UNIVERSITY OF CINCINNATI

Date: 16-Nov-2009

I, Bartley D Richardson **,**

hereby submit this original work as part of the requirements for the degree of:

Doctor of Philosophy

in Computer Science & Engineering

It is entitled:

A Performance Study of XML Query Optimization Techniques

Student Signature: Bartley D Richardson

This work and its defense approved by:

Committee Chair: Karen Davis, PhD
Karen Davis, PhD

Raj Bhatnagar, PhD
Raj Bhatnagar, PhD

John Schlipf, PhD
John Schlipf, PhD

Fred Annexstein, PhD
Fred Annexstein, PhD

Hsiang-Li Chiang, PhD
Hsiang-Li Chiang, PhD

A Performance Study of XML Query Optimization Techniques

A dissertation submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science
of the College of Engineering

November 2009

by

Bartley Douglas Richardson
B.S., University of Cincinnati
June 2003

Dissertation Advisor and Committee Chair:
Karen C. Davis, Ph.D.

Abstract

As computers and technology continue to become more commonplace and essential to everyday life, more data is captured, stored, and analyzed by a variety of institutions in government, education, and the private sector. As this amount of data grows, so does the need for efficient methodologies and tools used to store, retrieve, and transform the data. A common method used to store this schemaless, semi-structured data is through the Extensible Markup Language, XML. In this way, an XML document is viewed as a database. With this sizable amount of data stored in a common format, one problem is how to efficiently query XML documents. While relational database management systems contain built-in query optimizers, no such framework exists for XML databases. A multitude of document shapes, query shapes, index structures, and query techniques exist for XML databases, but the implications of these choices and their effects on query processing have not been investigated in a common framework. This dissertation identifies a set of representative query techniques, document structures, and query styles for XML databases and provides a common framework for classifying the various query techniques, structures, and styles. We identify two broad classifications of query techniques, native XML and non-native XML, and develop a cost-based model for each technique that models query performance from an execution standpoint. We also develop our own query technique, RDBQuery, as an extension and major enhancement to a previously existing non-native XML query technique that leverages a relational database management system to efficiently process XML queries. To evaluate relative query performance, we compare the techniques for various parameters that impact their performance, including query shape and document shape/size, and the results are presented through a series of graphs. These graphs and their underlying cost models are used to present an optimization framework for XML queries, and this provides the essential foundation in development of an integrated cost-based XML query optimizer.

Acknowledgements

First and foremost, I would like to thank Dr. Karen Davis for her constant guidance over the past six years. She has been and will continue to be an amazing source of knowledge and support, and I consider her my greatest academic role model. I have learned so much from Dr. Davis that it would be difficult to contain everything in this brief section. She has taught me how to be an effective researcher and provided me with invaluable feedback and comments on my work. I could sit in my office and think about a problem for hours, but all it would typically take to make the answer crystal clear is a single question or comment from her. In addition to research, I use Dr. Davis as a role model when teaching my undergraduate courses. Her ability to continually push for the best from her students while simultaneously providing immense support for them is something I strive to model in my instruction. I am the researcher and teacher I am today because of her, and I can think of no better person to serve as my mentor.

I would also like to thank Dr. Fred Annexstein, Dr. Raj Bhatnagar, Dr. Hsiang-Li Chiang, and Dr. John Schlipf for sitting on my committee, dedicating their time to read my dissertation, and providing me with their comments and valuable suggestions. In addition, I would like to think Dr. Anant Kukreti for affording me my first professional teaching experience. The teaching positions I have had since all built on that solid foundation.

I am thankful to my employer, Thomas More College, and the immense support they have shown me over the past year while finishing this dissertation. A special thanks go to Dr. Jim Swartz, the entire Computer Information Systems Department, and Dr. Brad Bielski. Thank you for placing your confidence in me and allowing me to teach at Thomas More.

Without the support of my friends, I would not be where I am today. I would like to thank all of my friends in the College of Engineering for not only their friendship and support but also for their willingness to help me with difficult problems and then go play some poker. To my friends at Mercy Healthplex, Northern Kentucky University, and Thomas More College, thank you for being there for me and your many welcomed distractions from work. An special note of thanks goes to my friends Amy Dimmerling and Nico Gonzalez. You both found so many ways to support me

through times both good and rough, and I cannot express how fortunate I am to have both of you in my life.

I would like to thank my family for their unwavering support and unconditional love. To my parents, Sarah and Jerry Richardson, who taught me everything I know about determination and hard work, I am where I am today because of you. I look to both of you as my personal heroes, and I know that I am a teacher today because of your example. Although I will probably have to read this to him, I would like to thank Smoke, my Ragdoll/Maine Coon mix cat, for his constant companionship during my graduate work. Last but certainly not least, I would like to thank my girlfriend, Misty Laderer, for her immense love and frequent help with tough problems, both in research and in life. She has learned more than she ever wanted to know about computers in her constant willingness to help me when it seemed as thought I had too much work to bear on my own. Her ability to decipher my hand-drawn diagrams and create beautiful computer-generated figures is nothing short of a miracle. Misty has given me so much support through tough times, and she has shared with me joy and happiness of good times. I am extremely grateful and fortunate to have such a loving woman in my life, and I look forward to our long and happy future together.

Any questions?

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

As computers and technology become more commonplace and essential to everyday life, more and more data is captured, stored, and analyzed by a variety of institutions in government, education, and the private sector. As this amount of available data grows, so does the need for efficient methodologies and tools used to store, retrieve, and perform operations on the data. The relational model was first proposed by Codd in 1970 [Cod70] as a way of describing data using only its natural structure. Specifically, the natural structure of the data refers to the relations between data elements. It is based on the notions of set theory and first order predicate logic and has, at its core, the idea of a mathematical relation as the basic building block. Data in the relational model must conform to a global schema (a description of the type or structure of the data). A relational schema is typically developed by a database administrator before data is loaded into the system.

As the relational model gained popularity, it inspired many end-user database management systems (DBMS) to be created using it as a theoretical backbone. Since relational algebra (the mathematical notation used to manipulate relational data) can be complex, a higher-level query language was developed to ease user interaction with the DBMS. The Structured Query Language (SQL) was standardized by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) in 1986 [ANS86]. This version of SQL was revised and expanded in 1992 and is commonly referred to as SQL-92. While SQL allows complex queries to be written and executed, it does not optimize queries to improve performance and query return times.

In order to improve query return time, commercial DBMS packages currently include query op-

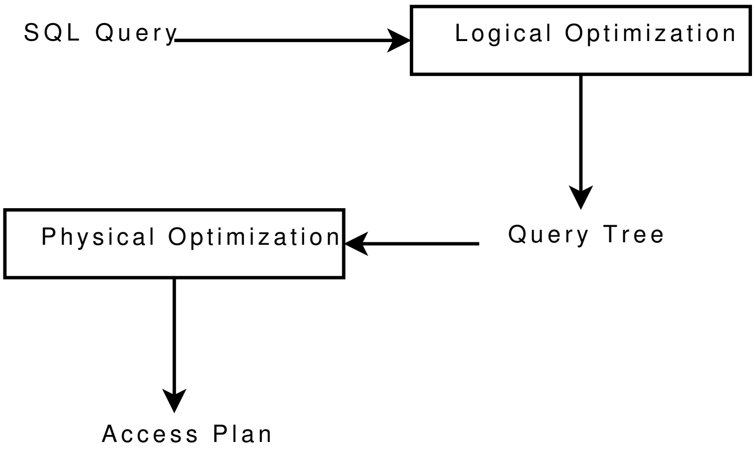


Figure 1.1: Traditional Query Optimization

timization techniques built-in to the software. These types of optimizations fall into two categories: logical and physical (Figure 1.1). When a SQL query is presented to the database, the first step is logical optimization. The high-level SQL query is converted to a corresponding relational algebra tree. Transformations are then performed on the tree in order to optimize the query, i.e., reduce the data retrieved and operated on. The goal of logical optimization is to rewrite the user query into an equivalent form that is more efficient to execute. For example, Figure 1.2 shows the result of logical optimization.

While Figure 1.2 shows a query tree, we can intuitively discuss the operations performed on the query tree represented. Before logical optimization, the cross product (represented by the × symbol) of relations S and T is formed. Then a selection (σ) is performed on the data to retrieve specific rows from the cross product. Finally, unwanted columns are projected out (π) and the final answer set is given. Since the cross product matches every record in S with every record in T, the resulting answer will be very large. In addition, the time needed to compute this large cross product will be lengthy. The result of logical optimization (shown to the right of the arrow in Figure 1.2) is an equivalent query tree that is faster to process. Assuming the selection (σ) has some conditions that operate only on S and others that operate only on T, those conditions can be pushed down the tree past the cross product. This will reduce the number of rows involved in the cross product. In addition, the projection (π) can be moved past the cross product as well. Columns in S and columns in T that are not required in the cross product can be removed before it

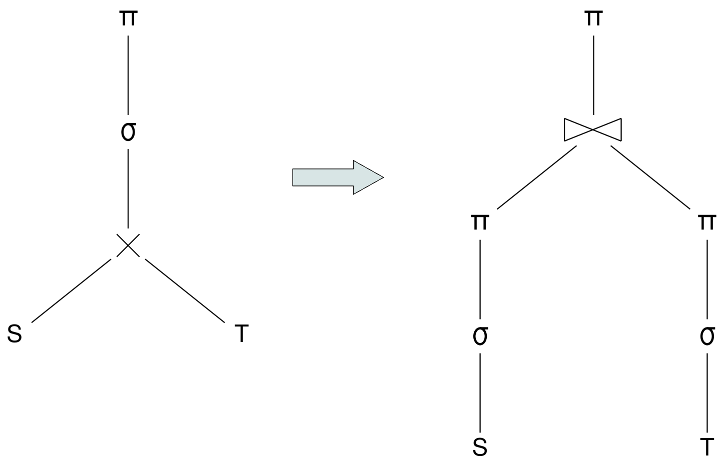


Figure 1.2: Logical Optimization (Relational Algebra)

is computed. The cross product (×) and the remaining selections (σ) that operate on both S and T are then converted into the join operation (shown in the figure by $\bowtie$). Finally, any remaining unwanted columns are projected out (π) of the final answer.

The result of logical optimization is an equivalent query tree, and this tree is then passed on for physical optimization. Physical optimization takes into account file organization and auxiliary access and mechanisms. How the data is stored on disk and the indexes or other access methods available to the database are crucial in retrieving the requested data quickly. A result of physical optimization is shown in Figure 1.3. Each of the operators has been assigned an access procedure based on the physical storage scenario.

For example, each of the operators from Figure 1.3 is assigned an access method (procedure). Since an index (presumably a B$^+$-tree index) is built on S, the optimizer uses this index for the selection (σ). Since no index exists on T, the optimizer instead uses a hash function. If T is small, a linear scan (used for the π operator) is sufficient to project out unwanted data. Other access methods, determined by availability and cost to the system, are assigned to the remaining operators accordingly. The DBMS is aware of the physical storage and auxiliary access methods available to the system. Since there is always a cost to access the data on disk, choosing an efficient access plan among all possible choices is referred to as cost-based optimization.

The relational model and associated optimization techniques are mature technologies. When data is highly-structured and uses a well-defined schema, relational databases are an excellent choice

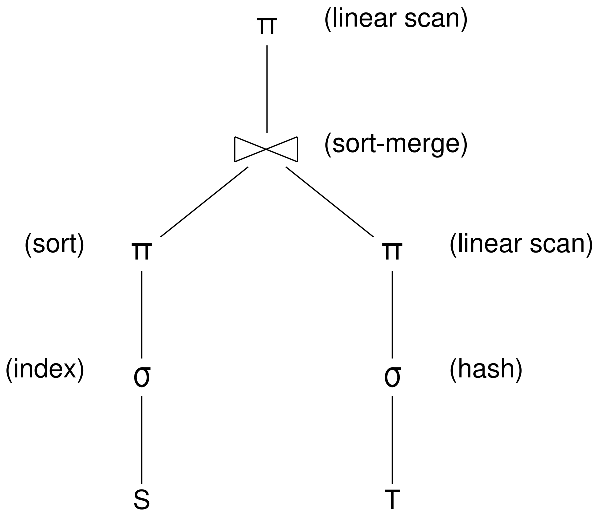


Figure 1.3: Physical Optimization (Relational Algebra)

for storing and accessing data. However, with the growth of the Internet in the past decade, new ways of structuring and describing data have become available. One such data model, XML, is discussed below. These new types of data present challenges for traditional query processing and optimization techniques.

1.1 XML and OEM

Most data on the web is said to be semistructured or loosely-structured data as well as schemaless or self-describing. In other words, unlike data in the relational model, there exists little or no metadata [ABS00] separate from the data itself. The Extensible Markup Language (XML) is a new standard for data exchange on the Internet and between different processing platforms. An open-standard specification for XML is kept by the W3C [xml]. While XML is syntactically similar to HTML, it does more than simply specify the appearance of text on a page. Data represented in XML is self-describing, i.e., it contains embedded descriptive information, and generally does not require an outside schema.

A brief example of an XML document is shown in Figure 1.4. Information is represented both in the text and the tags around the text. The two main methods to represent data are as elements or attributes. An example of an element if shown in line 3 of Figure 1.4. The element identifier is

```
<FoodDrink>
  <restaurant id=‘‘R001’’>
    <name>Chili’s</name>
    <phone>671-1102</phone>
    <owner>G. Peppard</owner>
  </restaurant>
  <restaurant id=‘‘R002’’>
    <name>Maggiano’s</name>
    <owner>G. Peppard</owner>
    <manager>Crowley</manager>
  </restaurant>
  <bar id=‘‘B001’’>
    <name>Crowley</name>
    <style>Irish</style>
  </bar>
</FoodDrink>
```

Figure 1.4: XML Example

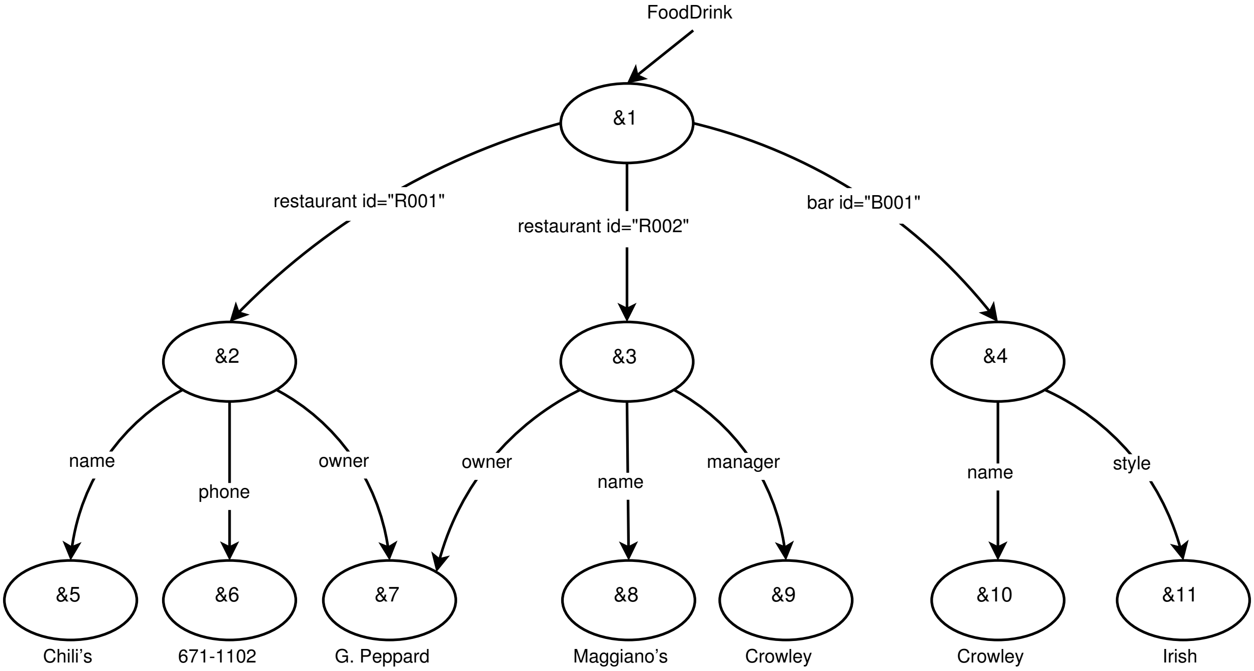


Figure 1.5: Corresponding OEM Representation

`name`, and the corresponding element value is `Chili's`. Information can also be represented as an attribute of an element (as shown in line 2). The element `restaurant` has an attribute of `R001`. The nesting of XML elements gives it a tree (or graph) structure, and this yields information about hierarchical relationships (such as parent-child or ancestor-descendant) in the data.

While XML is robust and highly-adaptable (attributes, elements, and element tags can be dynamically specified and defined by the user), it can be somewhat daunting to read and understand. The Object Exchange Model (OEM) was proposed in 1995 [PGMW95], and it serves as a diagrammatical representation for XML documents. Data represented in OEM is self-describing and therefore does not require additional schema definitions. An object in OEM is defined as the quadruple `(label, oid, type, value)`. The variable `label` gives a character label to the object, `oid` provides the object's unique identifier, and `type` can be either an atomic value or `complex`. If `type` is an atomic value, then the object is an atomic object and `value` is an atomic value of the corresponding type. Otherwise, if `type` is `complex`, then the object is a complex object and `value` is a list of object identifers (oids) [ABS00]. An OEM diagram that corresponds to the XML example is shown in Figure 1.5. The OEM retains the simplicity of relational models but allows some of the flexibility given by object-oriented models [CBB$^+$97] for specifying nested objects. OEM is one example of a graphical convention used to display an XML document. It is important because the document has an inherent structure, data labels, and data that are readily visible to the reader. A similar graphical construct will be used to illustrate examples shown in our work.

1.2 XPath and XQuery

The simplest type of query in XML is an XPath expression [xpa09]. XPath expressions resemble the UNIX directory structure with some extensions. The slash (/) and double-slash (//) retain their UNIX interpretations (parent-child and ancestor-descendent relationship, respectively), and the text in brackets (`[ ]`) acts as a filter on the data to be returned. Examples in this research are specified in XPath expressions. An example of a simple XPath expression is given by `/FoodDrink/Restaurant[owner='G.Peppard']` and corresponds to the XML document shown in Figure 1.4. This expression results in a positive match to two restaurant nodes, one with `id` equal to `R001` and the other with `id` equal to `R002`. The single slash represents a strict parent-child relationship. The expression `//[style='Irish']` matches only one node, the `bar` node with `id` equal to `B001`. The double-slash represents an ancestor-descendant relationship. In this case, we are only interested in nodes that, at some point in their list of descendants, has a `style` of `Irish`.

XQuery is a query language for XML designed to be broadly applicable across many types of XML sources [xqu09]. Designed to meet the requirements identified by the World Wide Web Consortium (W3C), XQuery operates on the logical structure of an XML document, and it has both human-readable syntax and XML-based syntax. A grammar for XQuery is defined by the W3C [xqu09]. While XQuery can successfully extract information from XML documents, there are no built-in optimization techniques that relate to the relational optimization techniques discussed earlier. The current version of XQuery (1.0) is an extension of XPath 2.0. For our purposes, XPath expressions convey the necessary ideas and XQuery will not be used here.

1.3 Native and Non-Native Techniques

There currently exists two broad methodologies, native and non-native techniques, used to query XML documents. Native techniques implement XML queries on XML documents. The original document, while perhaps slightly transformed, maintains the inherent properties of an XML document. This means that the document is tree shaped, has both depth and breadth, and is constructed by linking individual nodes (elements) together. In contrast, non-native techniques transform the original XML document into another format that is not XML. An example of a non-native technique is to take an XML document, flatten it, and store the contents in a relational database. Some of these techniques allow standard XPath expressions to be executed over the transformed data, but the underlying document is no longer an XML file.

1.4 Problem Statement

As a new and evolving model for representing semistructured data, XML presents new challenges and options for query processing and cost-based optimization. A multitude of tree shapes, query styles, query models, index styles, and index data structures exist for XML databases, but the implications of these choices and their effects on query processing have not been investigated. The problem of creating the framework and foundation for an effective cost-based optimizer that can leverage various XML-related parameters has not been studied.

1.5 Research Objectives

The general objective of this research is to investigate options for and develop the foundation framework for a unified cost-based optimizer for XML query processing. Our work focuses on the analyses of several representative query techniques and the comparisons between them. The increasing volume of semistructured data available on the Internet and other areas makes such an objective relevant and necessary. Specific objectives of this research related to this goal are as follows.

1. It is necessary to identify and characterize a set of representative query styles, tree shapes (database statistics), and index styles and structures. No common framework and terminology exists for characterizing common representative XML queries that can be presented to a document.

2. A representative set of query evaluation techniques are selected and analyzed. Each method is formally measured as to its effectiveness in producing results to the query styles and tree shapes mentioned above. A cost model for each technique is developed to aid in evaluation.

3. The results of the analyses above are presented in a series of graphs/plots to examine the effects of individual parameters.

4. General conclusions and recommendations are proposed that address which algorithm best performs given a particular query style and tree shape.

5. An optimization framework for XML queries is proposed.

1.6 Research Approach

After our representative set of query evaluation techniques are selected, we develop a cost model for each technique that allows us to model its behavior mathematically. We utilize Wolfram Mathematica, a powerful software package that allows for complex equations and graphs, to study the effect of each parameter in the individual query techniques. Native techniques are compared to each other, and non-native techniques are similarly studied. The leading technique from each category is then selected and compared, and a general recommendation about the technique that outperforms the others in particular scenarios is made.

1.7 Overview of Chapters

In Chapter 2, we discuss related work and techniques on which this research is based. Chapter 3 provides a detailed description of TwigStack [BKS02]. We analyze the TwigStack algorithm and develop a cost model for the technique. In a similar fashion, we discuss Constraint Sequencing [WM05] in Chapter 4. The encoding technique and potential problems with queries are presented, and we create a cost model for this technique. Chapter 5 provides a detailed discussion about a non-native XML query technique that stores XML data in relational databases. A leading technique, SS-Join [SLFW05], is presented and a cost model developed. We also present our own algorithm, RDBQuery, that uses the same underlying premise as SS-Join but utilizes the relational database query optimizer to aid in efficient query processing. In Chapters 7 and 8, we present detailed analyses of individual native and non-native techniques, respectively. Our experimental results are discussed using graphs generated by our cost models. The native XML query techniques are compared in Chapter 9. The native technique that outperformed the other technique is then compared to RDBQuery in Chapter 10.

Chapter 2

Related Work

This chapter discusses research literature regarding indexing and querying XML data. We begin with a brief historical summary of indexing techniques, then identify a technique, TwigStack, that out-performs the historical techniques. The chapter concludes with an overview of an alternative technique, Constraint Sequencing, that encodes both the document and the query and performs pattern matching to evaluate queries. TwigStack and Constraint Sequencing are studied in more detail in later chapters.

2.1 Indexing Techniques

Indexing structures used in relational databases are well-known and highly efficient. Using these indexing structures as a starting point for indexing XML documents, a natural evolution in the features and efficiency of said indexes has occurred and will likely continue to develop. This section starts by introducing a labeling scheme for nodes in a tree, presents preliminary index structures (B^+-tree and XR-tree) used for XML documents, moves on to more sophisticated and efficient index methodologies (XB-tree, DataGuide, and ToXin).

2.1.1 Node Labeling

When constructing a B^+-tree, XR-tree, or XB-tree index on an OEM structure, the nodes must be labeled with a standard labeling scheme. Many labeling methods exist [HR05], but the most common and widely-used is an extension to Dietz's numbering scheme (tree traversal order [Die82]) called extended preorder traversal [LM01]. Using this labeling method, each node in the tree is labeled with a pair of numbers `<order,size>`. This extension allows insertions to be made into the tree without the need for global reordering. It maintains the original idea of Dietz's scheme by imposing three conditions on the values for `order` and `size`.

1. For a tree node y and its parent x, $order(x) < order(y)$ and $order(y) + size(y) \leq order(x) + size(x)$. In other words, the interval $[order(y), order(y) + size(y)]$ is contained in the interval $[order(x), order(x) + size(x)]$.

2. For two sibling nodes x and y, if x is the predecessor of y in preorder traversal, then $order(x) + size(x) < order(y)$.

3. For any node x,
$$size(x) \geq \sum_{y} size(y)$$
for all y's that are a direct child of x.

By using an arbitrarily large integer for $size(x)$, future insertions into the structure can be made without the need for global reordering. Using Figure 1.5 as a starting point and with $size(x) = 100$, appropriate node labels are generated and shown in Figure 2.1. This set of labels is not the only possible set of labels for the OEM tree. Other equally valid sets exist.

