. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2002.

[9] Caml language. `http://caml.inria.fr/`.

[10] CDuce. `http://www.cduce.org/`.

[11] Y. Chen, S. Davidson, C. Hara, and Y. Zheng. RRXS: Redundancy reducing XML storage in relations. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.

[12] Y. Chen, S. Davidson, and Y. Zheng. Constraint preserving XML storage in relations. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2002.

[13] Z. Chen, S. Chaudhuri, K. Shim, and Y. Wu. Storing XML (with XSD) in SQL databases: Interplay of logical and physical designs. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2004.

[14] Z. Chen, H.V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R.T. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2001.

[15] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 2002.

[16] S. Davidson, W. Fan, C. Hara, and J. Qin. Propagating XML constraints to relations. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2003.

[17] DB2 XML extender. `http://www.ibm.com/software/data/db2/extenders/xmlext/`.

[18] DBLP. `http://dblp.uni-trier.de/xml`.

[19] A. Deutsch, M. Fernandez, and D. Suciu. Storing semi-structured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1999.

[20] Document object model. `http://www.w3.org/DOM/`.

[21] F. Du, S. Amer-Yahia, and J. Freire. A comprehensive solution to the XML-to-relational mapping problem. In *Proceedings of the ACM International Workshop on Web Information and Data Management (WIDM)*, 2004.

[22] F. Du, S. Amer-Yahia, and J. Freire. ShreX: Managing XML documents in relational databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[23] Extensible Markup Language (XML). `http://www.w3.org/XML`.

[24] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML in a relational database. Technical Report 3680, INRIA, 1999.

[25] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[26] J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making XML Count. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2002.

[27] J. Freire, M. Ramanath, and L. Zhang. A flexible infrastructure for gathering XML statistics and estimating query cardinality. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2004.

[28] Galax. `http://www.galaxquery.org`.

[29] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson Education, Inc, 2002.

[30] P. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM Transactions on Database Systems (TODS)*, 27(3), 2002.

[31] HyperText Markup Language (HTML) Home Page. `http://www.w3.org/MarkUp/`.

[32] Internet Movie Database. `http://www.imdb.com`.

[33] ISO 8879. Information Processing – Text and Office Systems - Standard Generalized Markup Language (SGML), 1986.

[34] H. V. Jagadish, S. A.-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *The VLDB Journal*, 11(4), 2002.

[35] H. Jiang, H. Lu, W. Wang, and J. X. Yu. Path materialization revisited: An efficient storage model for XML data. In *Proceedings of the Australasian Database Conference (ADC)*, 2002.

[36] H. Jiang, H. Lu, W. Wang, and J. X. Yu. XParent: An efficient RDBMS-based XML database system. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2002.

[37] R. Krishnamurthy, V. Chakaravarthy, and J.F. Naughton. On the difficulty of finding optimal relational decompositions for XML workloads: A complexity theoretic perspective. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2003.

[38] D. Lee and W. Chu. CPI: Constraints-preserving inlining algorithm for mapping XML DTD to relational schema. *Journal of Data and Knowledge Engineering (DKE)*, 39(1), 2001.

[39] P. Lehti. Design and implementation of a data manipulation processor for an XML query language. Master's thesis, Universität Darmstadt, 2001.

[40] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Parr. XPathLearner: An online self-tuning markov histogram for XML path selectivity estimation. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002.

[41] Making web services work at Amazon. `http://www.xml.com/pub/a/2003/12/09/xml2003amazon.html`.

[42] M. Mani and D. Lee. XML to relational conversion using theory of regular tree grammars. In *Proceedings of the Workshop on Efficiency and Effectiveness of XML Tools and Techniques (EEXTT)*, 2002.

[43] Microsoft SQLXML. `http://msdn.microsoft.com/sqlxml`.

[44] M. Muralikrishna and D. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1988.

[45] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.

[46] Natix XML repository. `http://www.dataexmachina.de/natix.html`.

[47] M. Nicola and J. John. XML parsing: a threat to database performance. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, 2003.

[48] Oasis - the coverpages. `http://www.oasis-open.org/cover`.

[49] Oracle XML DB. `http://www.oracle.com/technology/tech/xml/xmldb/index.html`.

[50] Oracle's XML SQL utility. `http://technet.oracle.com/tech/xml/oracle_xsu`.

[51] R. Paige and R. Tarjan. Three paritition refinement algorithms. *SIAM Journal on Computing*, 16(6), 1987.

[52] P. Patil. Holistic source-centric schema mappings for XML-on-RDBMS. Master's thesis, Indian Institute of Science, 2005.

[53] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1984.

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

[55] N. Polyzotis and M. Garofalakis. Structure and value synopses for XML data graphs. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002.

[56] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity estimation for XML twigs. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2004.

[57] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1997.

[58] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1996.

[59] M. Ramanath, J. Freire, J. Haritsa, and P. Roy. Searching for efficient XML-to-relational mappings. In *Proceedings of the XML Database Symposium (XSym)*, 2003.

[60] M. Ramanath, L. Zhang, J. Freire, and J. Haritsa. IMAX: Incremental maintenance of schema-based XML statistics. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2004.

[61] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2000.

[62] C. Sartiani. A framework for estimating XML query cardinality. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2003.

[63] C. Sartiani. A general framework for estimating XML query cardinality. In *Proceedings of the International Workshop on Database Programming Languages (DBPL)*, 2003.

[64] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2000.

[65] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1999.

[66] Tamino. `http://www.softwareag.com/tamino/architecture.htm`.

[67] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2001.

[68] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2002.

[69] F. Tian, D. DeWitt, J. Chen, and C. Zhung. The design and performance evaluation of alternative XML storage strategies. *SIGMOD Record*, 31(1), 2002.

[70] W3C XML query. `http://www.w3.org/XML/Query`.

[71] W3C XML schema. `http://www.w3.org/XML/Schema`.

[72] W. Wang, H. Jiang, H. Lu, and J.X. Yu. Bloom histogram: Path selectivity estimation for XML data with updates. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[73] World wide web consortium. `http://www.w3c.org`.

[74] X. Wu, M.-L. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered XML trees. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2004.

[75] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2002.

[76] Xerces Java parser 2.5.0. `http://xml.apache.org/xerces-j/`.

[77] XML Applications and Initiatives. `http://xml.coverpages.org/xmlApplications.html`.

[78] XML database products: Native XML databases. `http://www.rpbourret.com/xml/ProdsNative.htm`.

[79] XML path language (XPath). `http://www.w3.org/TR/xpath`.

[80] XQuery update facility requirements. `http://www.w3.org/TR/xquery-update-requirements/`.

[81] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology (TOIT)*, 1(1), August 2001.

[82] S. Zheng, J.-R. Wen, and H. Lu. Cost-driven storage schema selection for XML. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, 2003.