2.1.2 B$^+$-, XR-, and XB-Trees

In relational database systems, the *B$^+$-tree* (a variation of the B-tree) is used to implement a dynamic multilevel index [EN00]. Offering advantages to indexed sequential files, a B$^+$-tree does not require reorganization of the entire file to maintain performance. In other words, the tree will automatically reorganize itself with small, local changes when insertions and deletions occur. Due to its hierarchical nature, the B$^+$-tree was used in an algorithm for processing XML structural joins [CVZT02]. Although structural joins are discussed in greater detail in a later chapter, it is sufficient to mention that they require information about ancestors and descendants of a given element (possibly through multiple levels). For this reason, an algorithm and index structure that allows ancestors and descendants to be found and evaluated quickly will improve performance of

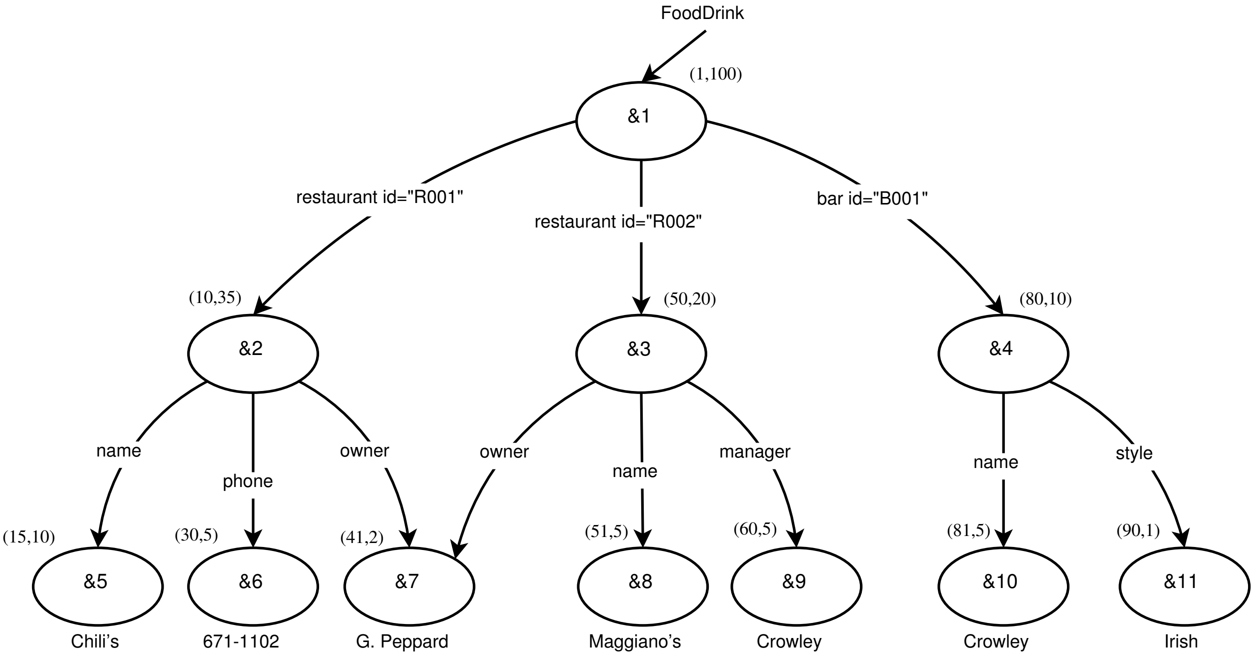


Figure 2.1: OEM Representation with Intervals

structural joins. While it showed an improvement over a previous algorithm using R-trees for the same purpose, the B$^+$-tree was later improved upon to produce the XR-tree and later the XB-tree.

The *XR-tree* [JLWO03], known as the XML Region Tree, is a B$^+$-tree that is built on the start points of the element intervals. Designed for strictly nested XML data, this type of index structure allows all ancestors and descendants for a given element to be identified with optimal worst case disk input/output cost. The XR-tree outperforms the B$^+$-tree for processing structural joins, but it lacks the capability to handle highly recursive XML elements with the same efficiency [LLHC04].

The *XB-tree* was developed by Bruno et al. [BKS02] for use in processing holistic twig joins (a specialized version of structural joins). The XB-tree combines the structural features of both the B$^+$-tree and the R-tree. It indexes the pre-assigned intervals of elements in the tree (similar to a one-dimensional R-tree) and then constructs the index on the start points of the intervals (similar to the standard B$^+$-tree) [LLHC04]. The main difference is that the *size* portion of the `<order,size>` label must be propagated up the index. A sample XB-tree formed using Figure 2.1 is shown in Figure 2.2. The main advantage of the XB-tree is that it quickly processes requests to find ancestors and descendants. A performance study [LLHC04] found that the XB-tree outperforms

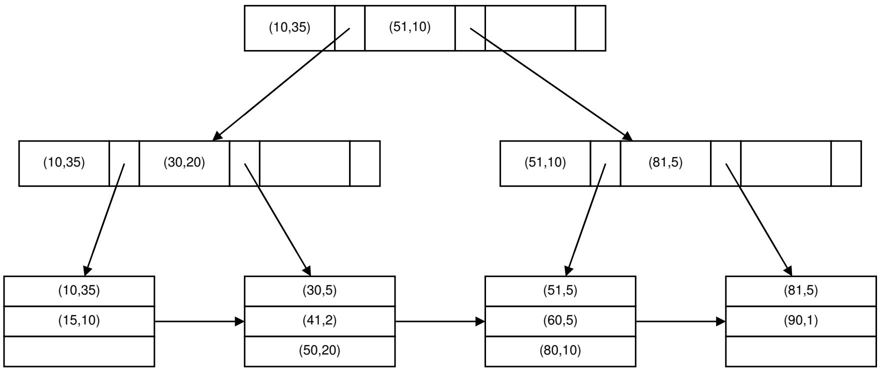


Figure 2.2: Sample XB-tree Using Figure 2.1

both the B$^+$-tree and XR-tree for processing structural joins in XML documents.

2.1.3 DataGuide

Moving away from indexes based on traditional methodologies, *DataGuides* provide a visual way to summarize information contained in an OEM source document. At its most basic level, a DataGuide [GW97] is a concise, accurate, and convenient summary of the structure of an OEM document (and therefore of an XML document as well). It describes every unique label path exactly once, and a DataGuide does not contain any label path that is not in the source document. The DataGuide itself is an OEM object, and this allows it to be accessed, stored, and updated using already established techniques for OEM documents. In addition, multiple DataGuides can exist for the same OEM source. A sample DataGuide for our OEM example (Figure 1.5) is shown in Figure 2.3. Referring to the original OEM object (Figure 1.5) and to the corresponding DataGuide, we notice that the path for *restauraunt* is encoded only once (although it appears twice in the original source).

A DataGuide can also serve as a path index [GW97]. The effectiveness of using a path index in traditional object-oriented systems has been evaluated, but their use and effectiveness for indexing XML documents on their own has not been addressed. The use of a DataGuide (serving as a path index) as a portion of an index structure has been proposed and is discussed in the next section.

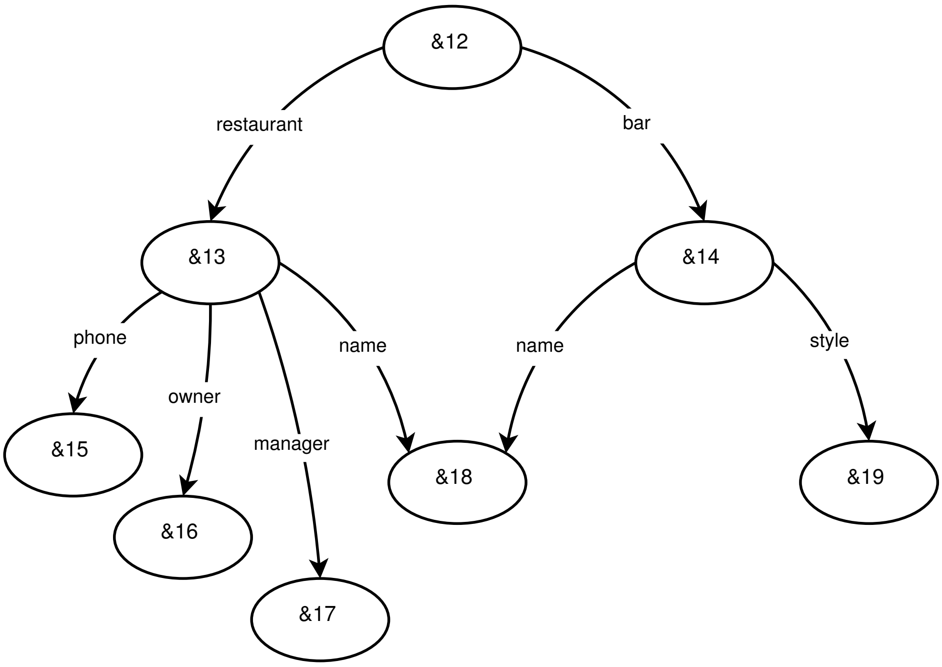
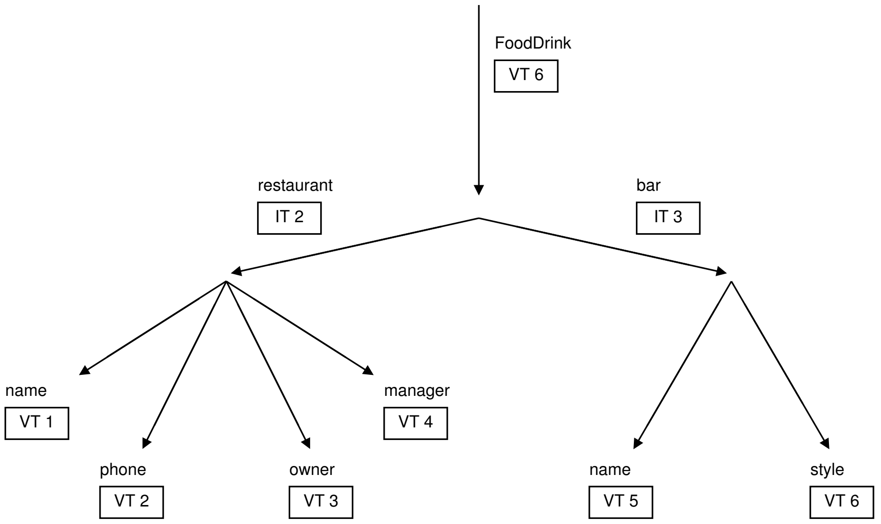


Figure 2.3: A Sample DataGuide

2.1.4 ToXin

Developed within the ToX (Toronto XML Engine) project at the University of Toronto, *ToXin* [RM01] seeks to exploit the overall path structure of an XML database in all stages of query processing. The index consists of two different structures: the Value Index and the Path Index. The latter index has two components, the index tree (a DataGuide) and a set of instance functions (one for each edge in the index tree). These functions are used to identify parent-child relationships between XML elements. The Value Index, as the name implies, stores XML nodes and values corresponding to those nodes. A sample ToXin tree and associated tables is shown in Figure 2.4. The node labels are taken from the OEM diagram in Chapter 1 (Figure 1.5). The IT boxes represent instance tables, and the VT boxes represent value tables.

One limitation and potentially costly issue with ToXin is the redundancy of information. In Figure 2.4, the information in VT1 and VT5 contain the same type of information (*name* values for establishments), yet they are broken into separate tables and therefore must be indexed independently. ToXin performs best on queries that yield large answer sets. While the effect is minimal for our example, the impact for query processing when using a larger XML database that splits



IT 1

Parent	Child
&1	&2
&1	&3
&1	&4

IT 2

Parent	Child
&2	&5
&2	&6
&2	&7
&3	&7
&3	&8
&3	&9

IT 3

Parent	Child
&4	&10
&4	&11

VT 1

Node	Value
&5	"Chili's"
&8	"Maggiano's"

VT 2

Node	Value
&6	"671-1102"

VT 3

Node	Value
&7	"G. Peppard"

VT 4

Node	Value
&9	"Crowley"

VT 5

Node	Value
&10	"Crowley"

VT 6

Node	Value
&11	"Irish"

Figure 2.4: Sample ToXin Tree and Tables

early and contains similar types of information farther down the tree has not been investigated.

2.2 TwigStack

Bruno et al. [BKS02] present the concept of a twig query (referred to by the authors as a holistic twig join) as an extension and improvement on previous index-only techniques such as ToXin. TwigStack is also closely related to ViST [WPFY03] which was developed parallel to TwigStack by different authors. TwigStack builds upon the ideas of PathSatck [BKS02]. PathStack was developed to process linear (non-twig) queries only. Therefore, they cannot answer queries that include branching. TwigStack utilizes a similar approach to query processing as PathStack but increases the number of stacks to allow for twig queries. The TwigStack technique has been shown to outperform other previous indexing methods such as Dataguide and ToXin [BKS02, JWLY03]. For that reason, we select it as a representative technique in this style of query processing. The TwigStack algorithm and its performance is investigated in more detail in later chapters.

2.3 Constraint Sequencing

Presented by Wang and Meng [WM05], Constraint Sequencing (referred to simply as sequencing) takes an entirely different approach to encoding (building) the index from an XML or OEM source document. With previous methods, the index was built sequentially, typically starting at the root node and inserting/adding to the index until all information was encoded. Some of these methods (such as ToXin and DataGuides) used multiple data structures to store the index. Sequencing operates by encoding the entire tree at once. Using a linked list of linked lists as the underlying data structure, an index is built that allows selection of an object or path by matching subsequences. The encoded information can be easily represented by adding prefixes (termed a forward prefix) to value nodes that encode their path along the tree. The labeling scheme utilized is similar to that used in extended preorder traversal [LM01], but it uses a depth-first traversal of the tree to assign the value $order(x)$ to each node x. Constraint sequencing is shown to outperform previous index approaches in most regards, but there is a problem when querying over a document that contains identical sibling nodes. When present, these nodes slow query performance by a factor of 10 (reducing times from 10-60ms to 100-600ms). Other indexes (such as TwigStack) do not suffer under the same conditions. The topic of Constraint Sequencing is investigated in more detail in later chapters.

Chapter 3

The TwigStack Method

Multiple techniques for analyzing XPath and XQuery expressions exist, but as was discussed in Section 2.2, many of these historical techniques are out-performed by TwigStack. This chapter provides a detailed discussion of the TwigStack approach by Bruno et al. [BKS02]. We also present an applicable example of the TwigStack algorithm and analyze the complexity of the TwigStack algorithm.

3.1 An Introductory Example

To better illustrate the advantages of the TwigStack approach, this section introduces a small example that will be used throughout the explanation. Figure 3.1 shows the tree representation of a sample XML document (a portion of a library database). A possible query over the data in Figure 3.1 may be represented in XPath as the expression:

```
/Library//book[date = ‘1983’ AND publisher = ‘KIT Press’]
```

This expression corresponds to the twig query shown in Figure 3.2. A twig query refers to a query with a structure that branches at some point. If a query is strictly linear and does not branch, it is not considered to be a twig query. Solutions to the query involve books that have a date of `1983` and a publisher of `KIT Press`. The only book node that satisfies these criteria is indicated by node 7 in Figure 3.1.

For the purposes of brevity, the root node *Library* will be ignored for the remainder of the example in this chapter. It does not participate in the bulk of twig query processing, and its

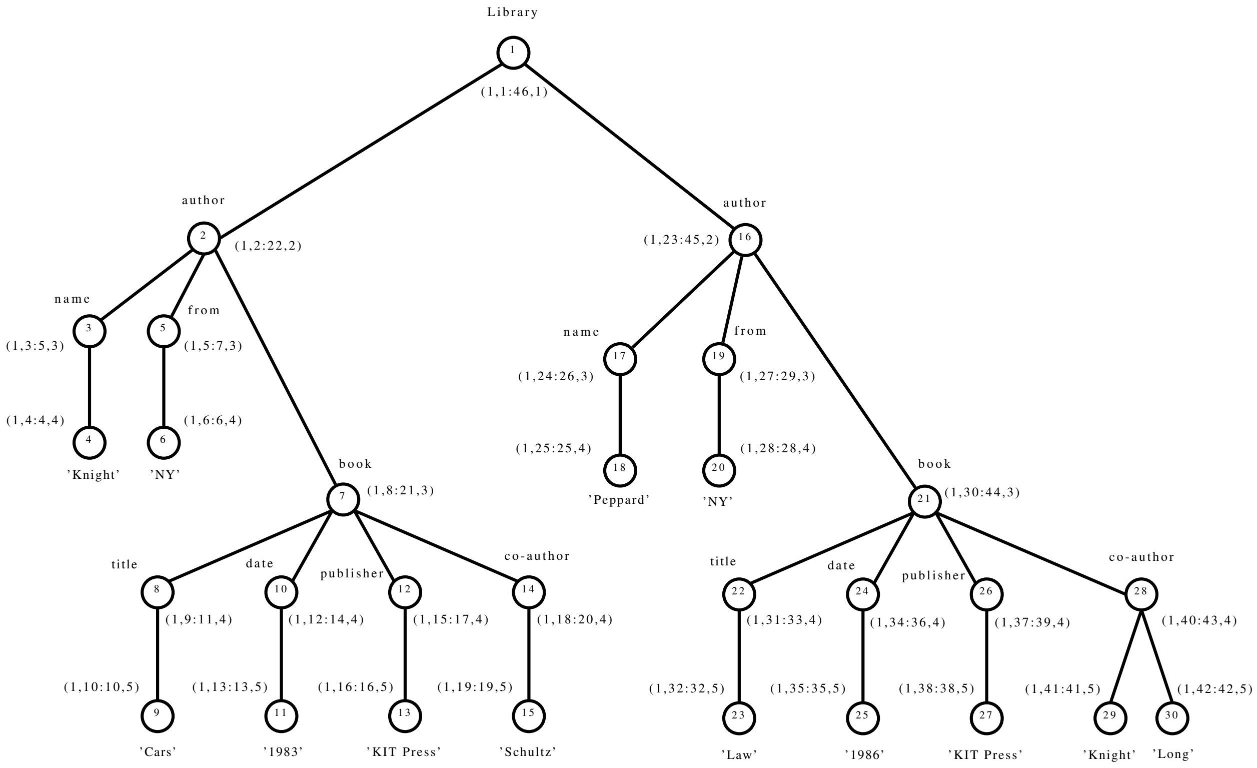


Figure 3.1: Sample XML Tree Representation

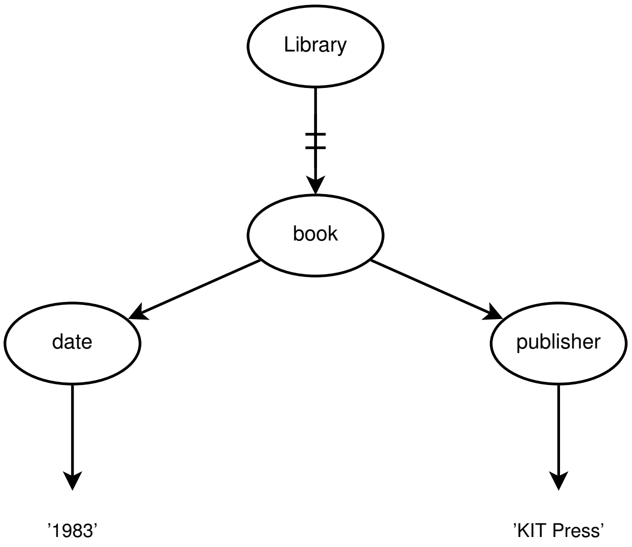


Figure 3.2: Sample XML Twig Query

elimination from the accompanying illustrations does not negate the validity of the examples.

3.2 Node Labeling

The TwigStack algorithm requires that the XML nodes be labeled. The labels represent a 3-tuple of `(DocID, LeftPos : RightPos, LevelNum)` [BKS02]. `DocID` refers to the document identifier and has been simplified for our example. The values for `LeftPos` and `RightPos` can be generated by simply counting word numbers from the beginning of the document to the start and end of the element. The `LevelNum` number represents the nesting depth of the element. It is important to note that for leaf-level nodes, the `RightPos` value and `LeftPos` value are the same. This numbering scheme is an extension of the preorder traversal method [LM01].

Information about the document's structure, including ancestor-descendent and parent-child relationships, is encoded in the node labels. If a node n_2 is encoded as $(D_2, L_2 : R_2, N_2)$ and is a descendant of node n_1 with encoding $(D_1, L_1 : R_1, N_1)$, then the following must hold.

1. $D_1 = D_2$; the `DocID` of both nodes must be the same.

2. $L_1 < L_2$; the `LeftPos` (start) of the ancestor must be less than the `LeftPos` of the descendant.

3. $R_1 > R_2$; the `RightPos` (end) of the ancestor must be greater than the `RightPos` of the descendant.

When encoding the more specific parent-child relationship, the additional condition $N_2 = N_1 + 1$ is imposed on the nodes. Referring to the example shown in Figure 3.1, the node `book` with position $(1, 8 : 21, 3)$ is a descendant of the `Library` node with position $(1, 1 : 46, 1)$. The node `book` with position $(1, 8 : 21, 3)$ is a child of `author` with position $(1, 2 : 22, 2)$. One advantage of using such a labeling technique is that checking for the general ancestor-descendent relationship is as simple as checking for the more exacting parent-child relationship. It also allows for checking order and structural proximity relationships [BKS02].

In Figure 3.1, we also give the nodes a unique label separate from the start/end positions. This label is shown in the node. For example, the node `book` with position $(1, 8 : 21, 3)$ can also be referenced using the single number 7 (shown inside the node). This is a convention we use for clarity throughout the remainder of this chapter. It is equally valid to represent nodes using just their start positions.

3.3 Stack Encoding

The TwigStack algorithm, as its name implies, uses a stack as its underlying data structure. A twig pattern (also known as a query twig pattern) is represented by q. Note that any twig pattern q can contain one or more sub-patterns, denoted by q'. The root of a twig pattern is denoted as q_{root}, but a shorthand notation is to refer to both the root of a twig pattern and the pattern itself by q[1]. By using the node labeling technique in Chapter 3.2, operations such as `children(`q`)`, which returns the set of nodes that are children of q, and `subtreeNodes(`q`)`, which returns q and all of its descendants, can be easily implemented. Similarly, the operation `parent(`q`)` returns the parent of q.

Associated with each twig pattern q is a stream T_q. This stream consists of the positional representation of nodes that match the node predicate at q [BKS02]. In other words, T_q contains all nodes, along with their descendants, that satisfy the twig pattern at node q. Nodes in T_q are sorted according to their `DocID` and `LeftPos` values. Two important stream operations are `nextL` and `nextR`, which return the `LeftPos` and `RightPos` of the next element in T_q.

Finally, each node q is associated with a stack S_q. Each item in the stack consists of the pair (`position(`T_q`)`, pointer to $S_{\texttt{parent}(q)}$). The function `position(`T_q represents the positional representation of a node from T_q. Traditional stack operations (`empty`, `pop`, and `push`) as well as the additional operations `topL` and `topR` are available. The last two operations return the `LeftPos` and `RightPos`, respectively, of the top element in the stack S_q. In any given twig pattern, there will exist one or more stacks. In the general case, a twig pattern containing k nodes q requires k stacks S_q.

3.4 Algorithm

As originally presented by Bruno et al. [BKS02], the TwigStack algorithm operates in two phases. The first phase of the algorithm discovers individual solutions for the various arms (sections) of the twig pattern. The second phase takes these individual solutions and merges them together to compute the final set of answers to the query twig pattern.

[1] The notation presented here is a clarification of the notation presented by Bruno et al. [BKS02]. This notation creates a unified terminology and set of definitions that are consistent with the TwigStack algorithm.

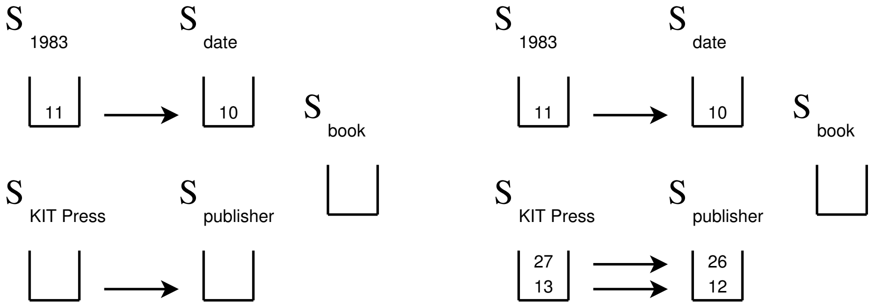
```
Input: a query, q
Output: a collection of stacks that contain nodes that satisfy the query

Algorithm TwigStack($q$)
  // Phase 1
  while ¬end($q$)
    $q_{act}$ ← getNext($q$)
    if (¬isRoot($q_{act}$))
      cleanStack($S_{parent(q_{act})}$), nextL($q_{act}$))
    if (isRoot($q_{act}$) ∨ ¬empty($S_{parent(q_{act})}$))
      cleanStack($q_{act}$, nextL($q_{act}$))
      moveStreamToStack($T_{q_{act}}$, $S_{q_{act}}$, top($S_{parent(q_{act})}$))
      if (isLeaf($q_{act}$))
        showSolutionsWithBlocking($S_{q_{act}}$,1)
        pop($S_{q_{act}}$)
    else advance($T_{q_{act}}$)

  // Phase 2
  mergeAllPathSolutions()

Function getNext($q$)
  if (isLeaf($q$)) return $q$
  for $q_i$ ∈ children($q$)
    $n_i$ ← getNext($q_i$)
    if ($n_i \neq q_i$) return $n_i$
  $n_{min}$ = minarg$_{n_i}$ nextL($T_{n_i}$)
  $n_{max}$ = maxarg$_{n_i}$ nextL($T_{n_i}$)
  while (nextR($T_q$) < nextL($T_{n_{max}}$))
    advance($T_q$)
  if (nextL($T_q$) < nextL($T_{n_{min}}$)) return $q$
  else return $n_{min}$

Procedure cleanStack($S$, $actL$)
  while (¬empty($S$) ∧ (topR($S$) < $actL$))
    pop($S$)
```

Figure 3.3: TwigStack Algorithm

3.4.1 Phase 1 - Individual Solutions

The most important part of the first phase is the getNext function. This function call guarantees that an individual solution can be merged with at least one other individual solution to produce an intermediate result that is not larger than the final answer to the twig query. In essence, getNext functions as a look-ahead routine. For every node h_q, getNext ensures that it has a descendent node h_{q_i} in each of the streams T_{q_i} for all $q_i \in$ children(q). Since getNext is called recursively, every node h_{q_i} also satisfies this property. In addition, a call to getNext ensures that a node h_{q_i} has a subtwig solution to the query but its parent, parent(h_{q_i}), does not have a subtwig solution. A subtwig (also known in the more general sense as a subtree) solution exists if the root-to-leaf path rooted at h_q forms a partial solution to the query q [GC07].

Assume that the TwigStack algorithm is called using the query shown in Figure 3.2 on the XML database shown in Figure 3.1. The first phase computes the individual root-to-leaf paths of the twig query. Figure 3.4 shows the progress of TwigStack during Phase 1. In Figure 3.4(a), the twig of the query shown in Figure 3.2 corresponding to the left-hand path is shown completed. The arrows between the individual stacks denote parent/child relationships between the elements in the stacks. Since there is only one node that matches to `1983`, there can be at most one node pushed onto both S_{1983} and S_{date}. Figure 3.4(b) shows the stacks after the right-hand path of the same query is executed. Note that there are two nodes that correspond to the value `KIT Press` and two nodes that satisfy to parent relationship for these nodes. For that reason, nodes 13 and 27 are pushed onto stack $S_{KITPress}$ and their respective parent nodes, 12 and 26, are pushed onto stack $S_{publisher}$.



(a) Date Twig Completed (b) Publisher Twig Completed

Figure 3.4: Stacks During TwigStack Execution

Figure 3.4 shows the result of the TwigStack algorithm before pushing the node of interest, *book*, onto the stack S_{book}. Recall that the purpose of `getNext` is to ensure that there exist a root-to-leaf path that satisfies the twig query. Before a *book* node can be pushed onto S_{book}, it must hold that it has a child node with a date equal to `1983` and a child node with a publisher equal to `KIT Press`. Referring back to Figure 3.1, the only candidates for pushed nodes are 7 and 21. Node 7 can be pushed onto S_{book} since it satisfies both child requirements. However, node 21 only satisfies one of the twig paths. Therefore, node 7 is the only book node pushed onto S_{book}, and this is shown in Figure 3.5.

Since the other nodes in stacks $S_{KITPress}$ and $S_{publisher}$ do not participate in a solution to the twig query, they are removed from the stacks via the function `cleanStack` shown in Figure 3.3 on line 8. The dashed line in Figure 3.5 shows the partial path that is cleaned from the stacks.

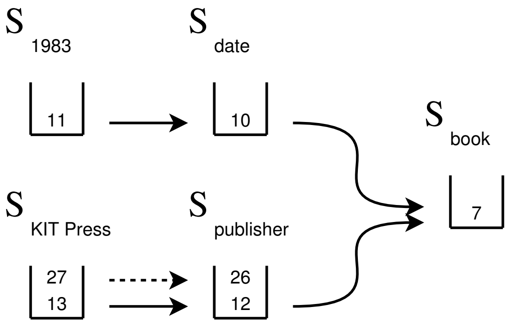


Figure 3.5: Stacks Before Cleaning

3.4.2 Phase 2 - Merge Individual Solutions

The second phase of the TwigStack algorithm requires all of the individual solutions in the first phase to be joined together. This can be accomplished in linear time with respect to the sum of the input (solutions to individual paths created in the first phase) and output (final answer) sizes [BKS02]. This phase is shown by the function `mergeAllPathSolutions()` in Figure 3.3. In essence, this function performs a union between all of the individual path solutions. Since this requires linear time with respect to the input and output, the effect on the overall complexity is minimal.

3.5 Algorithm Analysis

In order to build a concise yet fully applicable analysis of the complexity of TwigStack, it is necessary to make a few observations about its operation. Although the authors [BKS02] never state as such, the TwigStack algorithm can be viewed as a simple look-ahead function. Referring back to Figure 3.5, before a node is pushed onto Q_{book}, it must have a child in the stream T_{date}. This child must, in turn, have a child in the stream T_{1983}. This type of look-ahead continues until the leaf level of the query is reached, and it is completed for every twig in the query. The stream T_q contains all of the nodes that correspond to the condition q. For example, stream $T_{publisher}$ contains the nodes 12 and 26. They are ordered according to their left (start) positions, 15 and 37, respectively. Similar streams exist for all nodes in the XML document. Therefore, the complexity and scalability of the TwigStack algorithm must at least partially depend on the length of these streams. For the analysis, we are able to separate the two phases of TwigStack and focus on them separately.

The first part of the TwigStack analysis focuses around phase 1. The majority of this phase is dominated by the recursive calls to `getNext` shown on line 21 of Figure 3.3. We observe that on line 20, the function `getNext` is called inside a loop that iterates through all of the children of a given node. For this reason, the breadth of a query at a given point is important. If there are more children, then more iterations of the loop will be required and more recursive calls to `getNext` will be performed. For the purpose of analysis, we define a new parameter called local breadth and denote it by ψ. The term ψ_x refers to the breadth of the children of node x. To see a simple example, refer to Figure 3.1. The value of $\psi_{Library}$ is 2, since it has two children nodes. Similarly, the value of ψ_{book} is 4 for both *book* nodes.

Analyzing the TwigStack algorithm for a single iteration on node x while focusing on the first phase and the `getNext` function yields the equation

$$\sum_{i \in children(x)} |T_i|. \tag{3.1}$$

The summation is the result of the multiple executions of `getNext` and its included loop on line 25 of Figure 3.3. Since `getNext` will eventually return a single leaf node (line 19), expansion of the recursion results in a simple summation of the lengths of the streams T_i for all children of x. In the worst-case scenario, the algorithm would need to look through all of the nodes in all of the streams T_i to find that there are no nodes that match the required criterion (to be a child of a node in the partial solution). In other words, the worst-case is realized when there is no solution to a particular twig path of the query. The summation is over all the children of node x, but this is simply the local breadth around the node. We rewrite Equation 3.1 to take into account local breadth as

$$\sum_{\psi_x} |T_i|. \tag{3.2}$$

The other major component of phase 1 is stack cleaning (lines 6 and 8 of Figure 3.3). The same worst-case scenario applies applies here. The algorithm must remove all partial root-to-leaf paths that do not contribute to a final solution. Since a node is not pushed onto its corresponding stack unless a correct partial path has already been established, the worst that would occur is for every node in the stack $parent(x)$ to be popped. This is illustrated by the first condition of the loop on line 31. Taking this into account, we add that to Equation 3.2 to obtain

$$\sum_{\psi_x} |T_i| + |S_{parent(x)}|. \tag{3.3}$$

Equation 3.3 represents a single iteration of phase 1. As shown on lines 3 and 13, we must iterate over all nodes in the query q. This adds an additional summation to Equation 3.3 and results in

$$\sum_{x \in q} (\sum_{\psi_x} |T_i| + |S_{parent(x)}|). \tag{3.4}$$

Nothing can be pulled out of the summation in equation 3.4 since ψ_x represents the local breadth at each node x in q.

Phase 2 of the TwigStack algorithm requires all of the individual root-to-leaf paths be joined together to produce the final twig results. For our example shown in Figure 3.2 and the corresponding stacks shown in Figure 3.5, there is only one solution, node 7, to the query. If we are only interested in the *book* nodes that satisfy the query `/Library//book[date = '1983' AND publisher = 'KIT Press']`, then solution output is as simple as iterating through the length of the stack S_{book}. All that needs to be done is pop each node from S_{book}. This is a linear operation based on the size of the stack. Adding this to Equation 3.4 gives the final complexity for TwigStack as

$$\sum_{x \in q} (\sum_{\psi_x} |T_i| + |S_{parent(x)}|) + |S_{root}|. \tag{3.5}$$

The size of S_{root} is determined by the number of nodes pushed onto the root stack. For our example, the size of S_{book} is 1.

3.6 Summary

In this chapter, we discussed the TwigStack query technique as developed by Bruno et al. [BKS02]. In the original work, the authors present their algorithm and experimental results of its execution. We use that algorithm to build the cost model presented in Equation 3.5. That cost model is utilized in Chapter 7 when we perform a detailed analysis of the performance of TwigStack across its various parameters. In Chapter 4, we discuss another native XML query technique, Constraint Sequencing, and develop a cost model that illustrates its behavior.

Chapter 4

Constraint Sequencing

This chapter expands on the topic of the Constraint Sequencing of tree structures used for XML query processing. The work cited in this chapter is taken from Wang and Meng [WM05]. This chapter summarizes the work, presents an applicable example of the Constraint Sequencing algorithm, and formally analyzes the complexity of the Constraint Sequencing algorithm.

4.1 Overview

In processing XML twig queries, the most expensive operation is the join. In the relational database model, the join operation is used to connect two or more relations (tables) for the purpose of forming a new relation with the attributes of both relations. When using XML, the expensive join operation is referred to as a structural join. As the name implies, a structural join is used to connect or otherwise relate two or more sections of an XML tree. While other techniques seek to address efficient ways to index the XML tree to provide for efficient structural joins, Constraint Sequencing seeks to avoid these expensive join operations during query processing. Constraint Sequencing transforms the XML tree and associated queries into sequences and answers queries across the data via subsequencing matching.

4.2 Encoding the Tree

In order to facilitate the matching of subsequences, an XML tree must be encoded or otherwise labeled. The notation used by Wang [WM05] is a technology adopted from ViST [WPFY03]. In

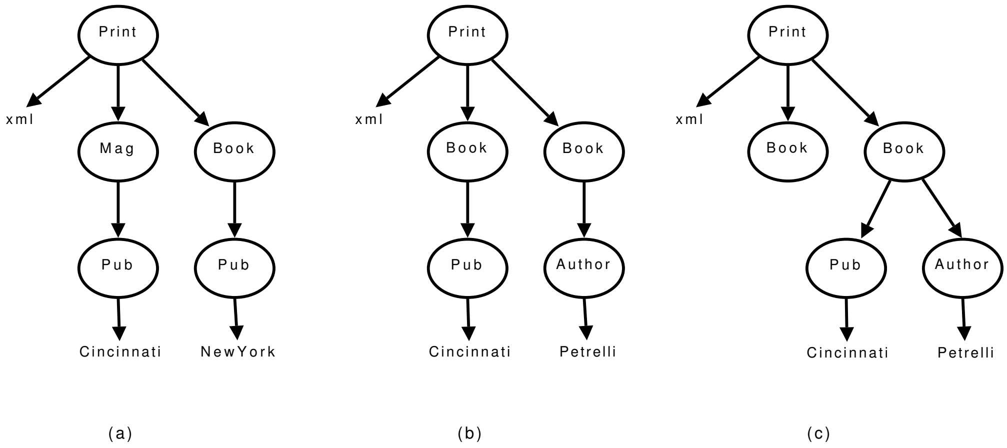


Figure 4.1: Tree Structure and Representation

this scheme, each element and attribute in an XML document is given a designator. For example, in Figure 4.1, the names `Print`, `Mag`, `Book`, `Author`, and `Pub` (abbreviated `R`, `M`, `B`, `A`, and `P`, respectively) are the designators. At the leaf level, the attributes are denoted by using a single designator derived from a hash function. For the example shown, the attribute values v_0, v_1, v_2, and v_3 represent `xml`, `Cincinnati`, `NewYork`, and `Petrelli`, respectively.

With the notation established, a simple path such as `/Print/Magazine[Published=Cincinnati]` is represented by <R,M,P,$\mathtt{v}_1$>. Using this notation, the goal of Constraint Sequencing is to include as much structural information as possible in the node encoding. This is done to make the actual sequencing easier and increase flexibility.

4.2.1 Sequencing

When there are no identical sibiling nodes in a tree, the sequencing (representation) of the tree is trivial. The tree in Figure 4.1(a) can be constructed directly from the encoded nodes:

$$\texttt{<R,Rv}_0\texttt{,RM,RB,RMP,RBP,RMPv}_1\texttt{,RMPv}_2\texttt{>.}$$

This is known as a path-based approach to node encoding. It is important to note that the order of the encoded nodes, for Figure 4.1(a), is irrelevant. The nodes could be reordered into any sequence and still produce the same figure. In other words, the full root-to-node path is encoded as part of each node and this is now referred to as the encoded node. However, if we consider Figure 4.1(b) and Figure 4.1(c), we observe that there are two identical sibling nodes B under node R. Therefore, the two trees, although structurally different, result in the same sequenced representation:

$$\texttt{<R,Rv}_0\texttt{,RB,RB,RBP,RBA,RBPv}_1\texttt{,RBAv}_3\texttt{>}.$$

This shows that, in the presence of identical sibling nodes, simple path-based encoding is insufficient. In these cases, constraints must be used to eliminate the ambiguity.

4.2.2 Root-to-Node Constraint

The idea of imposing a constraint on the sequence allows freedom to order the nodes arbitrarily while still producing a unique tree structure. In the case of Wang and Meng [WM05], the constraint is two-fold. The first part of the constraint enforces the inherent ancestor-descendent relationship in the tree. Assuming that a tree T consists of a sequence of path-encoded nodes $< p_1, \ldots, p_n >$, allow $f(p_i, p_j)$ to be `true` if node p_i is an ancestor of node p_j. The second part of the constraint states that $\forall p_j \in T$ and $\forall t \subset p_j$, there exists one and only one $p_i \in T$ such that $f(p_i, p_j)$ is `true` and $p_i = t$ [WM05]. In other words, for each path-encoded node in a tree, there is only one other path-encoded node that is its ancestor and contains the same path information (t) as the node in question. The authors refer to this as constraint f.

Assuming the tree has no identical sibling nodes, as shown in Figure 4.1(a), then the constraint $f_1(p_i, p_j) \equiv p_i \subset p_j$ holds [WM05]. It is important to note that constraint f_1 holds only if there are no identical sibling nodes. In this case, each path-encoded node p_i is unique. If identical sibling nodes exist, as in Figure 4.1(b) and (c), then a forward prefix is required to eliminate ambiguity.

4.2.3 Forward Prefix Constraint

In the presence of identical sibling nodes, a method of unambiguously determining the path of a given node is required. A valid sequence for Figure 4.1(b) is given as:

$$\texttt{<R,Rv}_0\texttt{,RB,RBP,RBPv}_1\texttt{,RB,RBA,RBAv}_3\texttt{>}$$

The first RB represents a portion of the path to $\texttt{RBPv}_1$, but the second RB does not. This introduces an additional constraint to f_1. Formally, if $T =< p_1, \ldots, p_n >$, then $p_k \subset p_i$ is a forward prefix of p_i if $\forall p_j = p_k, i < k < j$ and $\nexists p_j = p_k, k < j < i$ [WM05]. In other words, the constraint

Table 4.1: Constraint Sequences for Figure 4.1(b)

<R,Rv$_0$,RB,RB,RBP,RBPv$_1$,RBA,RBAv$_3$>
<R,RB,Rv$_0$,RB,RBP,RBPv$_1$,RBA,RBAv$_3$>
<R,RB,RBP,Rv$_0$,RBPv$_1$,RBA,RBAv$_3$,RB>
...

now relies on the order of the path-encoded nodes to determine ancestry relationships. This still allows arbitrary ordering of nodes in a sequence. Table 4.1 gives other valid sequences for Figure 4.1(b).

The technique used for generating such a sequence is controlled not only by a constraint f_2 but also by a user strategy g. One such user strategy defined by Wang and Meng is to first select the root node then repeatedly invoke a method to select a node whose parent node has already been selected [WM05]. If there are identical sibling nodes, a sibling of an already selected node x must not be selected until all descendants of x have been added to the sequence.

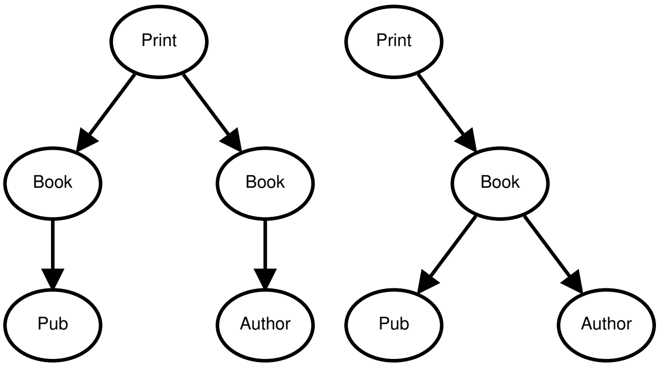
4.3 Querying a Sequence

After a sequence is generated, it can be successfully matched to another sequence using subsequence matching. For the purposes of this research, this subsequence matching is referred to as querying a sequence for a match (or simply querying). When performing a query, two problems unique to Constraint Sequencing, false alarms and false dismissals, must be handled properly.

4.3.1 False Alarms and False Dismissals

The ability to match a query (Q) to a document (D) via Constraint Sequencing depends on the constraint match. The correctness of the constraint match is diminished by false alarms and false dismissals. False alarms provide spurious results while false dismissals remove results that are valid [WM05].

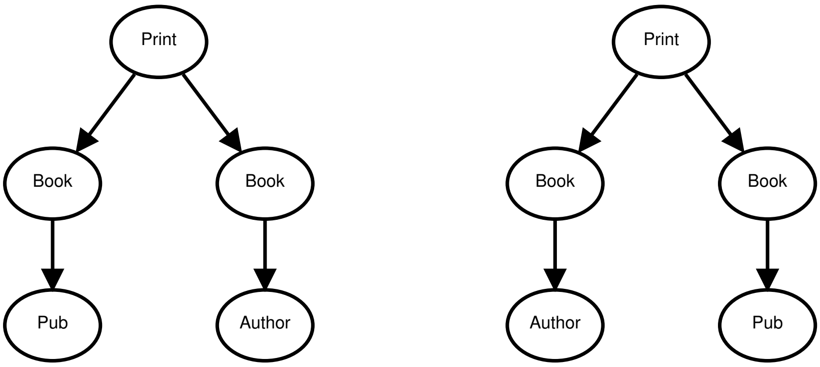
Figure 4.2(a) and Figure 4.2(b) represent different tree structures. A naïve subsequence would find a match between Q and D since $Q \subseteq D$. Without using the constraint match, the sequence `<R,RB,RBP,RBA>`, representing Q, is successfully located in D. This means that the query would be successful even though the document structure does not reflect equivalence. While this problem is not unique to Constraint Sequencing, other methods, such as ViST [WPFY03] and PRIX [RM04],



(a) D : <R,RB,RBP,RB,RBA> (b) Q : <R,RB,RBP,RBA>

Figure 4.2: False Alarm Triggered by Identical Sibling Nodes

solve this problem with expensive join operations. One of the underlying goals of Constraint Sequencing is to avoid such expensive operations.



(a) D_1 (b) D_2

Figure 4.3: False Dismissal Triggered by Tree Isomorphisms

In addition to false alarms, sequencing matching is subject to the problem of false dismissals. Unlike false alarms, false dismissals result from tree isomorphisms such as the mirror images shown in Figure 4.3. Based on the forward prefix technique discussed in Section 4.2.3, the same data (a document in this case) could be encoded as Figure 4.3(a) or Figure 4.3(b), documents D_1 and D_2, respectively. If a query is run on D_1 and the query is in a form written for D_2, the match will be dismissed.

While the false dismissal problem is trivial to solve, the false alarm problem requires more processing and the addition of a constraint. The following section discusses the constraint match and its improvements upon the naïve sequence match.

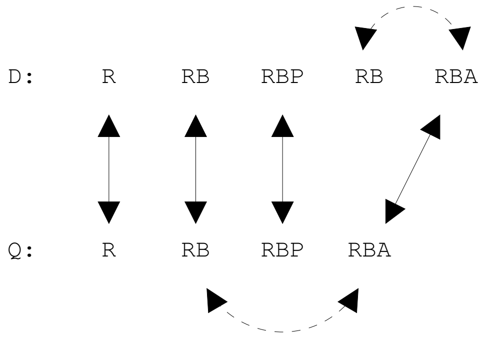


Figure 4.4: Sequence Match

4.3.2 Performing A Constraint Match

The naïve subsequence match is insufficient in the presence of identical sibling nodes. Specifically, their occurrence allows for false alarms and dismissals when running a query over a document. The false dismissal problem is trivial to solve. Assuming we have a query Q and a document D, simply allow Q to be run over D and all of its isomorphisms. The results from these queries are saved and then unioned together to produce the final result [WM05]. Unlike other techniques [RM04, WPFY03] that require join operations to solve the false alarm problem, sequencing can handle the problem directly if a constraint match is introduced [WM05].

Figure 4.4 shows a document D and query Q. This figure is analogous to the information presented in Figure 4.2. A match between D and Q is represented by a solid line. Wang and Meng [WM05] add a function $m(\cdot)$ that maps an item in Q to its matched item in D. Given a match $m(\cdot)$ between Q and D, which is based on constraint f, it is considered to be a constraint match is the following criteria are satisfied:

1. $m(a) = b \Rightarrow a = b$, and
2. $f(a, b) \Leftrightarrow f(m(a), m(b))$

Recall from Section 4.2.2 that constraint f embodies the ancestor-descendent relationship of two items (nodes or elements) in a document or query. The naïve subsequence match only guarantees that $m(a) = b$. This is shown by the solid lines in Figure 4.4. The second criterion, $f(a, b) \Leftrightarrow f(m(a), m(b))$ is violated in Figure 4.4, and this is shown by the dotted lines. In Q, node `RB` is an ancestor of node `RBA`. This can also be seen in Figure 4.2(b). However, in D, the matched nodes for

RB and RBA do not embody an ancestor-descendent relationship. In other words, $m(\texttt{RB})$ is not an ancestor of $m(\texttt{RBA})$. Since this violates the second criterion of a constraint match, the false alarm is avoided. If there are no identical sibling nodes in the document, then the second condition is implied by the first. Without identical sibling nodes, the relative positions of the items are uniquely defined by their respective paths.

4.4 Algorithm Analysis

In order to perform a constraint sequence match, an index must first be constructed. After the sequences are inserted, a widely adopted [BKS02, CVZT02, SLFW05] XML labeling scheme is implemented [LM01] that assigns each node n a pair of integers $(n^{\vdash}, n^{\dashv})$. The value of $n^{\vdash}$, also known as n's serial number, is derived from a depth-first traversal of the index tree (assigning 0 to the root node). The value of $n^{\dashv}$ is the largest serial number of n's descendants. For example, in Figure 3.1, the pair of integers that correspond to the first author node (shown on the left) is (2,22). The first number refers to the number of the node itself while the second number tells us that the highest numbered node that is a descendant of that author node has a serial number of 22. Given two nodes p and q, this allows us to determine if p is q's descendent if $p^{\vdash} \in (q^{\vdash}, q^{\dashv}]$.

After the nodes are numbered, horizontal path links are created for each unique path that appears in the sequences [WM05]. In the absence of identical sibling nodes, these links appear to be strictly horizontal throughout the index. This means that they do not traverse a single sequence (branch of the tree) down to any depth. In this case, node labels that belong to a single link are organized in ascending order of their serial number $n^{\vdash}$ [WM05]. Figure 4.5 illustrates what happens to a single link, RB, in the presence of identical sibling nodes. Note that the link begins to traverse the sequence in a depth-wise fashion and then continues to the next branch of the index (sequence). For multiple identical sibling nodes in a single sequence, the path link would continue down the same sequence to an undetermined depth.

Wang and Meng [WM05] note that these horizontal links can be implemented via an efficient structure that supports binary search. Thus, Constraint Sequencing is dominated by the process of finding a subsequence match and not by the creation of an index. The algorithm, shown in Figure 4.6, is adapted from the algorithm presented by Wang and Meng [WM05]. The analysis is divided into three main components, (1) binary search, (2) searching for identical sibling nodes, and (3)

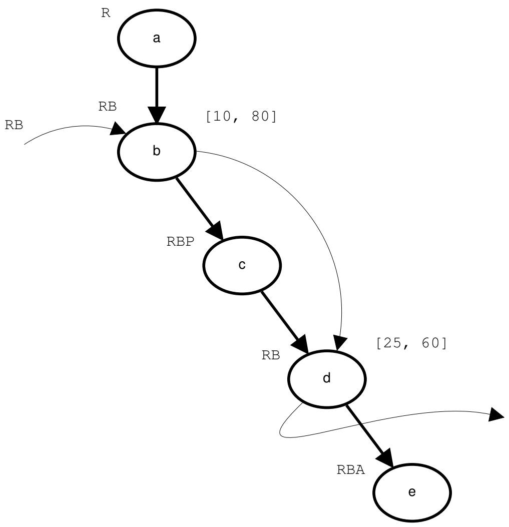


Figure 4.5: Path Links with Identical Sibling Nodes

recursively searching.

4.4.1 Search for Nodes in Range

A binary search is performed on line 14 of Figure 4.6. In order to find nodes in the range $[v_s, v_m]$, two binary searches are performed to find the upper- and lower-bounds of the range. Two binary searches are required since we must first search for v_s and then perform another search for v_m. This gives us the position of nodes v_s and v_m and we can then iterate through all nodes inclusive of this range. The nodes in the range are then iterated through sequentially. The time to search through the list I is given as

$$2(2\log_2|I|). \tag{4.1}$$

The term $|I|$ refers to the size of the horizontal list of node p_i. To keep this term consistent with other analyses, allow this to represent the branching factor of the XML index. Substituting this in Equation 4.1, we arrive at

$$4\log_2(b) \tag{4.2}$$

```
1  Input: a query $Q$
2  Output: docs $\in$ $D$ that contain query structure $Q$
3
4  Let $Q$ $\leftarrow$ $< p_1,\ \ldots,\ p_i,\ \ldots >$
5  Let $r$ $\leftarrow$ root node of index tree
6
7  search($r, 0, \{\}$)
8
9  Function search($v, i, isn$)
10   if $i$ $<$ $|Q|$
11     $i$ $\leftarrow$ $i$ $+$ $1$
12     $I$ $\leftarrow$ horizontal link of $p_i$
13
14     Perform binary search in $I$ to find nodes $\in$ $[v_s, v_m]$
15
16     for each $r$ $\in$ $I$ where $ID(r)$ $\in$ $[v_s, v_m]$
17       if $\nexists$ $x$ $\in$ $isn$ such that $x$ sibling-covers $r$ then
18         if $r$ embeds identical siblings
19           $isn$ $\leftarrow$ $isn$ $\cup$ $r$
20         search($r, i, isn$)
21   else
22     output $L[v_s\ \ldots\ v_m]$ document ID lists of node $v$
23     and all nodes under $v$
```

Figure 4.6: Subsequence Matching Algorithm

Disregarding constants, the complexity $log_2(b)$ is to be expected from a binary search. Once the binary search is complete, the list I must be searched for identical sibling nodes.

4.4.2 Search for Identical Sibling Nodes

The time required to perform a search over the list I of identical sibling nodes is tied to the number of items in the list. In the best possible case, there are no identical sibling nodes and the complexity is trivial. A more interesting result is found by analyzing the worst and average cases for this step. Much like the original search, a binary search is performed on the list that contains previously matched identical sibling nodes. Line 17 of Figure 4.6 is where the binary search is performed. If a node r is found that embeds (contains) identical sibling node, then it is added to the list isn. If the node is already in the list, it ignores it and proceeds to the next iteration. In the worst case scenario, the search will need to look at every item in the list isn, denoted as $|isn|$ in Equation 4.4.

$$|isn|(2(2\log_2|isn|)) \tag{4.3}$$

$$4|isn| \times \log_2(|isn|) \tag{4.4}$$

Substituting s for the size of the identical sibling node list, we arrive at Equation 4.5.

$$4s \times \log_2(s) \tag{4.5}$$

Equation 4.5 only represents a single iteration of the recursive search function. On line 20 of Figure 4.6, the search function is recursively called with the parameters r, i, and isn. Note that $i = |Q|$ is the base condition for the search algorithm, and the search function will be called a total of $|r|$ times. The term $|r|$ refers to the number of nodes $r \in I$ that satisfy the necessary condition $ID(r) \in [v_s, v_m]$. Taking into account the recursion of the search, Equation 4.2 and Equation 4.5 can be combined to give Equation 4.7.

$$|r| \times (4\log_2(b) + 4s\log_2(s)) \tag{4.6}$$

$$4|r| \times (\log_2(b) + s\log_2(s)) \tag{4.7}$$

Upon examining Figure 4.6 and Figure 4.5 and assuming that there may be multiple identical sibling nodes in any given sequence, the size of r could continue to expand. In the worst case, r approaches the depth of the tree (sequence). Given that fact, Equation 4.7 can be rewritten as

$$4m \times (\log_2(b) + s\log_2(s)) \tag{4.8}$$

where m is the depth or length of the sequence.

The total complexity for a subsequence match that takes into account the constraint f to sufficiently accommodate identical sibling nodes and produce correct results is given by Equation 4.9.

$$4|q| \times m \times (\log_2(b) + s\log_2(s)) \tag{4.9}$$

Equation 4.9 combines Equation 4.8 and the size of the possible sequence. In this case, the size of the sequence is represented by the size of the query, $|q|$. As mentioned earlier, the worst case scenario for this technique occurs when there are many identical sibling nodes. In this case, Equation 4.9 performs like a standard depth-first search and degrades to $O(b^m)$. This is a dramatic shift from the complexity of Equation 4.9 that is essentially logarithmic in nature. As the number of identical sibling nodes increases, performance of the sequencing algorithm degrades into a depth-first search.

$$4 \sum_{x \in q} (m_{seq_x} \times s_x \log_2(s_x) + m_{doc_x} \times \log_2(b_x)) \tag{4.10}$$

Equation 4.10 shows a more general form of Equation 4.9. If the worst case does not occur, then the sequencing algorithm need not execute across the entire document m. Equation 4.10 introduces parameters m_{seq} and m_{doc} which represent the length of the document required to answer the query q and the total length of the document respectively. In the case where $m_{seq} = m_{doc}$, then Equation 4.10 becomes Equation 4.9 and represents the worst case performance of Constraint Sequencing. The summation is introduced since various iterations of the algorithm may have different values for m_{cs}. The subscript x on both m_{seq} and m_{doc} denotes the specific values for the two terms for a given node x in query q.

4.5 Summary

In this chapter, we discussed the Constraint Sequencing method for querying XML documents as developed by Wang and Meng [WM05]. In the original work, the authors present their algorithm and experimental results of its execution. We use that algorithm to build the cost model presented in Equation 4.9 and the general case model in Equation 4.10. Those models are used in Chapter 7 when we perform a detailed analysis of Constraint Sequencing across all of its parameters. In Chapter 5, we switch our focus to non-native query techniques and discuss the SS-Join algorithm.

Chapter 5

Querying Ordered XML Data Using Relational Databases

In contrast to native techniques used for querying XML documents, several techniques leverage existing relational database management systems and their associated tools to store and query XML documents using the relational model. This chapter discusses one such solution by Shui et al. [SLFW05], presents an applicable example of the algorithm, and formally analyzes the complexity of the solution. It also discusses the limitations of that solution and provides motivation for a new alternative method (presented in Chapter 6) that builds upon the work by Shui et al. [SLFW05].

5.1 Overview

Using a relational database (RDB) to store and query XML documents presents a set of challenges. An XML document is inherently hierarchical, and using the relational model requires reducing hierarchy or otherwise flattening a document into tables. In addition, both of the prevailing XML query techniques, XPath and XQuery, require the XML document to be ordered. With the relational model, there exists no inherent order within the individual tables. While an order may be enforced at the time of query execution, this order does not persist inside of the database. Previous work by Tatarinov et al. [TVB$^+$02] introduces shredding and proposes multiple encoding schemes to accomplish the shredding task. Once the XML document has been translated to relational tables, the problem of executing structural joins surfaces. Research by Shui et al. [SLFW05] introduces two structural join algorithms that operate without the need of the RDB index. The next section introduces the concept of XML document shredding and various encoding schemes and discusses the necessary task of maintaining document order.

5.2 Storing XML Data in an RDB

In order to store an XML document in an RDB, the document must be flattened to some extent. The end result of such a translation is a table or multiple tables that not only maintain leaf-level information of the original document but also capture the hierarchical structure of the document. This structure must be preserved in order to execute a structural join. It is sufficient to visualize this table as a large list where each entry (sometimes referred to as a record or tuple) corresponds to a single node (internal or leaf) of the XML document. Each entry tuple must have information that refers back to its parent and to any children nodes. This requires the XML document to be encoded in a manner that allows this parent/child relationship to be extracted from the structure and given a numerical value. Research by Tatarinov et al. [TVB$^+$02] proposes three such encoding schemes: global, local, and Dewey order.

5.2.1 Encoding Schemes

The three encoding schemes proposed by Tatarinov et al. [TVB$^+$02] are global order, local order, and Dewey order. Each of the encoding schemes has their own methodology, strengths, and weaknesses.

Global order assigns each node a number that represents its absolute position in the document. For example, a depth-first or breadth-first traversal of the XML document results in a global encoding. This value could be easily computed using the byte offset between a node's opening tag and the beginning of the document. A potential problem with global order is the insertion of new nodes in the middle of the XML document. When this occurs, many nodes may need to be relabeled if they have a higher encoding number than the newly inserted node. In shredding, a node encoded using global order would appear as the tuple *Edge(id,parent_id,end,path,value)* in the *Edge* table. The *id* column is the numerical value of the node and also serves as the primary key while the *parent_id* column is the numerical value of the parent node. The *parent_id* column is also a foreign key that refers back to the primary key, *id*, of the *Edge* relation. The last descendent node is stored as *end*, and the path to the represented node is stored in the *path* column. This column holds the entire root-to-node path for the represented tuple/node, and this gives us the ability to answer parent/child XPath expressions with a simple selection operation. To save space, a separate *Path* relation can be created to store unique paths and an identifier. The *Path* table is typically small since it only stores unique root-to-node paths. In other words, multiple children of the same parent would reference the same path_id in the *Path* table. Since the algorithm presented by Shui et al. [SLFW05] does not use the RDB index, it also does not utilize the *Path* relation since this would necessitate index hits for the joins. The final column, *value*, contains the text value of text nodes (where appropriate). Performance studies show that global ordering results in the best query performance [TVB$^+$02]. Since our focus is on query optimization, this encoding scheme is favored over the others.

Local (sometimes called sibling) order assigns each node a number that represents its position relative to its siblings. For example, if a parent node is encoded as value 1, its three children nodes would be encoded as 1, 2, and 3. Similarly, the children of node 2 would be encoded as 1, 2, and so on. An advantage of local order over global order is the low overhead created by updates, including new insertions. Where global order has to renumber every node after a newly inserted node, local order only needs to update the following sibling nodes of a newly inserted node. However, experimental results show that local order has the worst query performance [TVB$^+$02] out of all three techniques. Since our focus is on query optimization and processing, we eliminate this encoding scheme from consideration.

The final encoding scheme, known as Dewey order, is a combination of the global and local ordering techniques and is based on the Dewey Decimal Classification system. Each node is assigned a numerical vector that represents its root-to-node path. Each component of the path represents the local order for that particular node. In terms of query performance, Dewey ordering is similar to that of global order. In addition, it requires fewer updates than global order when an insertion is performed. The main disadvantage of the Dewey ordering technique is that it does not allow fixed sizes for node IDs [SLFW05, TVB$^+$02]. The sorting algorithm proposed by Shui et al. [SLFW05] operates on the IDs (keys) of the nodes. Lacking a fixed size, the sorting algorithm will not have a constant time operator for node comparison.

5.2.2 Shredding Example

A sample XML tree is shown in Figure 3.1. The values inside the nodes represent a global encoding of the tree using a depth-first strategy. This document can be converted into a flattened relation using the shredding technique proposed by Tatarinov et al. [TVB$^+$02].

As stated in the previous section, a reduction of repeated values can be achieved by separating the *path* column into a separate relation. However, for the purposes of the algorithm and illustration, the full root-to-node path is left in the *Edge* table. In addition, we utilize the start/end (also known as start/stop) positions of the nodes shown in Figure 3.1 when we discuss shredding. The values inside the nodes (used for clarity in Chapter 3) can be ignored.

One of the complications with the shredding technique is maintaining the order of an XML document when a new node is inserted. Proper document order must be maintained in order to use the proposed structural join algorithm. The next section discusses this issue and presents an efficient solution to the problem as developed by Shui et al. [SLFW05].

5.2.3 Maintaining Document Order

Once the XML document has been shredded into a relation or multiple relations if using a separate *Path* table, the structural join algorithm discussed in Section 5.3 is utilized to answer XPath queries. However, if the XML document is altered by adding a new node, the document must be analyzed to determine if relabeling is required. Recall from Section 5.2.1 that when a global encoding scheme is used, an insertion at any point has the potential requirement that all nodes with an ID higher than the new node be relabeled. Two algorithms, XRU-Insert and XRU-Relabel, are presented by Shui [SLFW05] that accomplish relabeling in $O(\log_2(n))$, where n is the size of the database.

The algorithm XRU-Insert assumes there is always a gap of at least 1 between the encoding of the nodes adjacent to the newly inserted node. If this is the case, the new node can simply be inserted without the need to relabel any of the other nodes. If this is not the case, XRU-Insert calls XRU-Relabel which completes the task of reassigning node IDs. It accomplishes the relabeling by maintaining two relational cursors that point to the previous node and next node in document order. As the algorithm advances, the two cursors step outwards from the new node. This step mirrors a simple linear traversal. It terminates when the the previous cursor points to an ancestor node that has a density less than T^{-i} where i is the distance between the two cursors

id	*parent_id*	*end*	*path*	*value*
1	null	46	/	Library
2	1	22	/Library	author
3	2	5	/Library/author	name
4	3	4	/Library/author/name	Knight
5	2	7	/Library/author	from
6	5	6	/Library/author/from	NY
8	2	21	/Library/author	book
9	8	11	/Library/author/book	title
10	9	10	/Library/author/book/title	Cars
12	8	14	/Library/author/book	date
13	12	13	/Library/author/book/date	1983
15	8	17	/Library/author/book	publisher
16	15	16	/Library/author/book/publisher	KIT Press
18	8	20	/Library/author/book	co-author
19	18	19	/Library/author/book/co-author	Schultz
23	1	45	/Library	author
24	23	26	/Library/author	name
25	24	25	/Library/author/name	Peppard
27	23	29	/Library/author	from
28	27	28	/Library/author/from	NY
30	23	44	/Library/author	book
31	30	33	/Library/author/book	title
32	31	32	/Library/author/book/title	Law
34	30	36	/Library/author/book	date
35	34	35	/Library/author/book/date	1986
37	30	39	/Library/author/book	publisher
38	37	38	/Library/author/book/publisher	KIT Press
40	30	43	/Library/author/book	co-author
41	40	41	/Library/author/book/co-author	Knight
42	40	42	/Library/author/book/co-author	Long

Table 5.1: Shredding of Figure 3.1 into *Edge* Relation

and T is a threshold defined by the user. In other words, the cursors continue to move outwards from the new node in opposing directions through the *Edge* table. They terminate the move when reach a suitable distance apart (where the suitable parameter, T, is defined by the user), and then only nodes within that range need be relabeled. Previous research [BCD+02] suggests values for T that produce good practical results. The purpose of using the relational cursors is to avoid a potentially costly SQL statement[1]. Each time the statement is executed, a complete index scan of the *id* attribute is required [SLFW05]. The index-free method provided by the cursors allows for operation of XRU-Relabel in-memory without the need to scan an index. This concept of not using a relational index will be used again in the next section.

5.3 Structural Join for Relational Databases

The structural join algorithm developed by Shui et al. [SLFW05] is divided into four parts: SS Descendant Join, SS Ancestor Join, Skip Descendants, and Skip Ancestors. The authors intentionally divided the algorithm into two complementary parts representing the ancestor nodes and descendant nodes of a query result. Since only one part (either ancestor or descendant) is typically required for a given query, this allows faster query processing since only half the amount of work is being performed. In the cases where both components are required, it is sufficient to run both parts of the algorithm to achieve the necessary result. For the purposes of analysis, only two of the four components, SS Descendant Join and Skip Descendants, will be presented in their entirety. The other two components, SS Ancestor Join and Skip Ancestors, are similar to their descendant counterparts in complexity and execution. For simplicity, the entire algorithm is referred to as SS-Join.

5.3.1 The Structural Join Algorithms

The algorithms presented by Shui et al. [SLFW05] require the use of three XPath order-based axes: `preceding`, `following`, and `descendant` [xpa09]. The `preceding` axis contains all nodes, excluding ancestor nodes, that occur before a specified context node, and the `following` axis contains all nodes, excluding descendent nodes, that occur after a specified context node. The `descendant` axis contains all nodes that are descendants of a specified node. The `ancestor` axis is the complement

[1] The SQL query is given as: `SELECT count(id) from global where id between low and high`.

axis	*member nodes*
`preceding`	[3, 6]
`following`	[23, 42]
`ancestor`	[1, 2]
`descendant`	[9, 19]

Table 5.2: XPath Axes Examples Using Figure 3.1 and Node 8 as the Context Node

to the `descendant` axis with respect to a particular node. It contains all nodes that are ancestors of a specified node. These axes represent a concept in XPath and XML documents. Using Figure 3.1 as a reference, we present some examples of the various axes used. Selecting node 8 as the context node, Table 5.2 represents the various axes with respect to the context node. The notation for member nodes indicates a range, inclusive of the start and stop points, that belong to the given axis.

The SS Descendant Join algorithm is replicated in Figure 5.1. For the purposes of this algorithm, the nodes *aNode* and *dNode* can be interpreted as structures that have member elements. The member element *start* refers to the *id* column in the *Edge* table while the *end* element refers to the corresponding *end* column. The idea of the algorithm is to maintain a current ancestor node, *cNode* and progressively check descendant nodes against this ancestor node searching for matches. Lines 15 through 17 perform checks to determine if the current ancestor node (*cNode*) should be removed. It is removed (set to `null`) if it still contains the next *aNode* or the next *dNode*. If $aNode.start < dNode.start$ (line 18), then *aNode* is either an ancestor of *dNode* or appears before *dNode* in its `preceding` axis. If *cNode* is not set, then *aNode* becomes *cNode* on line 20. The function *skipAncestors* (line 23) uses index-free skipping through the *Edge* relation to skip past (dismiss) all ancestor nodes that are in the `preceding` axis of the current *dNode*. This function returns either `null` or an ancestor of *dNode*. Note that if $aPos \geq aSize$ (line 21), then the algorithm has reached the final ancestor node in the list of ancestor nodes, so the current ancestor node is set to `null`. This causes *skipAncestors* to return `null` as well. If *dNode* is a descendant of *cNode*, then *dNode* is appended to the output (result) list (lines 25 and 26). The function *skipDescendants* (line 30) is called if *dNode* is in the `preceding` axis of the current *aNode*. To be in this axis, the following conditions must hold - $aNode \neq$ null, $aNode.start \geq dNode.start$, and $dPos < dSize$. The function returns the next descendant node that is either in the `descendant`

```
aSize: size of ancestor nodes (list)
dSize: size of descendant nodes (list)
aPos: current position of aNode (ancestor node)
dPos: current position of dNode (descendant node)

Input: two nodes, aNode and dNode
Output: list that contains dNodes that satisfy the query, output

dPos $\leftarrow$ 0, aPos $\leftarrow$ 0, cNode $\leftarrow$ null

if(aNode = null or dNode = null) then
  return

while((dPos $\leq$ dSize) and (aPos $\leq$ aSize)) do
  if(cNode $\neq$ null and aNode.start > cNode.start and
     dNode.start > aNode.end) then
       cNode $\leftarrow$ null
  else if (aNode $\neq$ null and aNode.start < dNode.start) then
    if(cNode = null) then
      cNode $\leftarrow$ aNode
      if(aPos $\geq$ aSize) then
        aNode $\leftarrow$ null
      aNode $\leftarrow$ skipAncestors(aNode, dNode)
  else
    if(cNode $\neq$ null) then
      append(dNode, output)
      if((dNode $\leftarrow$ fetchNext(cursorD)) = null) then
        break
    else if (dPos < dSize and aNode $\neq$ null) then
      dNode $\leftarrow$ skipDescendants(aNode, dNode)
    else
      if(aNode.start > dNode.start) then
        break
      else
        if(aNode.contains(dNode)) then
          append(dNode, output)
          dNode $\leftarrow$ fetchNext(cursorD)
          if(dNode = null) then
            break
end while
```

Figure 5.1: SS Descendant Join Algorithm

```
1   Input: two nodes, aNode and dNode
2   Output: next descendant node that is either a descendant of aNode
3           or in the following axis of aNode
4
5   if(dNode.start ≥ aNode.start) then
6     if(dPos ≥ dSize) then
7       dPos ← dPos + 1
8     return dNode
9
10  if((dNode ← fetchNext(cursorD)) = null) then
11    return null
12  else if(dNode.start ≥ aNode.start) then
13    return dNode
14
15  r ← ⌈log(dSize - dPos)/log(2)⌉
16
17  for(i ← 0, g ← 1; i < r and dPos < dSize;
18      i ← i + 1) do
19        g ← g × 2
20        if(g + dPos > dSize) then
21          g ← dSize - dPos
22
23        moveCursor(cursorD, FORWARD, g - 1)
24        dNode ← fetchNext(cursorD)
25
26        if(dNode.start ≥ aNode.start) then
27          break
28        else
29          dPos ← dPos + (g - 1)
30  end for
31
32  return binarySearchDescendant(dPos, dPos + g, aNode)
```

Figure 5.2: Skip Descendants Algorithm

axis of in the `following` axis of $aNode$.

5.3.2 Index-Free Skipping

Before the algorithm analysis is presented, it is necessary to briefly discuss the *skipDescendants* and *skipAncestors* functions. The algorithm for *skipDescendants* is shown in Figure 5.2, and both algorithms were developed by Shui et al. [SLFW05]. The *skipDescendants* and *skipAncestors* algorithms, collectively known as the skipping algorithms, are similar in their purpose and scope, and the discussion and analysis of the *skipDescendants* algorithm subsumes the analysis of the *skipAncestors* algorithm.

The overall goal of the skipping algorithms is to eliminate the need for index lookups during execution. This serves to avoid the overhead of maintaining external indices. Another benefit of the skipping algorithms is the ability to perform fast structural join processing without any a priori knowledge of the node distribution [HBG$^+$03, SLFW05]. Line 23 of Figure 5.2 illustrates that the *skipDescendants* function uses the relational database cursor to skip through tuples. The cursor is moved forward by an exponentially increasing amount (1, 2, 4, 8, and so on). If the matching nodes are close to each other in the *Edge* relation, the next match is achieved within a few skips. The exponentially increasing skip factor allows a large amount of potentially unmatched nodes to be discounted quickly and in few iterations. If the target node (node that satisfies the next match) is skipped, then a binary search is performed *min* and *max* to find the target node. The value *min* represents the last cursor position before the over-skip while *max* is the cursor position after the over-skip occurred [SLFW05]. The search space for the binary search (called on line 32 of Figure 5.2) is $2^k - 2^{k-1}$, where k represents the iteration number of the skip as an exponent of 2. In short, the *skipDescendants* function serves to bypass all potential descendant nodes that are in the `preceding` axis of a given ancestor node and skip to descendent nodes in the `following` axis or `descendant` axis. This allows us to discount any node that cannot satisfy the XPath query and prevents it from being added to the output list. The *skipAncestors* function features a similar methodology and approach, and it also uses the relational cursor to achieve index-free skipping. It bypasses all potential ancestor nodes that are in the `preceding` axis of a given descendant node.

The next section presents a complexity analysis of the SS-Join algorithm. This algorithm includes the SS Descendant Join, SS Ancestor Join, Skip Descendants, and Skip Ancestors functions.

5.4 SS-Join Algorithm Analysis

The algorithm analysis begins by examining the Skip Descendants (Figure 5.2) portion, referred to as *skipDescendants*. Apart from a few trivial checks (lines 5 through 13), the majority of the function is contained in a single for loop (lines 17 through 30). It is important to note that the functions $moveCursor$ (line 14) and $fetchNext$ (lines 5 and 15) are a linear move through the relation and occur without the use of an index. Since they occur in constant time, their effects on the overall performance can be dismissed. The for loop is bounded by the value $|r|$. The time to execute this loop is given as

$$|\lceil \log(dSize - dPos)/\log(2) \rceil|. \tag{5.1}$$

Using logarithm rules, Equation 5.1 reduces to

$$\log_2(dSize - dPos), \tag{5.2}$$

and this value is represented as $|r|$. The time required for the binary search (line 32) is on the order $O(\log_2(g))$, where g is the current skipping increment. Line 11 shows that g increases by a factor of 2. Since the search space is between $dPos$ and $dPos + g$, the maximum size of g is $2^k - 2^{k-1}$. There is no guarantee that this binary search will not be required for each iteration, so the worst case of *skipDescendants* becomes

$$|r| \times \log_2(g) \tag{5.3}$$

$$|r_d| \times \log_2(g) \tag{5.4}$$

The subscript in Equation 5.4 denotes this value of $|r|$ as that coming from the *skipDescendant* function. Through a similar analysis, the complexity of the *skipAncestors* function evaluates to a nearly identical expression represented by Equation 5.5. In this case, $|r_a| = |\lceil \log(aSize - aPos)/\log(2) \rceil|$.

$$|r_a| \times \log_2(g) \tag{5.5}$$

Since both the SS Descendant Join and SS Ancestor Join algorithms use both *skipDescendants* and *skipAncestors*, their representations can be combined as

$$|r_d| \times \log_2(g) + |r_a| \times \log_2(g) \tag{5.6}$$

By substituting in the search space value for g and performing some factoring, Equation 5.6 becomes the following

$$\log_2(g) \times (|r_d| + |r_a|) \tag{5.7}$$

$$(\log_2(g)) \times (\log_2(dSize - dPos)) + \log_2(aSize - aPos) \tag{5.8}$$

$$(\log_2(g)/\log_2^2(2)) \times (\log_2(dSize - dPos) + \log_2(aSize - aPos)) \tag{5.9}$$

$$\log_2(g) \times (\log_2(dSize - dPos) + \log_2(aSize - aPos) \tag{5.10}$$

Upon closer inspection, the terms $dPos$ and $aPos$ simply represent the position of the descendant and ancestor cursors, respectively, within the *Edge* relation. At the first iteration of *skipDescendants* and *skipAncestors*, it is possible that both $dPos$ and $aPos$ are relatively small. By minimizing their impact, Equation 5.10 reduces to

$$\log_2(g) \times (\log_2(dSize) + \log_2(aSize)). \tag{5.11}$$

Finally, substituting the maximum value of g into Equation 5.11 yields the final result of

$$\log_2(2^k - 2^{k-1}) \times (\log_2(dSize) + \log_2(aSize)). \tag{5.12}$$

The analysis of the SS Descendant Join (Figure 5.1) and SS Ancestor Join is simpler than the skipping functions. Using Figure 5.1 as a reference, note that the majority of the algorithm (lines 4 through 29) are contained in a single *while* loop. This loop is bounded by the sizes of the descendant and ancestor nodes, represented as $|dSize| \times |aSize|$. A similar observation of SS Ancestor Join shows that this is bounded by the same value, thereby multiplying our final result by a constant factor of 2. Combining this observation with Equation 5.12, the final complexity is given as

$$2 \times |dSize| \times |aSize| \times (\log_2(2^k - 2^{k-1}) \times (\log_2(dSize) + \log_2(aSize))) \tag{5.13}$$

$$2|dSize||aSize|(\log_2(2^k - 2^{k-1}) \times (\log_2(dSize) + \log_2(aSize))) \tag{5.14}$$

In order to make sense of Equation 5.14 and compare it with native XML query techniques, it needs to be put into the same context and framework as the original XML document. The depth of the tree is reflected by the values $dSize$ and $aSize$. A deeper tree has a higher frequency of descendent nodes, and this inflates both of the size values. To understand where the breadth of the tree factors into the analysis, we must revisit the process of shredding discussed in Section 5.2.1. Since the *Edge* relation holds the nodes (as individual tuples) in document order, the distance between sibling nodes reflects part of the tree's breadth. Referring back to Table 5.1, note that node 7 and node 21 each appear four times in the $parent_{id}$ column. This suggests that these nodes have multiple descendants that are siblings with each other. In fact, an observation of the original

XML document shown in Figure 3.1 confirms that the tree has a larger local breadth around nodes 7 and 21 than elsewhere in the tree. These sibling nodes represent potential answers to XPath queries, and the skipping functions will skip to the sibling nodes in an attempt to produce a correct result. Therefore, the breadth of the original XML tree is reflected by the amount of skipping that is required, and this impacts the value g. By substituting in the maximum search space $2^k - 2^{k-1}$ for g, we observe that the breadth is reflected by the value k in Equation 5.14.

While Equation 5.14 represents a worst case execution of SS-Join. Using Equation 5.10, we can slightly modify this worst case scenario to allow for a more general case. Equation 5.15 allows for the case where the ancestor and descendant node positions are not zero.

$$2|dSize||aSize| \times (\log_2(2^k - 2^{k-1}) \times (\log_2(dSize - dPos) + \log_2(aSize - aPos))) \qquad (5.15)$$

This more general form of the SS-Join algorithm will be used in subsequent chapters that compare the performance of SS-Join to other query algorithms. Much like with Constraint Sequencing, the worst case scenario, Equation 5.14, is sufficient to observe the effects of the various parameters of SS-Join and how they interact with other SS-Join parameters.

5.5 Limitations of SS-Join

While SS-Join presents an alternative to native XML query techniques, it suffers from several limitations that preclude a complete comparison with native techniques such as TwigStack and Constraint Sequencing. One such limitation is the inability to process queries greater than length two. SS-Join must operate on two nodes, $aNode$ and $dNode$, and it returns a list of nodes that satisfy the ancestor/descendant query. In order to process a query greater than length two, multiple executions of the SS-Join algorithm must be utilized. The main SS-Join algorithm is separated into two sections, SS Descendant Join and SS Ancestor Join, but these two components must be used in conjunction to return results similar to TwigStack and Constraint Sequencing. SS-Join is also unable to answer queries on documents that contain nested recursive nodes (an issue similar to the identical sibling node problem in Constraint Sequencing), and it treats ancestor/descendant queries the same as parent/child queries. All of these limitations are noted by Shui et al. [SLFW05]. These limitations serve as a catalyst for our own algorithm, RDBQuery, that seeks to improve on the SS-Join algorithm and address its limitations. Chapter 6 presents the RDBQuery algorithm along with an analysis of its performance.

Chapter 6

A New XML Query Technique, RDBQuery

The work presented by Shui et al. [SLFW05] is different from the techniques presented in Chapters 3 and 4 because it is non-native. It also does not utilize the relational cursor and therefore does not rely on the RDBMS to perform selection operations. We propose an alternative technique named RDBQuery that makes full use of available RDBMS commercial systems to perform efficient queries across XML data stored in a relational database. Unlike the other techniques discussed in Chapters 3, 4, and 5, our technique has the ability to not operate solely in memory and utilizes the RDBMS to perform selections. For that reason, the cost analysis needs to consider additional parameters such as blocking factor, selectivity, and selection cardinality. In this section, we present the algorithm RDBQuery, explain its improvements over the existing technique, and create a cost model for its execution.

6.1 Overview

Using the XML document shown in Figure 3.1, we proceed to shred the XML document into the *Edge* relation shown in Table 5.1. By using the global encoding scheme discussed in Section 5.2.1, we observe that the ancestor/descendant relationship can be determined by simply looking for appropriate descendant nodes that fall within the range `(id,end)`. In other words, if a node's *id* value in the *Edge* table falls in the range between ancestor's *id* and *end* values, then that node

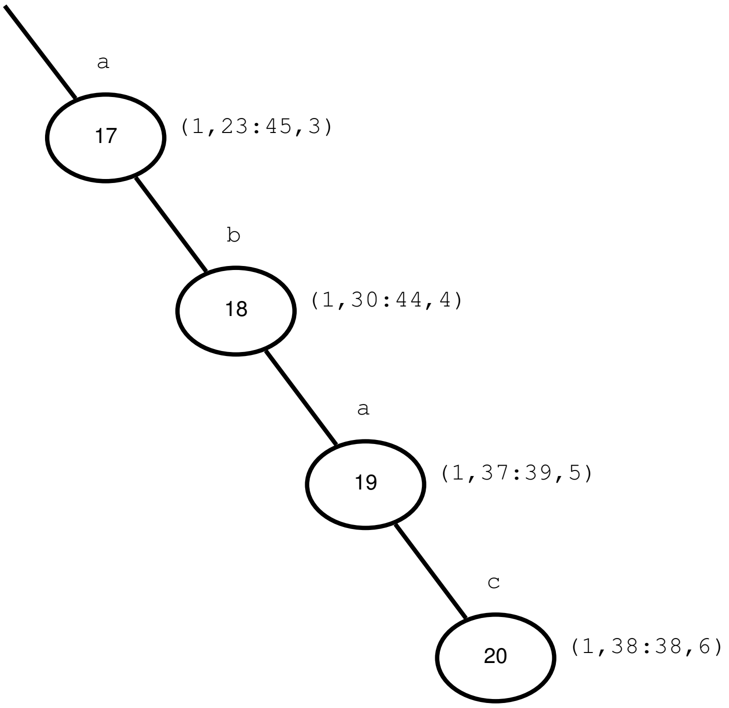
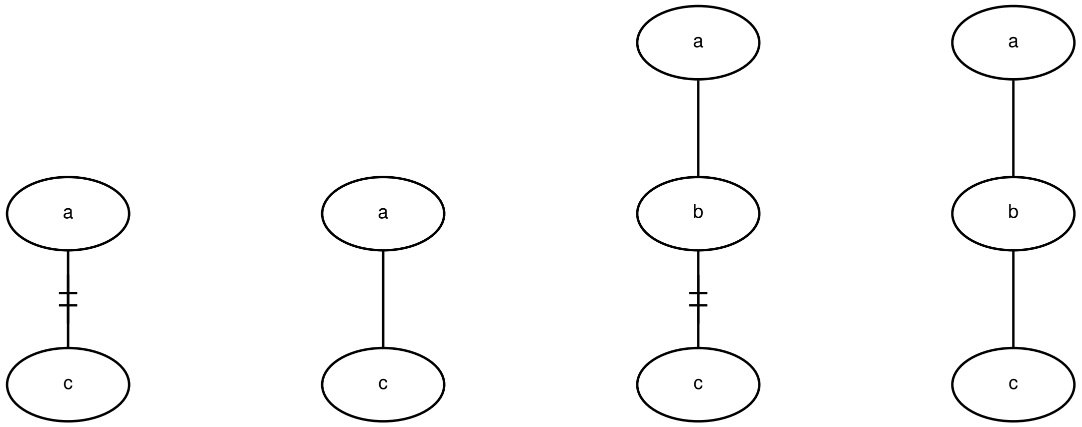


Figure 6.1: XML Document with Recursive Nodes

must be a descendant of that same ancestor. For a parent/child relationship, the type of query is simpler. In order to find a node *a* that is a child of node *b*, we simply need to run a selection query across the *Edge* table that selects tuples having a *parent_id* equal to the *id* of node *b*, the parent node.

Similar to the work presented by Shui et al. [SLFW05], RDBQuery operates in the presence of ancestor/descendant and parent/child edges. In contrast to their work, our RDBQuery also performs with the presence of recursive nodes in the original XML document. Figure 6.1 shows a portion of an XML document with recursive nodes. The two nodes labeled *a* represent different and unique items; one *a* node is a descendant of a separate *a* node, and there is another node *b* in between the two hierarchically. In the algorithm presented by Shui et al. and discussed in Section 5.3.1, all four queries shown in Figure 6.2 would be positively matched to the XML document shown in Figure 6.1. While three of the queries exist in the document, Figure 6.2(d) should not return a result when queried against Figure 6.1. Our RDBQuery algorithm solves this problem and will correctly ignore false results created by recursive nodes.

In addition, RDBQuery can execute queries that SS-Join cannot. A severe limitation of SS-



(a) Query 1 (b) Query 2 (c) Query 3 (d) Query 4

Figure 6.2: Query Styles Useful for RDBQuery

Join is that is can only process ancestor/descendant queries of length two (one ancestor and one descendant) during one execution of the algorithm. This prevents SS-Join from executing twig queries and queries longer than two nodes in length with a single pass. RDBQuery overcomes these limitations by leveraging the RDBMS to perform two specific selection queries that allow it to answer a query of arbitrary length and of an arbitrary shape.

6.2 RDBQuery Algorithm

The RDBQuery algorithm is shown in Algorithm 6.1. The algorithm works by accepting a query q and recursively checking all descendants and children in the query against the XML document to return a set of nodes that satisfy the query. The query in Figure 3.2 shows a twig query. For our example, we allow the *book* node to be the node of interest. In other words, we would like to execute the query shown over the XML document in Figure 3.1 and return all *book* nodes that satisfy the query. Like the algorithm presented by Shui et al. [SLFW05], our algorithm returns node 8 and does not include node 30 in the result set.

RDBQuery is divided into two main sections, a descendant case and a child case. The algorithm begins by examining the root node of a query q. It uses the main loop on line 2 to iterate through all edges that leave that node. In the case of an XML tree, all of the edges lead to a unique child node or specify a descendant relationship. For each edge that leaves the node q, we determine if it is a descendant edge or a child edge. This information, the classification of an edge as a child or

Algorithm 6.1: RDBQuery

```
Input: A query, $q$
Output: A list of nodes that satisfy the query, output
$q_c \leftarrow$ root($q_c$);
forall edge $\in$ leavingEdges($q_c$) do
    if edge = desc then                                  /* Next node is on descendant edge */
        $q_n \leftarrow$ nextNode($q_c$);
        $results \leftarrow \sigma_{value=q_n.value \wedge id>q_c.id \wedge end<q_c.end}$(edge);
        if $results = null$ then
            abort;
        else if $q_n$ is node of interest then
            RDBQuery($q_n$);
            append($q_n$, output);
        else
            RDBQuery($q_n$);
        end
    end
    if edge = child then                                 /* Next node is on child edge */
        $q_n \leftarrow$ nextNode($q_c$);
        $results \leftarrow \sigma_{value=q_n.value \wedge parent\_id=q_c.id}$(edge);
        if $results = null$ then  abort;
        if $q_n$ is node of interest then
            RDBQuery($q_n$);
            append($q_n$, output);
        end
        if $q_n$ is a leaf node then
            return;
        else
            RDBQuery($q_n$);
        end
    end
end
```

descendant edge, would take the form of metadata stored in conjunction with the query. If the edge specifies a descendant relationship, the algorithm executes lines 3 through 13. Lines 15 through 27 are executed if the edge represents a child node. In the case of a descendant edge, the algorithm looks ahead one node in the query and assigns it to q_n. It then performs a selection operation on line 5. The goal of this is to determine if there are any nodes in the RDB that can possibly satisfy this ancestor-descendant relationship. If there are no results, the algorithm does not need to continue; there cannot be any correct answers to the query. If the selection produces a result, then there must be at least one tuple that represents a successful ancestor-descendant relationship in the database. Using Figure 3.2 as an example, the RDBQuery algorithm would first look at the *Library* node. Inspecting the only edge leaving the node reveals that it encapsulates a descendant relationship. The next node, *book*, is assigned to q_n, and the algorithm then performs the selection on line 5. This selection operation includes three distinct checks:

1. Ensure that there is a *value* in the *Edge* table that matches the value of the node q_n,

2. Ensure that the starting point, *id*, of possible nodes is greater than the *id* of the ancestor node, q_c, and

3. Ensure that the ending point, *end*, of possible nodes is less than the *end* of the ancestor node, q_c.

Continuing our example, this means that line 5 will return all *book* tuples that fall within the range [1,46], inclusive. This results in two *book* nodes being returned, but only one, the node corresponding to node 8, is of interest. In order to determine this, the algorithm needs to continue its look-ahead feature and calls itself on line 10. This results in the next node, either *date* or *publisher*, being inspected. We assume that *date* is selected, and the new value of q_n becomes *date* on line 16. The selection operation on line 17 confirms that a tuple exists that satisfies the selection condition. In this case, since *book* and *date* have a parent-child relationship, the condition requires that there 1) exists a node with the value *date* and 2) that the node must have a parent *book*. Again, the algorithm returns two tuples that satisfy this condition, another level of recursion occurs on line 25. This time, the value of q_n is 1983, and we note that we have arrived at a leaf node of the query. The selection on line 17 confirms that there exists at least one tuple that matches the selection condition. For this case, we look for a node with value 1983 that has a parent matching a *date* node. We find only one node (node 11 in Figure 3.1) that satisfies this condition. Since this is a leaf node, line 22 is executed and the recursion halts. Note that this process would also need to be repeated for the right-hand side of the query in Figure 3.2. The execution and results are similar to those discussed for the left-hand side.

The results of the algorithm are contained in *output*. A node of interest can be specified within the query, and when this node is found and verified to have the appropriate children or descendants, it is appended to *output*. If either selection returns no tuples, then the algorithm can abort. The entirety of the query must be satisfied or else the document contains no results. In the next section, we present an analysis of the complexity of our RDBQuery algorithm.

6.3 RDBQuery Algorithm Analysis

Unlike native techniques or the technique presented in Section 5.1, our RDBQuery algorithm utilizes the built-in query optimizer of a relational database management system. Also unlike the other techniques mentioned, RDBQuery does not require the entire document to be preloaded into memory. For those reasons, it is necessary to consider disk access when performing a complexity analysis on RDBQuery.

Using cost models provided Elmasri and Navathe [EN00], we first build equations that model the selection queries on lines 5 and 15, σ_d and σ_c respectively, of Algorithm 6.1. The selection condition of σ_d contains three parts:

1. an equality operation on a nonkey attribute ($value = q_n.value$),
2. a comparison on a field with a primary index ($id > q_c.id$), and
3. a comparison on a nonkey attribute ($end < q_c.end$).

The first operation requires a cost of $x + \lceil (s/bfr) \rceil$, where x is the number of levels in the index, s is the selection cardinality, and *bfr* is the blocking factor. The second comparison incurs a cost of $x + (b/2)$, where b represents the number of index blocks needed. The final comparison is the most expensive of the three and costs $x + (b_{I1}/2) + (r/2)$. In this equation, b_{I1} represents the number of first-level index blocks required while the $r/2$ parameter is related to selectivity. The value $r/2$ assumes half of the records are returned by the selection query. If we have a more accurate estimate of the number of records returned, then this parameter decreases by increasing the denominator. Adding these three costs together and combining terms, we arrive at a cost for σ_d:

$$x + \left\lceil \left(\frac{s}{bfr} \right) \right\rceil + x + \frac{b}{2} + x + \frac{b_{I1}}{2} + \frac{r}{2} \tag{6.1}$$

$$3x + \left\lceil \left(\frac{s}{bfr} \right) \right\rceil + \frac{b + b_{I1}}{2} + \frac{r}{2} \tag{6.2}$$

The s term refers to the selection cardinality. When the attribute is nonkey, the value of s becomes r/d, where r is the number of records and d the number of distinct values [EN00]. Substituting this into Equation 6.2, we arrive at

$$3x + \left\lceil \left(\frac{r}{d \times bfr} \right) \right\rceil + \frac{b + b_{I1}}{2} + \frac{r}{2} \tag{6.3}$$

Through a similar analysis, the cost of σ_c on line 15 of Algorithm 6.1 is given as

$$2x + \left\lceil \left(\frac{r}{d \times bfr} \right) \right\rceil + \frac{b_{I1}}{2} + \frac{r}{2} \tag{6.4}$$

Note that σ_c requires only two selection conditions, an equality operation on a nonkey attribute ($value = q_n.value$) and an equality operation on a nonkey attribute ($parent\_id = q_c.id$).

In addition to the cost of the selection operations, the RDBQuery algorithm also depends on the distribution of descendant and child edges (nodes) in the query. We introduce the terms ϕ_d and ϕ_c to denote the number of descendant and child nodes in the query, respectively. Combining Equation 6.3 and Equation 6.4, we arrive at a total complexity of

$$\phi_d \sigma_d + \phi_c \sigma_c \tag{6.5}$$

$$\phi_d \left[3x + \left\lceil \left(\frac{r}{d \times bfr} \right) \right\rceil + \frac{b + b_{I1}}{2} + \frac{r}{2} \right] + \phi_c \left[2x + \left\lceil \left(\frac{r}{d \times bfr} \right) \right\rceil + \frac{b_{I1}}{2} + \frac{r}{2} \right] \tag{6.6}$$

Both values for $r/2$ are tuning parameters. As the selectivity decreases, the denominator will increase thereby lowering the value of $r/2$ to $r/3$, $r/4$, and so on. It is not necessary to consider the recursion of RDBQuery or the `forall` loop on line 7 of Algorithm 6.1. The effects of recursion and the for-all operator are subsumed by the parameters ϕ_d and ϕ_c. Since they represent the total number of descendant and child edges, they effectively represent all iterations of the RDBQuery algorithm. In the following chapters, the implications of the various parameters of all four techniques discussed are examined. We also establish an experimental framework for the analysis of each algorithm. Chapter 7 discusses TwigStack and Constraint Sequencing, the two native XML query methods, and Chapter 8 discusses the two algorithms, SS-Join and our own RDBQuery, that operate on XML data in a relational database.

6.4 Summary

In this chapter, we presented our original technique, RDBQuery, that queries XML documents using a single relation created from shredded XML data. This relation is implemented using a traditional relational database management system, and RDBQuery leverages the built-in methods and access paths to retrieve data from the relation. Our technique improves upon SS-Join [SLFW05] in that it allows for a query size larger than two nodes and enables us to answer twig queries. In Chapter 8, we investigate the performance of RDBQuery and SS-Join.

Chapter 7

Analysis of Individual Native XML Techniques

In previous chapters, we performed an algorithmic analysis on three existing techniques and a new technique of our own. Two of the techniques are native to XML and two use a relational database to store XML data. This chapter considers the two native XML techniques, TwigStack and Constraint Sequencing, presents a framework for analyzing their behavior, and presents the results of performance studies on the algorithms.

7.1 TwigStack

In Chapter 3, we present Equation 3.5, based on the work by Bruno et al. [BKS02], that represents the number of operations required during the execution of a TwigStack query. For ease of reference, this equation is repeated here as Equation 7.1. Table 7.1 presents the various parameters for the TwigStack algorithm that are used in Equation 7.1. Chapter 3 illustrates, on a small example, the salient parameters listed in the table.

$$\sum_{x \in q} (\sum_{\psi_x} |T_i| + |S_{parent(x)}|) + |S_{root}|. \tag{7.1}$$

Using the parameters in Table 7.1, we ran mathematical experiments to investigate the effects of the various parameters on overall query performance. The results of those experiments are

Parameter	*Name*	*Represents*
x	Node	Node in query q
q	Query	Query of length $\|q\|$ (number of nodes in query)
ψ_x	Local breadth	All nodes i that are children of node x
T_i	Stream of node i	Nodes in XML document that correspond to node i
$S_{parent(x)}$	Stack of $parent(x)$	Size of stack that corresponds to the parent of node x
S_{root}	Stack of root node	Size of stack that corresponds to the root node

Table 7.1: Parameters in the TwigStack Algorithm

discussed here with additional supporting material found in Appendix A. By observing the behavior of TwigStack as given in Equation 7.1, we postulate the following about its performance:

1. As the size of streams T_i increases, performance will degrade.
2. The effect of S_{root} will be minimal, if observed at all.
3. The four major contributing parameters will emerge as q, ψ_x, T_i, and $S_{parent(x)}$.
4. Of the four major contributing parameters, the size of the stream T_i will affect query performance the most with the local breadth ψ_x in second.

In addition, we make some general observations about XML documents that narrow the search space for our study. Typically, XML documents increase in specificity as they are traversed from the root node to leaf nodes. For this reason, stream sizes T_i tend to decrease as we approach the leaf level. Consider an XML document for movies. While there may exist many `movie` nodes, there are fewer `comedy` nodes and even fewer nodes for comedies produced in `2008`. While all comedies belong to the stream T_{comedy}, not all comedies will belong to the stream of comedies produced in `2008`. Since the TwigStack relies on intermediate results, the ability to prune those intermediate results efficiently is a key factor in its overall performance. The size of intermediate results is directly related to the size of the individual stacks S, and much attention is devoted to the exploration of the effect of the values of S in the following sections.

7.1.1 Effect of T_i

The size and behavior of the streams T of nodes in the query is of significant importance. This term relates back to the original XML document over which the query is processed. As stream

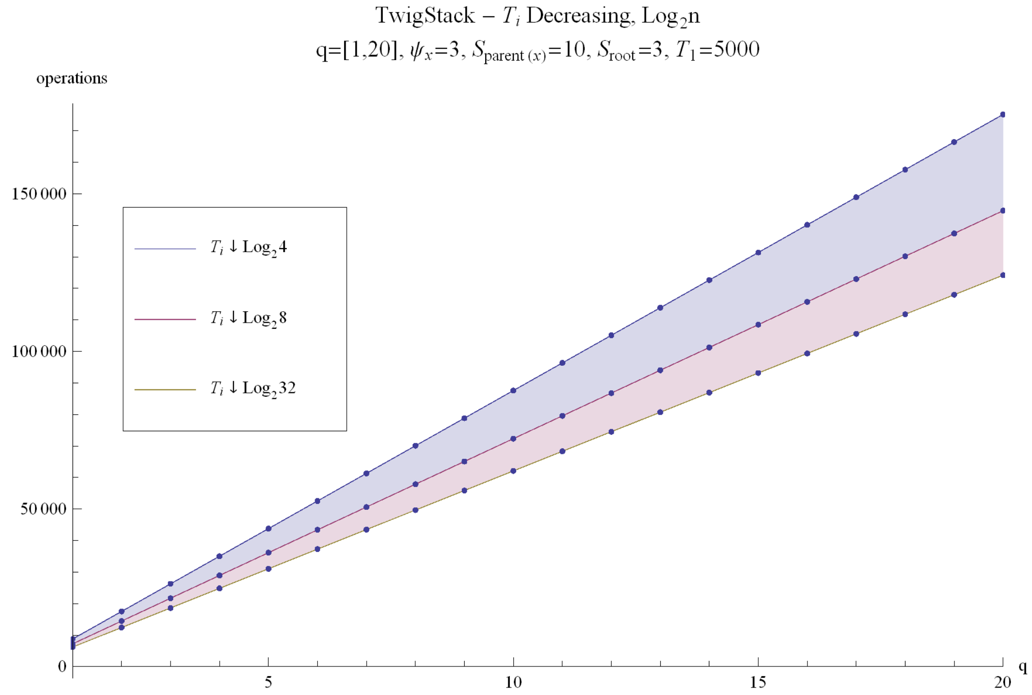


Figure 7.1: TwigStack, stream size decreasing by $\log_2 n$

size increases, TwigStack must do more work to search through the possible nodes and either push them onto the appropriate stack, not push them onto the appropriate stack, or push them only to later remove them if they do not contribute to a final result. We explore the effect of T_i and make some conclusions about its impact on query performance. Figure 7.1 and Figure 7.2 show the effect of decreasing the value of T_i by several factors. At the top of the figure, we note that this experiment is for query sizes q from one to 20. Unless otherwise stated, it is assumed that query sizes are discrete values in single-step increments. The rest of the TwigStack parameters are listed accordingly. When an equality is shown, it is read as a static value for the given parameters. If a parameter is allowed to assume a random value, that is noted by the capital letter R followed by the range for random values (first observed in Figure 7.4). The x-axis of Figure 7.1 illustrates query sizes (increasing from left to right) while the y-axis represents the number of operations required by the TwigStack algorithm. If an arrow is shown (as in the legend), it is interpreted as an increase or decrease in the value of the term. In Figure 7.1, we note that the stream sizes T_i are decreasing by a logarithmic amount.

In both cases, as T_i decreases, query performance increases. In Figure 7.2, there is a significant

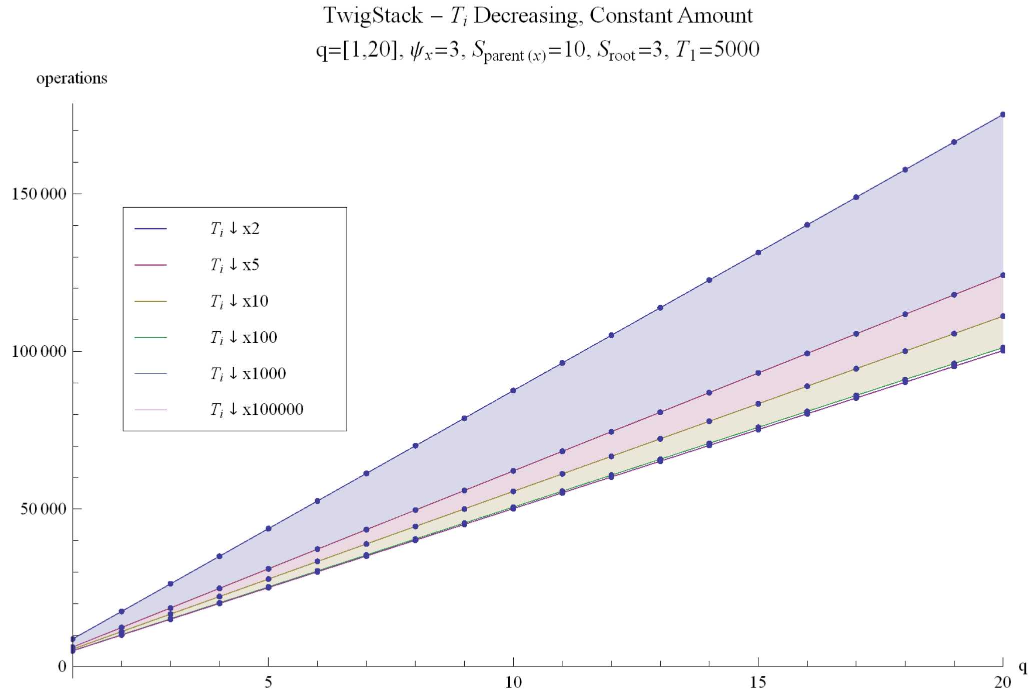


Figure 7.2: TwigStack, stream size decreasing by constant factors

decrease from 175,203 operations to 124,203 operations, a 30% reduction, when the stream size is changed from decreasing by a factor of two to decreasing by a factor of five. Notice that the gains produced by a smaller T_i value do not continue indefinitely. When T_i was decreased by a factor of 1000 and 100,000, the results were similar to those produced by T_i decreasing by a factor of 100. The same observation cannot be made when the value of T_i is strictly increasing as shown in Figure 7.3. For the same parameters, query performance quickly degrades when the size of T_i strictly increases by a constant factor. This leads us to conclude that the effect of T_i is significant, but there does exist a lower limit past which other factors of the algorithm overshadow the performance gains by a small stream size. These observations are substantiated by Figures A.1 and A.2.

While most XML documents are regular in structure and tend to decrease in stream size from root to leaf level, some documents do exhibit an erratic distribution of nodes at various levels. To best approximate this behavior, we created XML documents that allow for random stream sizes in various intervals running from 10 to 2000. A single run of TwigStack with this data is shown in Figure A.3, and Figure 7.4 shows the results of this same technique averaged over 250 executions. When stream sizes stay relatively low, between 10 and 100, TwigStack performance is relatively

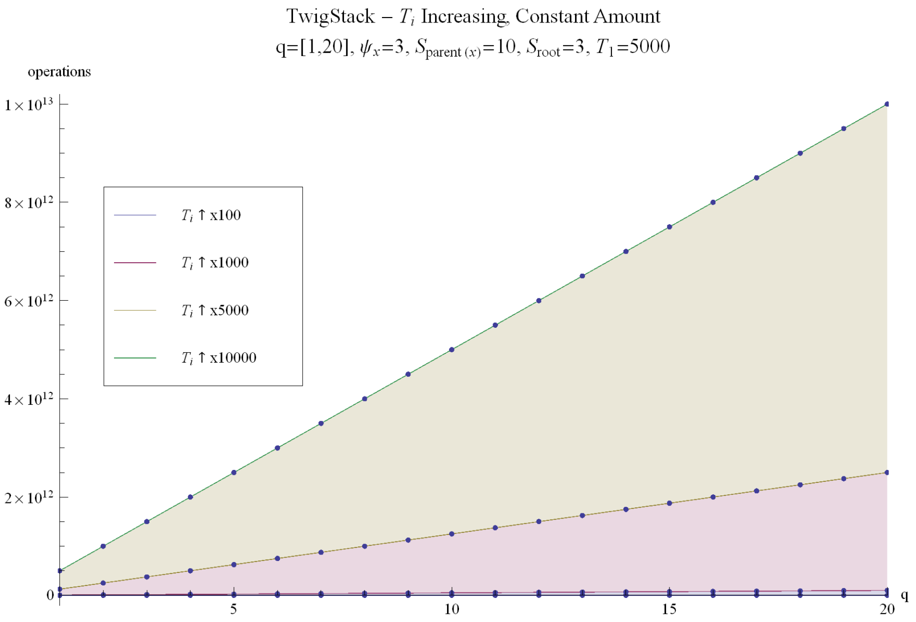


Figure 7.3: TwigStack, stream size increasing by constant factors

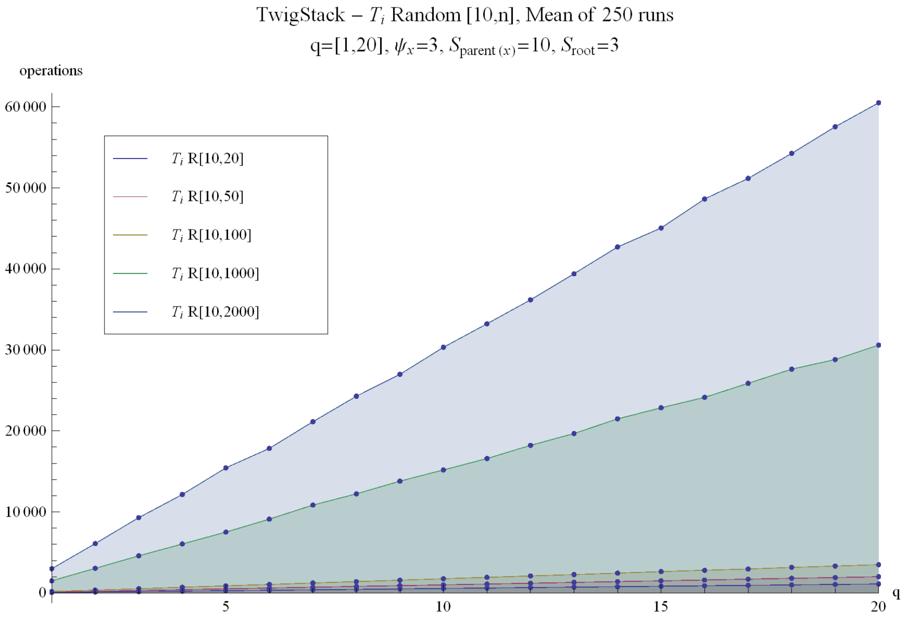


Figure 7.4: TwigStack, random stream sizes - 250 runs

consistent. It is important to note the scale on the graph. While the bottom three lines may appear similar, there is actually a 45% decrease in performance between the random execution with a maximum of 20 (represented by the bottom line) and the execution with a maximum of 50 (red line). This can be observed in Figure A.4. However, the importance of T_i becomes apparent when it is allowed to increase to a maximum of 1000 and 2000 nodes. These executions, represented by the top two lines in Figure 7.4, demonstrate a substantial degradation in TwigStack performance when stream sizes rise above low values. Figure A.5 shows similar behavior with a smaller query size.

From the experiments that isolate T_i, we draw several important conclusions about the TwigStack algorithm. The stream sizes have a beneficial impact on TwigStack query performance if the sizes are low or decrease as TwigStack progresses through the query. This benefit does have a limit, and the positive effects shown by smaller T_i values reaches a point of limiting returns. The same is not true in the reverse case, when T_i is large or allowed to steadily increase. As streams increase in size, operations necessary for query execution increase. In the case of random T_i values, query performance falls somewhere between the case where T_i is increasing and where T_i is decreasing. This is an expected result and corresponds to the effect of the inner summation in Equation 7.1.

7.1.2 Effect of $S_{parent(x)}$

As with the stream size, the size of the stacks TwigStack uses to build query results is an important factor that governs overall query performance. As outlined in Chapter 3, TwigStack builds intermediate results using multiple stacks, one stack for each node in the query. As nodes found in the streams are matched, they are pushed onto the appropriate stack. As the stacks grow in size, it takes additional operations to iterate through them to ensure that they represent a correct answer to a given partial query. In a set of experiments similar to those in Section 7.1.1, we present performance charts that isolate the $S_{parent(x)}$ parameter. Figure 7.5 shows the effect of increasing the stack size by a constant factor (2, 3, 4, and 5 times). These runs assume a starting case of $S_{parent(1)} = 3$. A similar result shown in Figure A.6 is produced by a larger base case. It is clear from the graph that the number of nodes pushed onto the individual stacks has a dramatic impact on query performance. With a relatively low base case of $S_{parent(x)} = 3$, the number of TwigStack operations quickly explodes for extremely small queries (less than four nodes in the query). Ex-

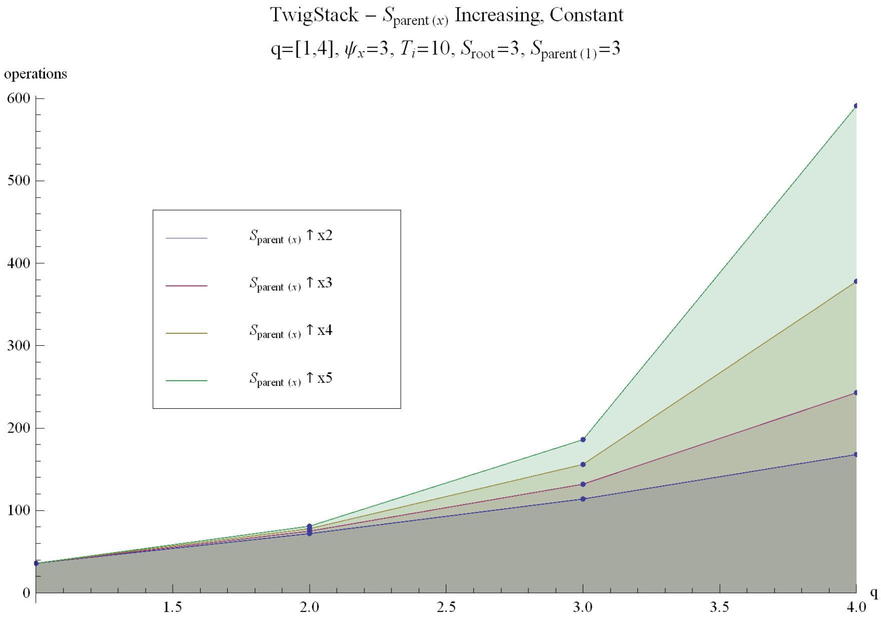


Figure 7.5: TwigStack, stack size increasing by constant factors

ploring the effect of the bottom two lines, Figure 7.6 shows that, as query size increases, the effect of the increasing stacks causes a dramatic degradation in the TwigStack algorithm. Figure 7.7 shows just the bottom line in Figure 7.6. Here it is clear that a strictly increasing value for stack size (translated as more intermediate results) causes TwigStack to incur additional overhead in the processing and potential pruning of the stacks. Three additional graphs shown in Figures A.7, A.8, and 7.7 show similar results for a smaller base case of $S_{parent(x)} = 1$.

By comparing these results with the results of T_i strictly increasing, it appears that $S_{parent(x)}$ has a less profound effect on TwigStack performance than T_i. However, as was the case in Section 7.1.1, it is a rare case when an XML document and query would result in strictly increasing stacks as nodes progress from root to leaf level. A more representative case can be shown by allowing the stacks to assume random values, as shown in Figure 7.8. While this appears somewhat regular, a graph with the same data run 250 times, Figure 7.9 shows a more regular pattern. While Figure 7.9 suggests that $S_{parent(x)}$ has less effect on query performance than T_i, it is useful to allow $S_{parent(x)}$ to become as large as a possible value for T_i in Section 7.1.1. Figure 7.10 shows the results of such an experiment after 250 runs (a single run is shown in Figure A.10). By examining Figures 7.4, 7.9, and 7.10, we can conclude that a larger value for T_i produces a larger decrease in

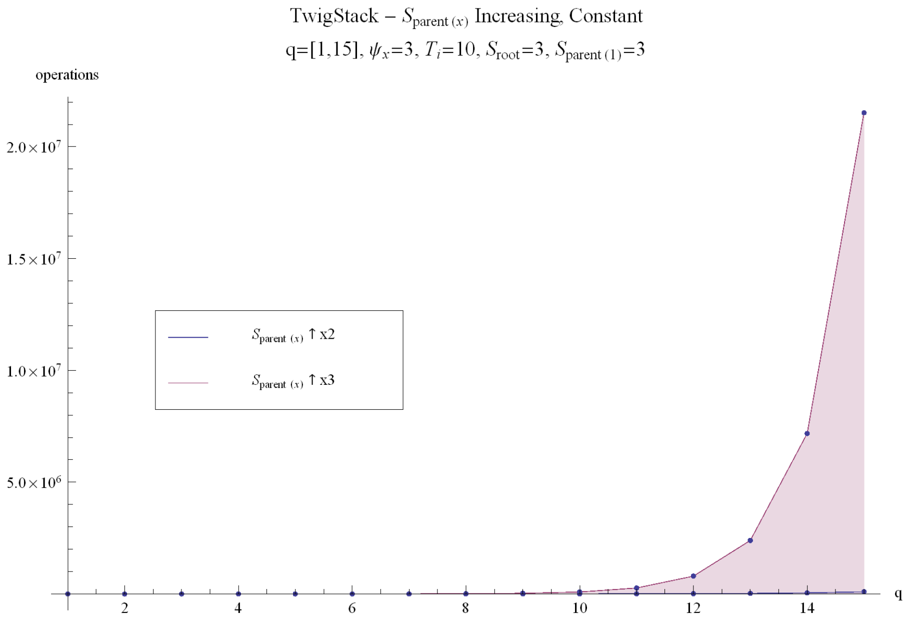


Figure 7.6: TwigStack, stack size increasing by constant factors (larger query)

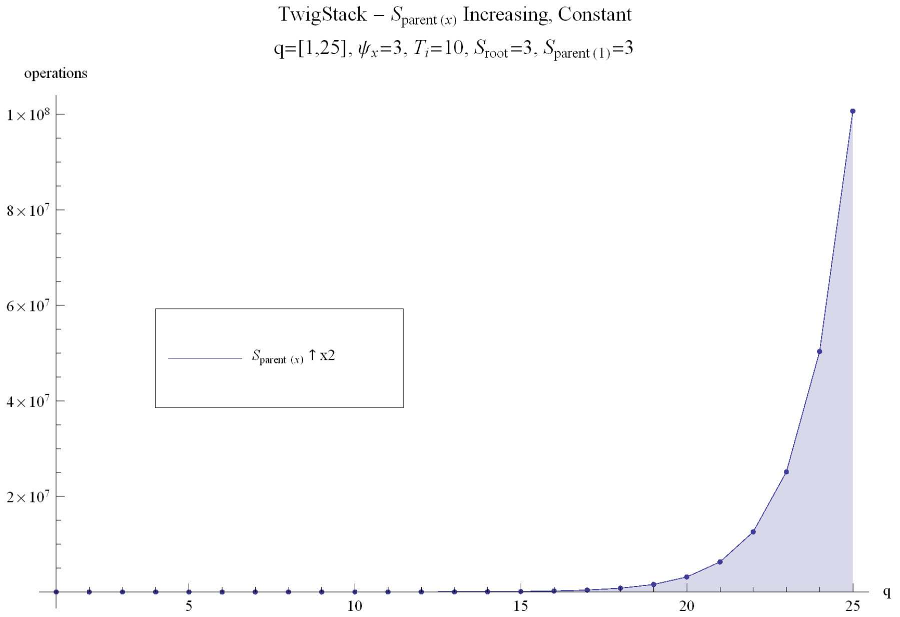


Figure 7.7: TwigStack, stack size increasing by constant factor 2 (larger query)

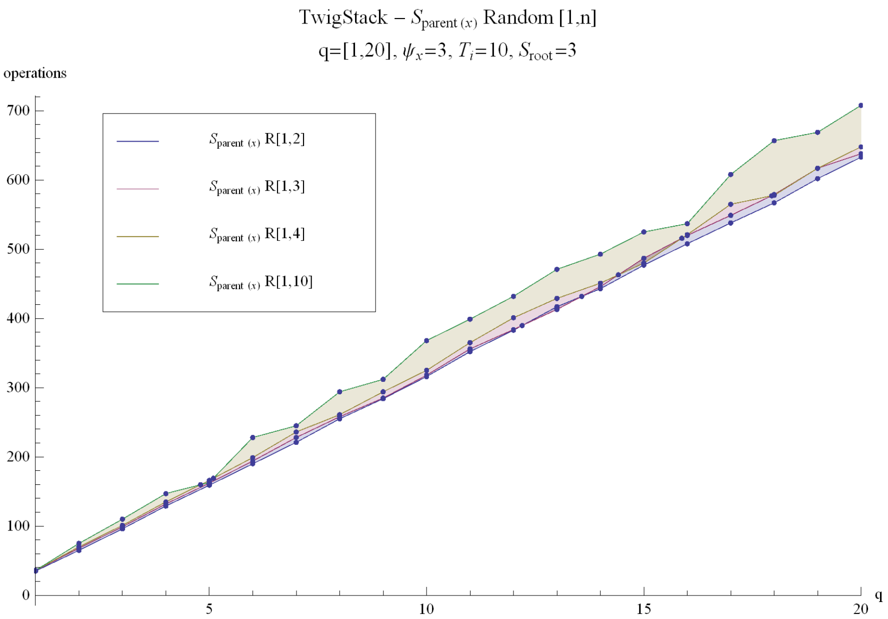


Figure 7.8: TwigStack, random stack size up to 10 - single run

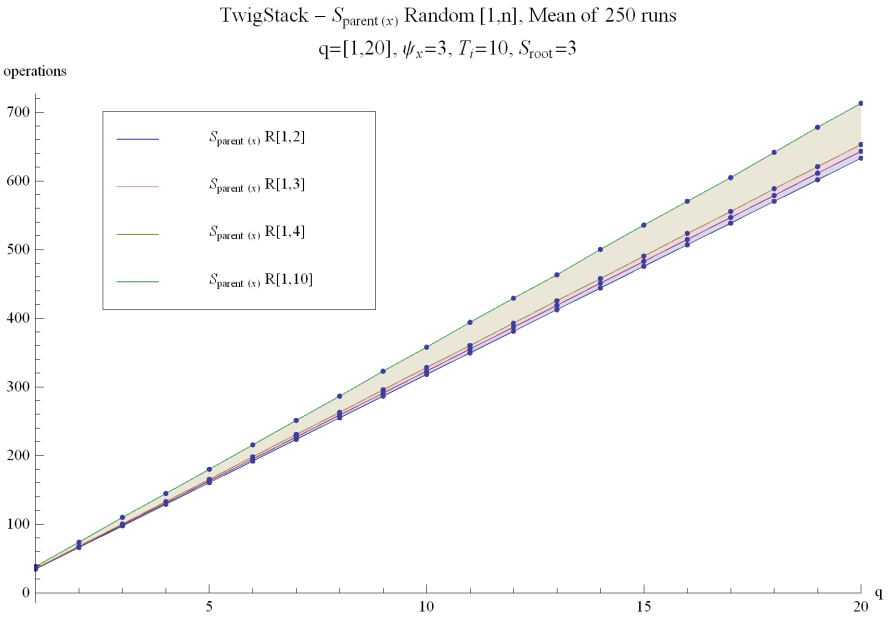


Figure 7.9: TwigStack, random stack size up to 10 - 250 runs

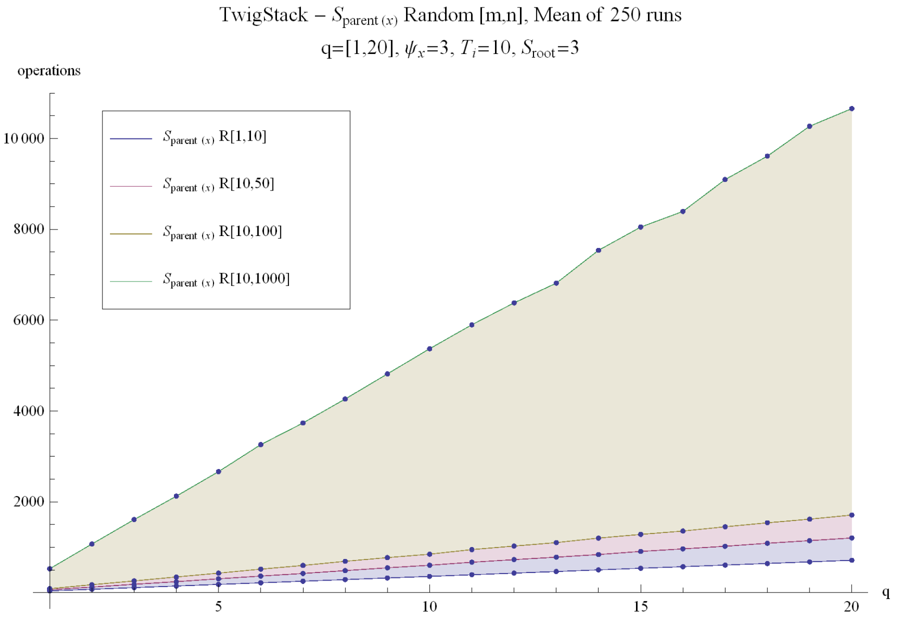


Figure 7.10: TwigStack, random stack size up to 1000 - 250 runs

query performance than larger values for $S_{parent(x)}$. While Figure 7.10 shows that large values for $S_{parent(x)}$ can negatively impact TwigStack performance, for a typical query size of 20 the number of operations is around 10,000 for a high $S_{parent(x)}$ and approaches 30,000 for a similar value of T_i. Equation 7.1 shows that the value for $S_{parent(x)}$ is outside of the inner summation and is only affected by the outer summation (over x). It stands to reason that the effect of $S_{parent(x)}$ and T_i, for similar values of each, has less impact on query performance than T_i. To further investigate this, we performed a series of experiments that varied both T_i and $S_{parent(x)}$ and show their results on the same graph.

Figure 7.11 shows the effect of decreasing T_i while simultaneously increasing $S_{parent(x)}$. The area shaded in green represents $S_{parent(x)}$ increasing by a factor of three while the area shaded in blue represents the same parameter increasing by a factor of two. After a query size of eight ($q = 8$ on the x-axis), the negative effect of an increasing value of the stacks overtakes the benefits felt by a decreasing stream size. This same effect is shown in Figure A.11 for a larger query. A more accurate model allows both T_i and $S_{parent(x)}$ to assume random values typical for a medium-sized XML document. Figure A.12 shows a single run while Figure 7.12 shows the results after 250 runs with random values for the two parameters. The upper lines correspond to higher values of T_i

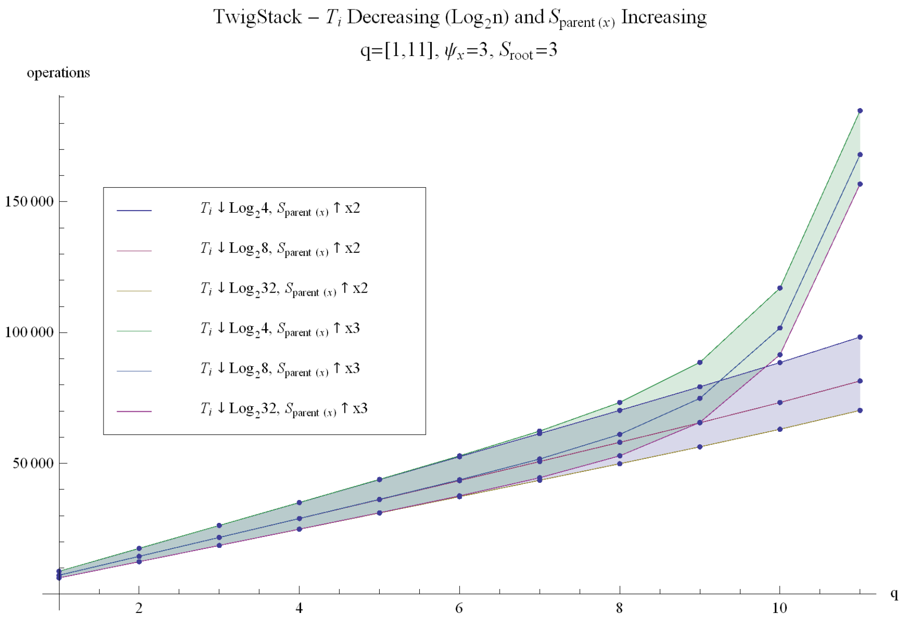


Figure 7.11: TwigStack, stream size decreasing and stack size increasing

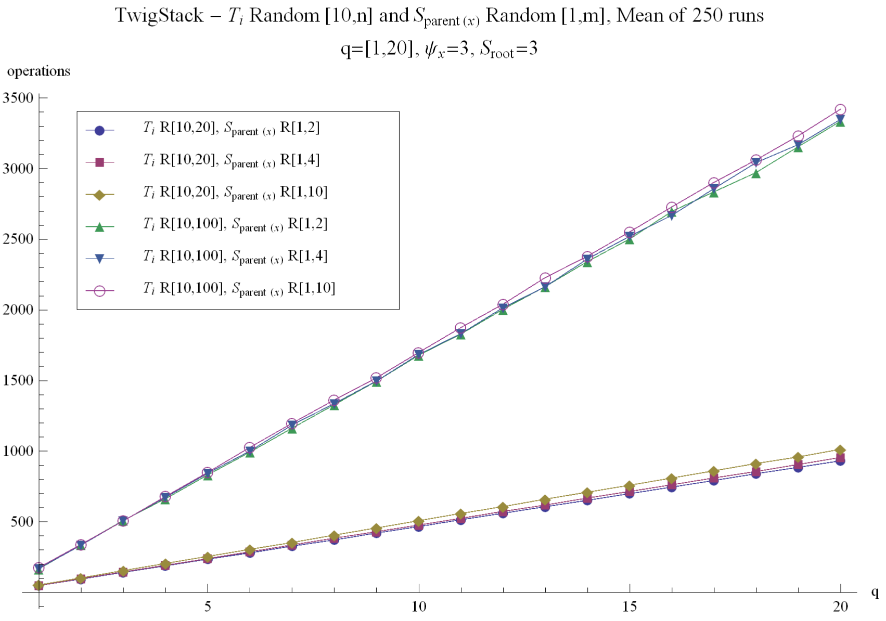


Figure 7.12: TwigStack, random stream and stack sizes - 250 runs

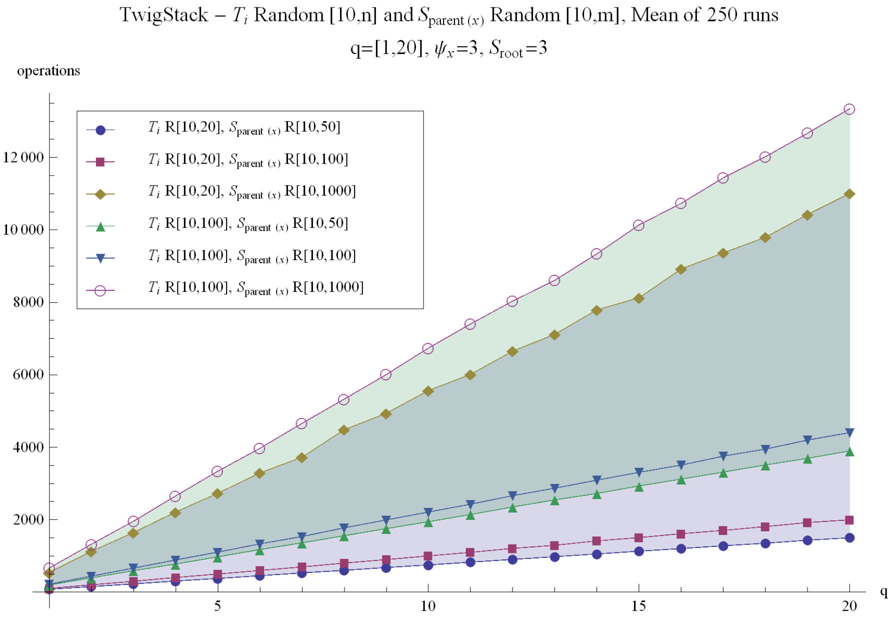


Figure 7.13: TwigStack, random stream and stack sizes (larger stacks) - 250 runs

and various values for $S_{parent(x)}$ while the lower lines corresponds to lower T_i values with the same $S_{parent(x)}$ values. This graph reinforces the earlier observation that $S_{parent(x)}$ has a smaller impact on TwigStack performance than stream size. However, if we allow the two parameters to assume similar values, as shown in Figure 7.13, the results become less clear. The darker region in the center (between the diamond line and the green triangle line) represents the area where $S_{parent(x)}$ has a more substantial impact on query performance than T_i. In more general terms, if a TwigStack query is performed over a relatively small document but results in a high number of intermediate and final results, performance is determined by the number of results and not the document size. However, in the case where fewer results are produced over a large document, performance is limited by the stream size (T_i).

7.1.3 Effect of ψ_x

The parameter ψ_x (abbreviated as ψ for simplicity) reflects the breadth or fan-out of the XML query by dictating the number of children for a given node x that the algorithm must process. In other words, ψ_x represents the set of nodes that are children of node x. A larger ψ corresponds to more children and, in turn, a higher fan-out at node x. Figure 7.14 illustrates the effect on

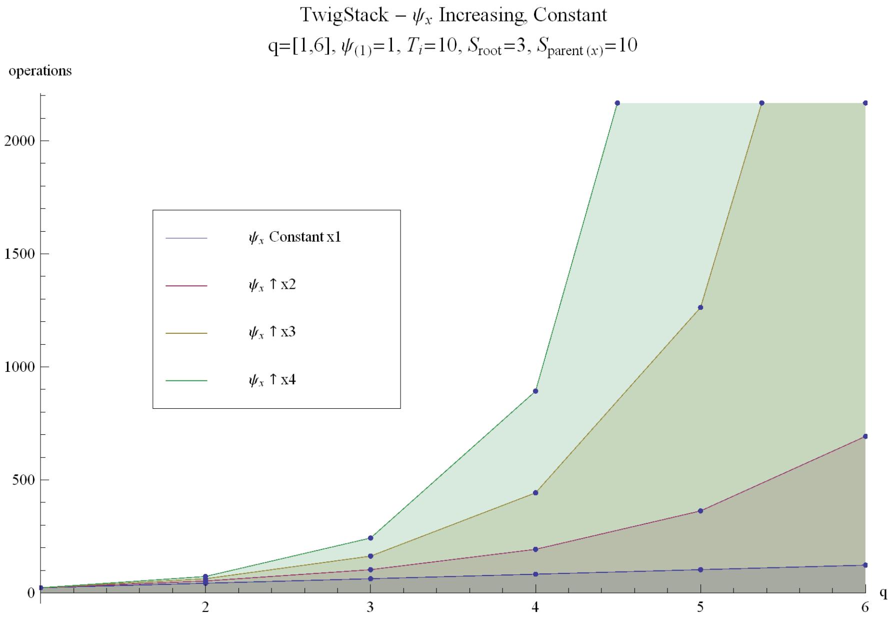


Figure 7.14: TwigStack, query fan-out increasing

query performance if the number of children nodes is constant, resulting in a completely regular query tree. The bottom line represents a baseline with $\psi = 1$, and this corresponds to a simple query where there is no branching. As the tree is allowed to have more children, query performance quickly degrades as TwigStack must perform the inner summation for each node that is a child of the outer summation. In other words, the effect of a high fan-out is felt in both summations, and this causes an explosion in the number of operations. By increasing ψ, TwigStack is forced to look at additional streams for all children of a given node in order to assure they satisfy the partial query before a query node is pushed onto the appropriate stack.

Since XML queries rarely result in such a regular tree structure, a more representative result is found by allowing ψ to assume random values under certain constraints. A single run of such an experiment is shown in Figure A.13, and a mean of 250 runs is shown in Figure 7.15 While the results are less dramatic than those shown in Figure 7.14, they clearly show that TwigStack performance is negatively impacted by more children in the query tree. This is what we expected from the TwigStack algorithm. By increasing the value of ψ, the inner summation must iterate through more streams in the original XML document.

In order to gauge the relative impact of ψ in relation to T_i and $S_{parent(x)}$, we designed experi-

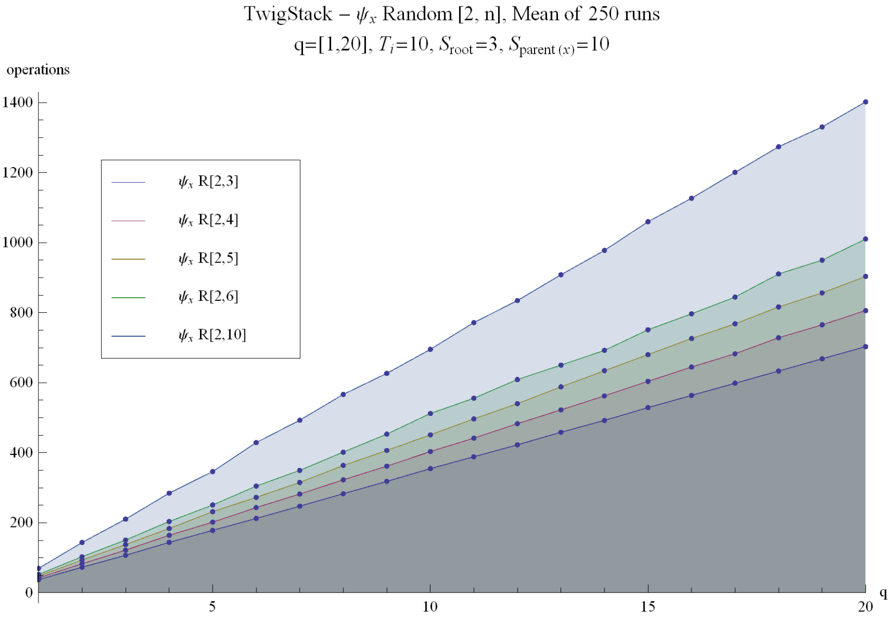


Figure 7.15: TwigStack, random query fan-out - 250 runs

ments that varied both parameters and show the results on the same graph. Figure 7.16 shows the effect of simultaneously decreasing T_i while increasing ψ. When the value of ψ remains low, the effect of a decreasing T_i is minimal. This is shown by the two lines at the bottom of the shaded region. They represent T_i decreasing by $\log_2 32$ while the value of ψ is increasing by factors of two and three. However, when the number of children increase to a factor of three, the effect of a decreasing stream size is observed more clearly. This is shown by the top two lines in Figure 7.16. This result is reinforced by the experimental results shown in Figure 7.17. This graph shows the results of an experiment where T_i is strictly decreasing while ψ is allowed to assume random values under various constraints. It shows a clear delineation between the queries with lower T_i values (the bottom three lines) and the queries with higher T_i values (the upper three lines). The decrease in stream size, which results in lower stream lengths, affects query performance more than the fan-out of children nodes. Figure 7.18 shows the results of an experiment where both T_i and ψ are randomly created and run 250 times. A single run is shown in Figure A.14. The area shaded in green represents higher potential streams (randomly created from 10 to 100) across various ψ values. The area shaded in blue represents lower stream sizes from 10 to 20. This graph shows two important properties of the TwigStack algorithm. First, for typical XML documents and queries,

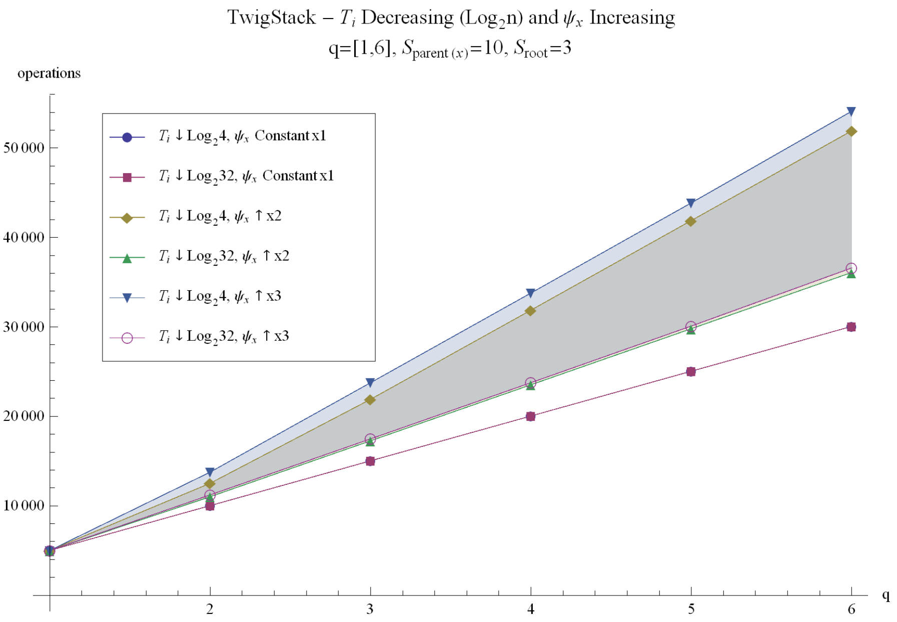


Figure 7.16: TwigStack, stream size decreasing and query fan-out increasing

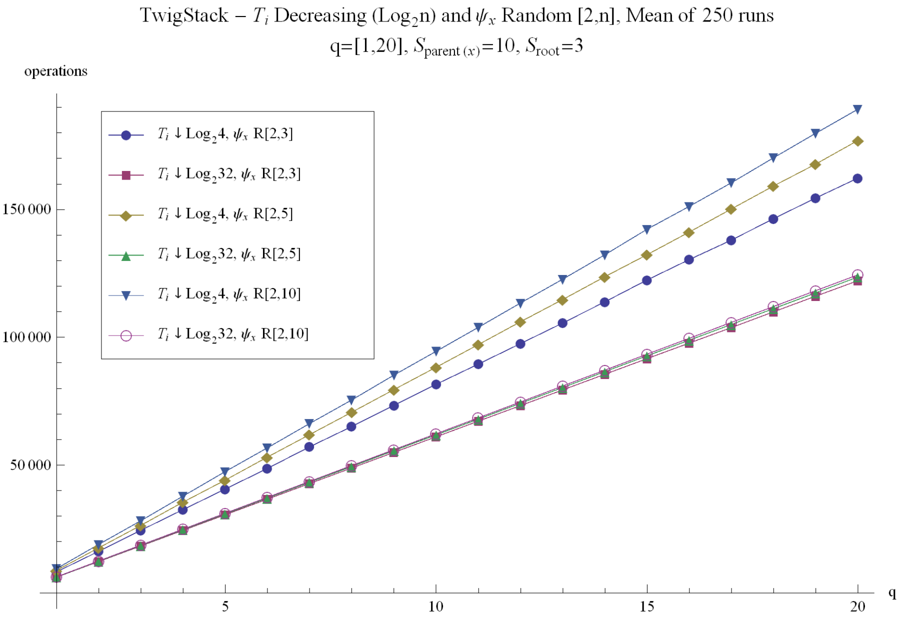


Figure 7.17: TwigStack, stream size decreasing and random query fan-out - 250 runs

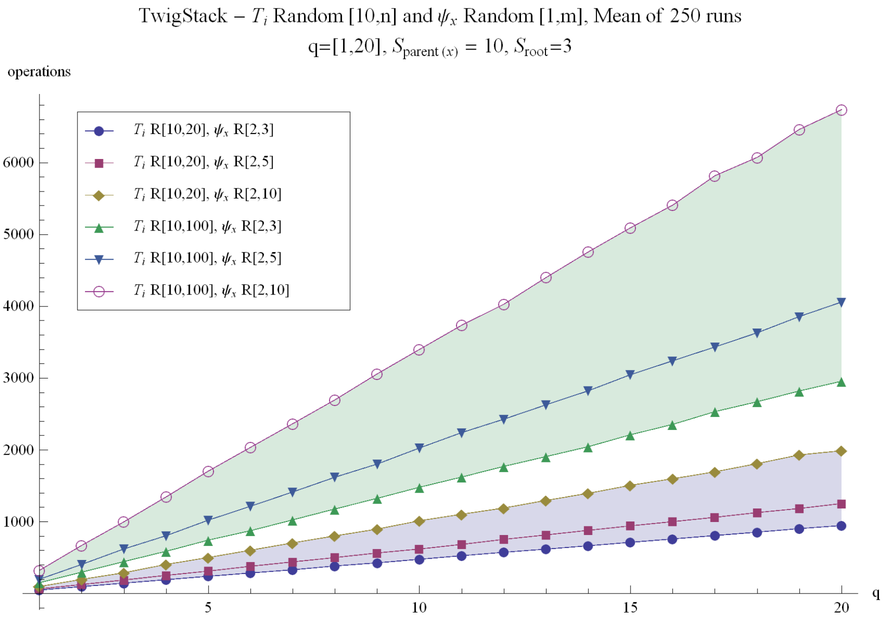


Figure 7.18: TwigStack, random stream sizes and query fan-out - 250 runs

increasing the stream size has a larger impact on query performance than increasing the number of children. Second, for larger stream sizes, increasing the number of children has a larger negative effect on query performance than by increasing the number of children with smaller stream sizes. Figure 7.19 is more complicated in that it allows similar values for T_i and ψ and results in a higher degree of fan-out. Similar to Figure 7.13, the dark shaded region represents where a higher stream size with a lower number of children performs better than a lower stream size with a higher number of children. In this area, queries with lower stream sizes and higher local breadth performs worse than those with a higher stream size. Due to the nature of XML documents, the stream size of a node will typically overtake the local breadth around a given node. While the local breadth may become high, it represents only the children around a specific node. The stream of a node represents all of the nodes of that type in the document.

The final experiment for TwigStack compares the effect of ψ to $S_{parent(x)}$. A single run of an experiment with random values for both parameters is shown in Figure A.15, and the mean of 250 runs is shown in Figure 7.20. The shaded slices represent runs with similar ψ values while the value of $S_{parent(x)}$ varies. For example, the bottom slice maintains ψ as a random value between two and three while the potential values for $S_{parent(x)}$ increase from a maximum of two to a maximum of ten.

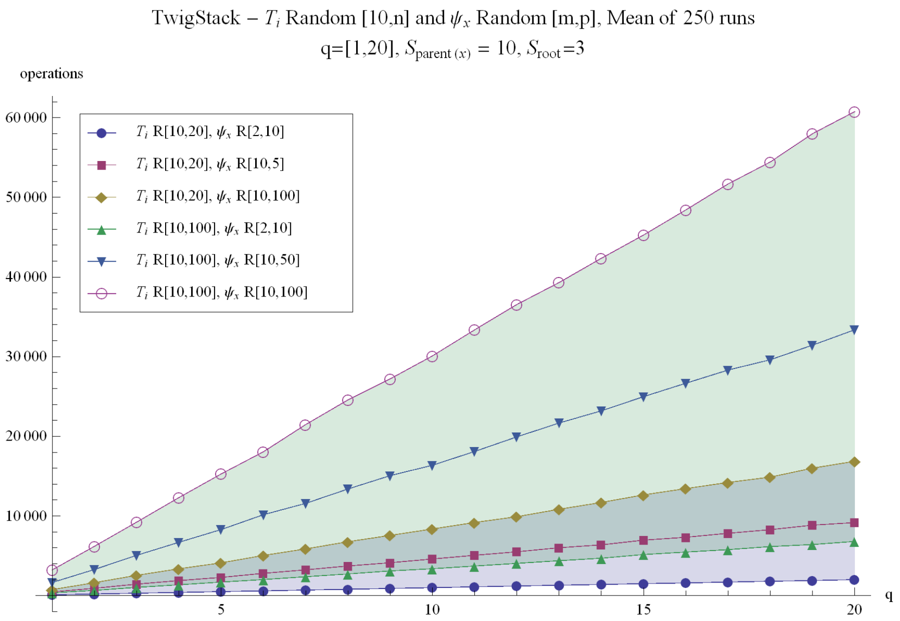


Figure 7.19: TwigStack, random stream sizes and query fan-out (larger fan-out range) - 250 runs

Since the local breadth (ψ) may overtake the amount of intermediate results, Figure 7.21 illustrates the performance of TwigStack when local breadth overtakes the stack size. Similar to Figure 7.20, lines that represent similar values for ψ group together and negate changes to $S_{parent(x)}$. Even with larger stack values, shown in Figure A.16, the higher local breadth drives the increase in TwigStack operations.

7.1.4 Summary of Effects by TwigStack Parameters

The experimental results show that four parameters in Table 7.1 contribute to the overall performance of the algorithm. First, the number of nodes involved in the query q affect the number of TwigStack operations by increasing the outer summation of Equation 7.1. This effect is linear since TwigStack must simply iterate through additional query nodes. All of the results presented show that an increase in q always results in increased operations for TwigStack. Second, the effect of the stream size T_i of a node i has the largest impact on query performance. With more nodes (data) in the original XML document, the time necessary for TwigStack to iterate through the stream T_i for each node in the query and all of its children nodes increases, and this effect is illustrated by the inner summation of the algorithm. Third, the local breadth ψ_x around a particular node x

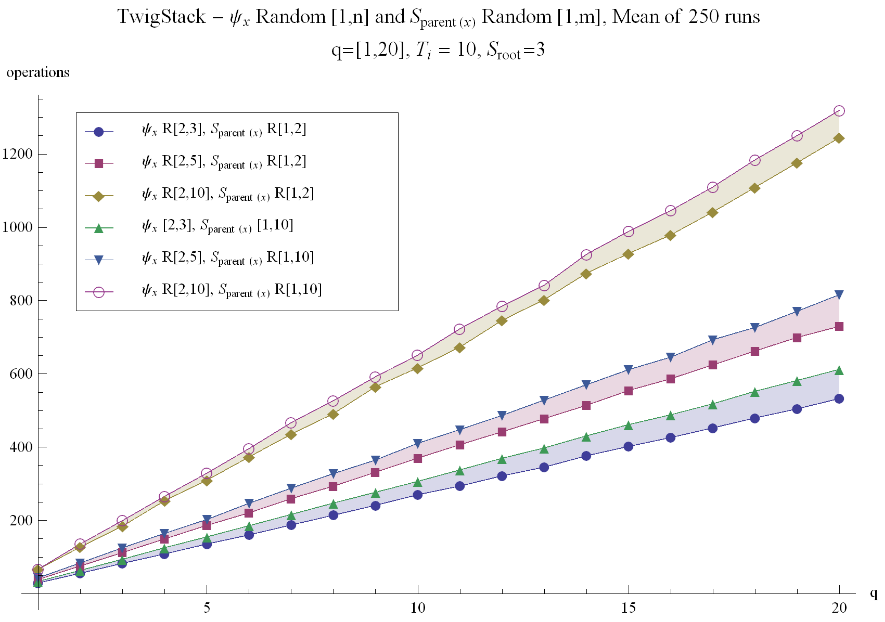


Figure 7.20: TwigStack, random query fan-out and stack sizes - 250 runs

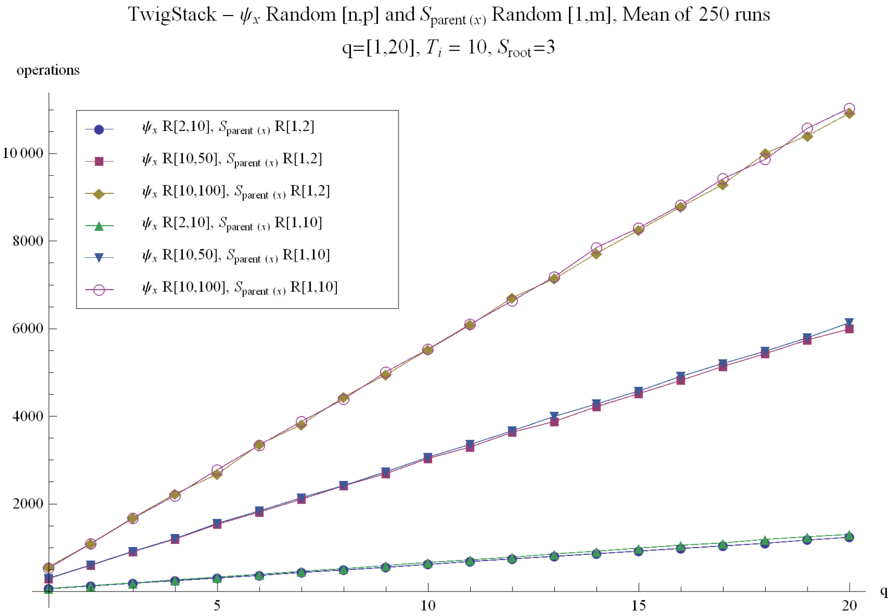


Figure 7.21: TwigStack, random query fan-out and stack sizes (larger fan-out range) - 250 runs

Parameter	*Name*	*Represents*
q	Query	Query of length $\|q\|$
m	Document length	Size of document that must be searched
b	Branching factor	Average amount of fan-out encountered in the document
s	Identical sibling nodes	Quantity of identical sibling nodes in the document

Table 7.2: Parameters in the Constraint Sequencing Algorithm

also significantly impacts query performance, and is second in that effect only to the stream size. Fourth, the size of the stacks $S_{parent(x)}$ also impact query performance but not to the degree that T_i and ψ_x degrade performance. The final parameter, S_{root} is not involved in any summation, and its effects can be ignored when compared to the large contributions by the other parameters.

7.2 Constraint Sequencing

In Chapter 4, we present Equation 4.9, based on the work of Wang and Meng [WM05], that represents the number of operations required during the execution of a constraint sequence query. Table 7.2 presents the various parameters for the Constraint Sequencing algorithm that are used in Equation 4.9. For ease of reference, this equation is repeated here as Equation 7.2. Chapter 4 illustrates, on a small example, the salient parameters listed in the table.

$$4|q| \times m \times (\log_2(b) + s\log_2(s)) \tag{7.2}$$

Using the parameters in Table 7.2, we ran mathematical experiments to investigate the effects of the various parameters on overall query performance. The results of those experiments are discussed here with additional supporting material found in Appendix B. By observing the behavior of the Constraint Sequencing algorithm as given in Equation 7.2, we postulate the following about its behavior:

1. The effect of the branching factor b will be small when compared with the other parameters.
2. The three major contributing parameters will emerge as q, m, and s.
3. Of the three major contributing parameters, the document size m and number of identical sibling nodes s will emerge as the factors that affect query performance the most.

The same general observations we made about XML documents in Section 7.1 also apply to the XML documents considered in this section. In addition, the parameter m is used to specify the document that must be considered by Constraint Sequencing. Due to the two binary searches performed that can significantly limit the search space in the document, m need not represent the entire document (although this can be the case). For the Constraint Sequencing algorithm, only the portion of the document that satisfies the two binary searches (Equation 4.2) factors into the total number of required operations. For simplicity, we state m as document size, but it could easily refer to just a portion of the document. For example, $m = 5000$ could represent the 5000 nodes of interest in a document of size 6000. It could also represent the entire document with a size (length) of 5000. In either case, the effect on query performance is the same since Constraint Sequencing ignores all nodes outside of a given range. In later chapters, we use Equation 4.10, a more general form of the Constraint Sequencing model that does not assume a worst-case scenario, to permit sequencing to operate over a set percentage of nodes in a document. Since this section is only concerned with Constraint Sequencing and how the parameters affect query performance, it is sufficient to assume a worst case scenario and use Equation 7.2 in lieu of the more general form.

7.2.1 Effect of m

The parameter m represents the size of the document over which Constraint Sequencing is executes. Figure 7.22 shows the result of increasing document sizes m across a query of length 20. As expected, as the document size grows, query performance degrades. Query performance is directly related to the size of the document over which the query is performed. Equation 7.2 illustrates this by multiplying the final result by m. As m increases, the number of operations Constraint Sequencing must perform increases with a linear relationship. The parameter m is investigated further when compared to the effect of the other parameters later in this chapter.

7.2.2 Effect of b

Branching factor is represented by b in Equation 7.2 and Table 7.2. Assuming a small document size and low identical sibling nodes, we observe the effect of a static branching factor in Figure 7.23. This experiment assumes a completely regular tree structure with constant branching factors shown in the legend. As the branching factor increases, query performance slightly degrades. Another

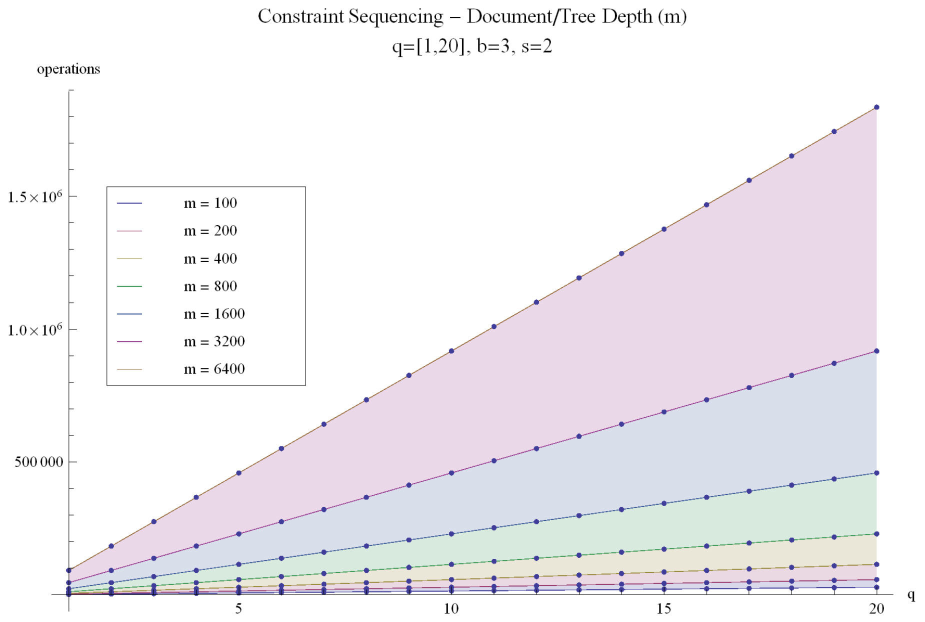


Figure 7.22: Constraint Sequencing, various document sizes

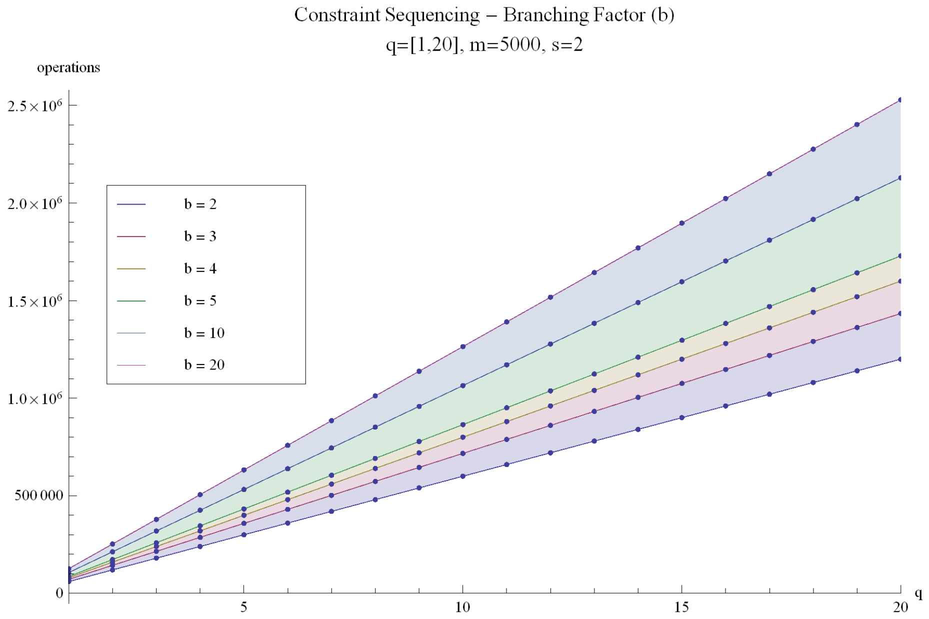


Figure 7.23: Constraint Sequencing, various branching factors

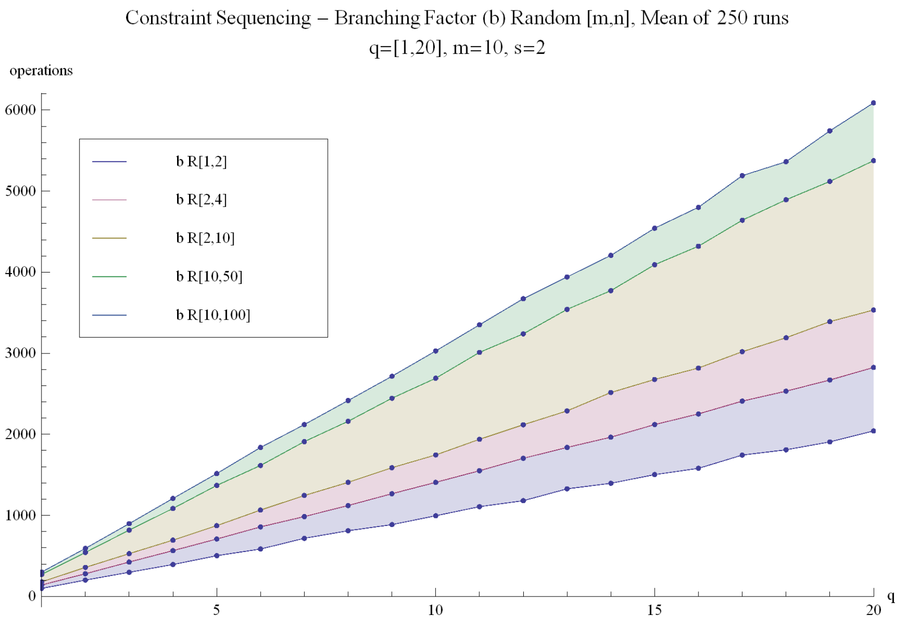


Figure 7.24: Constraint Sequencing, random branching factor - 250 runs

experiment allows b to assume random values as may be encountered in a typical XML document. Figure B.1 shows a single run with random data for b while Figure 7.24 shows an average after 250 runs with random data. The general trends of Figures 7.23 and 7.24 appear similar, but the scale for the number of operations (y-axis) is dramatically smaller for the random data. As expected, a completely regular tree has a high degree of fan-out compared with a tree that may have varying degrees of fan-out around particular nodes. In addition, the document size of Figure 7.23 is 500 times larger than that of Figure 7.24.

From these graphs, we conclude that the branching factor of a document does have an impact on query performance. However, even when the value of b is increased by a factor of 10 (Figure 7.23), this does not result in a corresponding linear increase in the number of operations. Rather, the number of operations is increased by $\log_2(b)$. The term is dominated by other parameters in the algorithm. Figure 7.25 further emphasizes that document size m is a more dominate factor in query performance than branching factor. The bottom region corresponds to small document sizes while the larger region on the top represents larger document sizes. The degradation in query performance caused by an increase in document size is larger than that caused by a higher branching factor. However, as document sizes increase, the effect of fan-out has a larger impact.

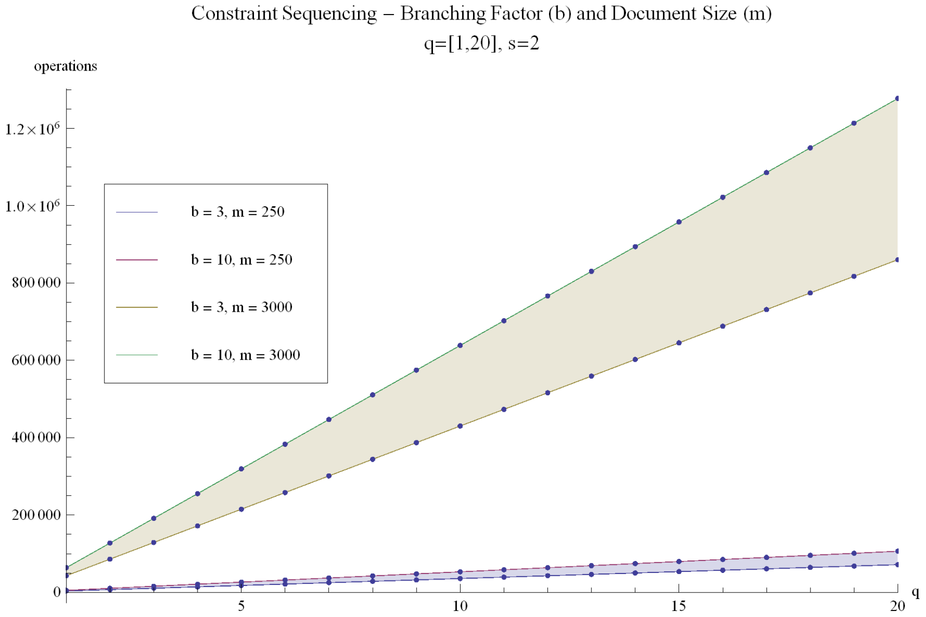


Figure 7.25: Constraint Sequencing, branching factor and document size high/low

7.2.3 Effect of s

In Section 4.4.2, the presence of identical sibling nodes (s) has a negative effect on the performance of Constraint Sequencing. In the worst case, an abundance of identical sibling nodes forces the Constraint Sequencing algorithm to perform like a depth-first search. In this section, we present experiments that explore the effect of identical sibling nodes. To illustrate the concept of identical sibling nodes, imagine an XML document that stores information about movies. One possible way to structure this document is with multiple children nodes (all of the root node) that correspond to the genre of the movie (comedy, action, and drama, for example). Instead of grouping every comedic movie underneath a single comedy node, suppose we allow the comedy node to be repeated for each movie. Then, at the level directly beneath the root node, we observe that there are many identical nodes in that they all specify a comedy genre. If all of these nodes are on the same level, they are siblings with each other (since they have the same parent). If they are the same node and value, they are considered identical sibling nodes. For our experiments, the value of s represents the total number of identical sibling nodes at each level in the original XML document.

Figure 7.26 shows the result of an experiment that fixed b and m and allows the number of identical sibling nodes to increase from two to 64. For lower values of s (shown in more detail in

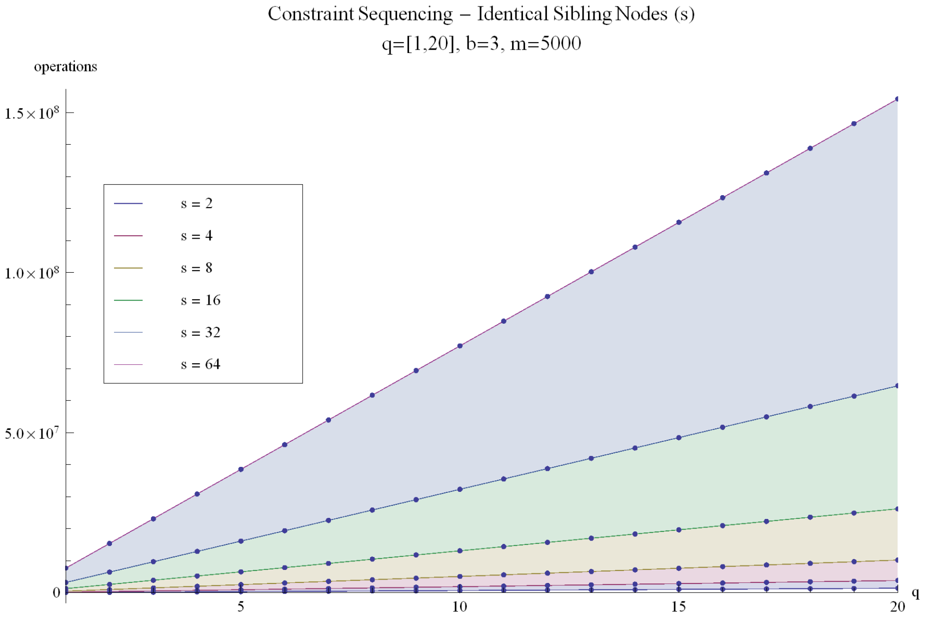


Figure 7.26: Constraint Sequencing, occurrence of identical sibling nodes

Figure B.2), smaller query sizes q result in similar performance to larger query sizes. However, when s increases, its effect is felt on larger queries. Figure B.3 shows a similar result but with a lower branching factor. While this is an expected result, XML documents may not have a uniform distribution of identical sibling nodes. We designed another set of experiments that explores the effect of random values for s.

Figure 7.27 shows the results of Constraint Sequencing queries with random s values in the ranges shown. A single run is shown in Figure B.4. The steep increase of the top line, corresponding to random s values between 10 and 1000, eclipses the behavior of smaller values. Figures 7.28 and B.5 show the same results but with the top line removed. Using Figure 7.28, we can make more accurate observations about the behavior of Constraint Sequencing in the presence of identical sibling nodes. Even a small value for s quickly results in more operations performed by the algorithm. We observe that documents that include identical sibling nodes incur a substantial slowdown in their execution time. However, the bottom two lines still appear to track with each other. Figures 7.29 and 7.30 isolate the two lowest lines shown in Figure 7.28. With a relatively small query size of $q < 5$, both scenarios behave similarly. However, once the number of query nodes passes five, the top line in Figure 7.29 starts to pull away from the bottom line. This leads us

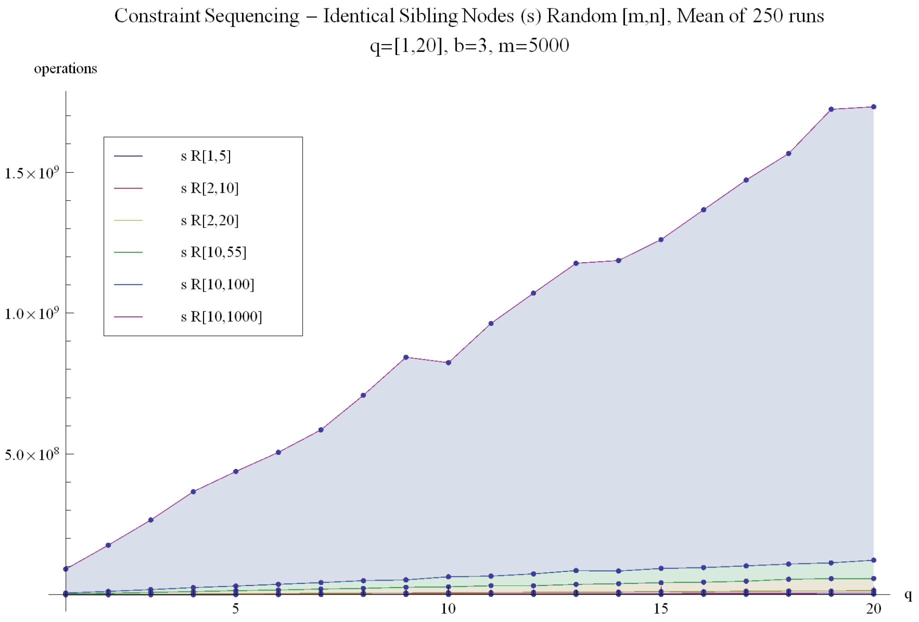


Figure 7.27: Constraint Sequencing, random identical sibling nodes (1000 max) - 250 runs

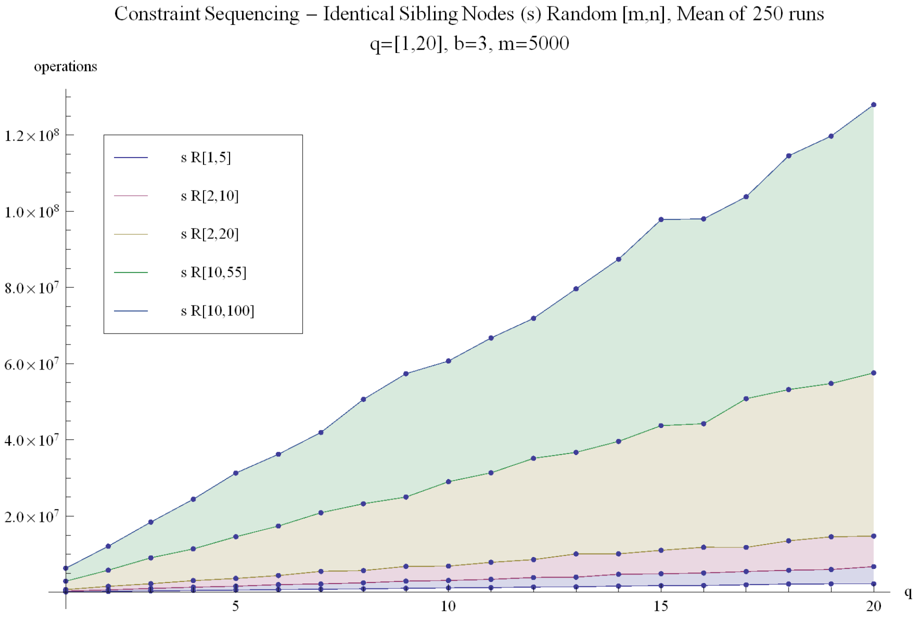


Figure 7.28: Constraint Sequencing, random identical sibling nodes (100 max) - 250 runs

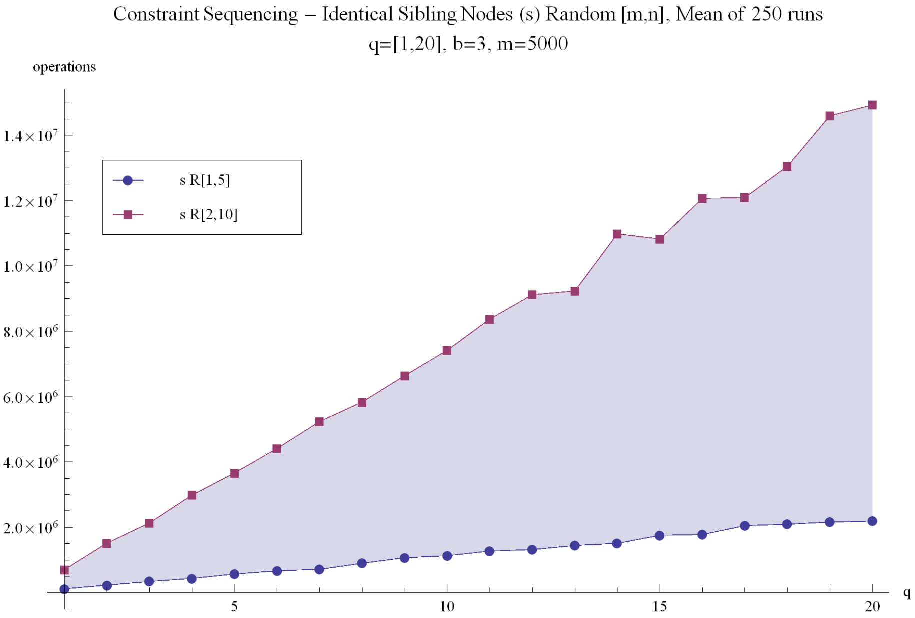


Figure 7.29: Constraint Sequencing, random identical sibling nodes (10 max) - 250 runs

to observe that the value of s is more significant in larger queries. To better illustrate this, Figure 7.30 shows the same ranges for s but with a maximum query size of $q = 100$. Here we observe that as query size increases, overall query performance degrades more quickly with a higher number of identical sibling nodes.

While we can observe the effect of s by itself, it is useful to design and run experiments that permute s with the other Constraint Sequencing parameters shown in Table 7.2. Figures 7.31 and B.6 show the effect of varying the branching factor b by random intervals while maintaining a constant value for s. An important observation from this graph is that query executions with similar s values cluster together regardless of their branching factors. This leads us to the conclusion that the effect of identical sibling nodes is higher than an increased branching factor, even when the branching factor is much larger than the number of identical sibling nodes. Figure B.7 displays similar results but with constant branching factor and sibling nodes values. When we allow both b and s to assume random values, the results, shown in Figures B.8 and B.9, do not significantly differ from those already discussed.

As further reinforcement to the observation that the presence of identical sibling nodes significantly degrades query performance, we observe the behavior shown in Figure 7.32. The blue line

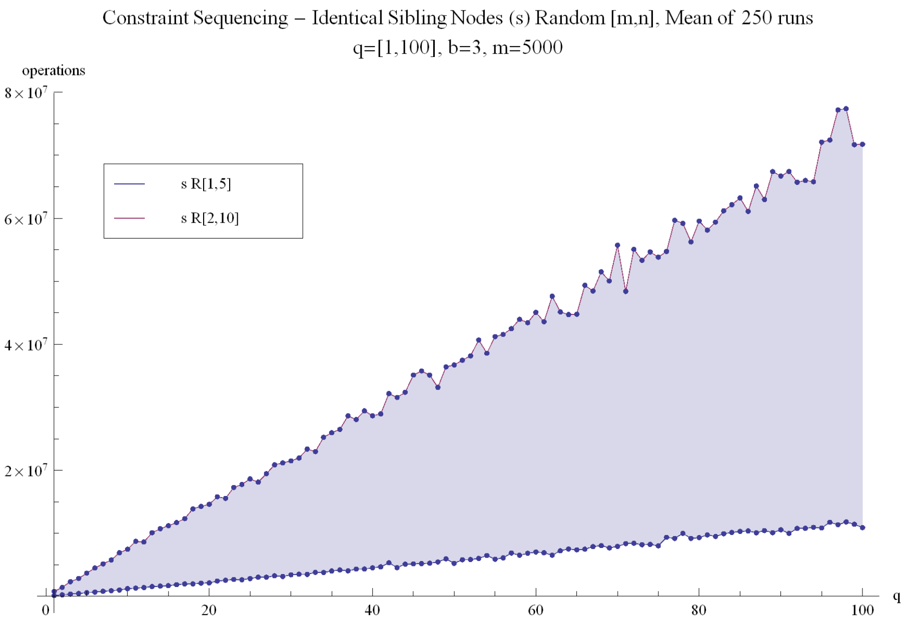


Figure 7.30: Constraint Sequencing, random identical sibling nodes (larger query) - 250 runs

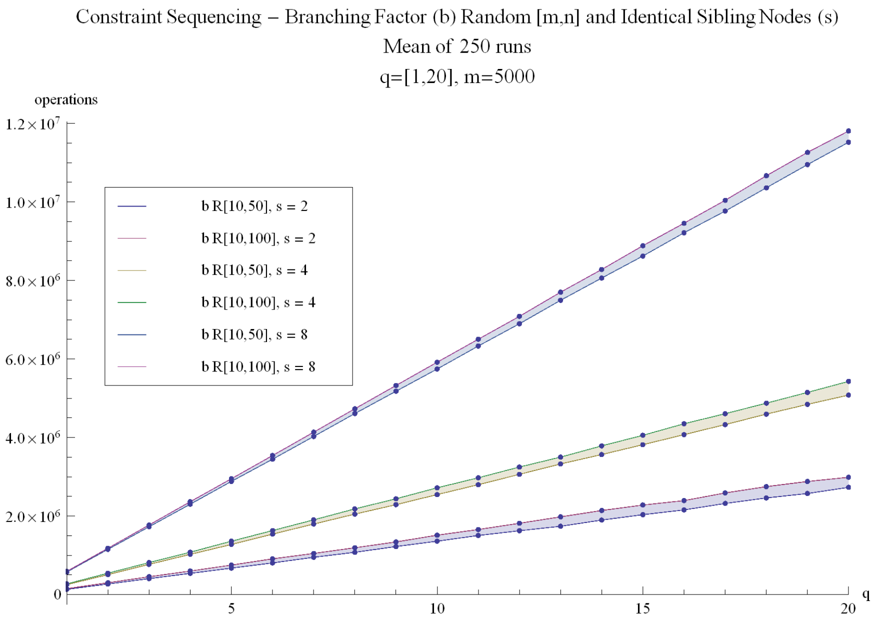


Figure 7.31: Constraint Sequencing, random branching factor and constant identical sibling nodes - 250 runs

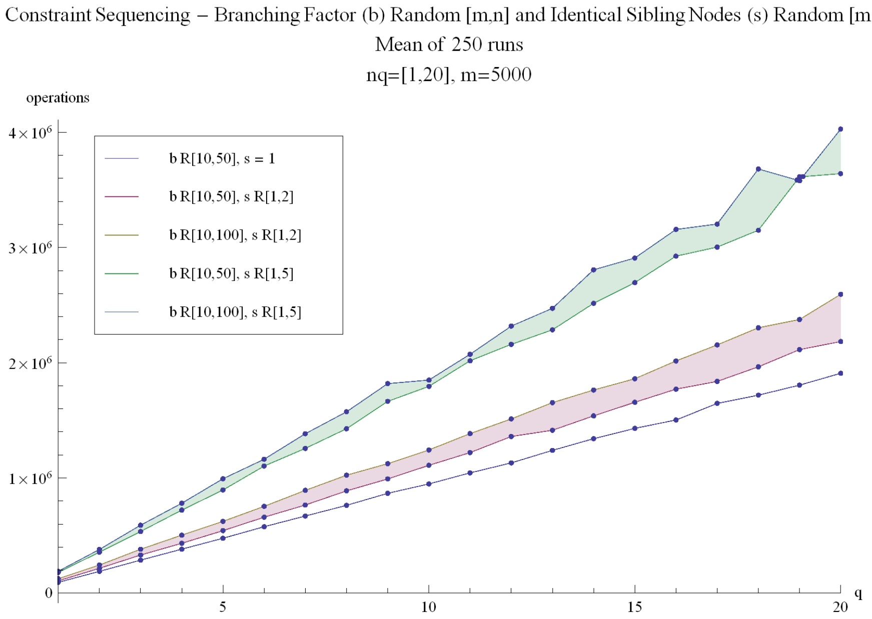


Figure 7.32: Constraint Sequencing, random branching factor and identical sibling nodes (with baseline) - 250 runs

on the bottom represents a baseline of $s = 1$ (constant). Note how an increase in s by four (not a factor of four, but just four) results in more than double the amount of operations from the baseline. This further substantiates our claim that s is a driving force when considering the query performance of Constraint Sequencing.

When we compared s to the document size m, we found that, for relatively small document sizes ($m < 10,000$), the document size produced a larger impact than the amount of identical sibling nodes. Figure 7.33 illustrates this point. The lower cluster corresponds to smaller document sizes ($m = 250$) while the upper cluster corresponds to a comparatively larger document size ($m = 3000$). However, when document sizes grow large, an interesting effect is observed (shown in Figure 7.34). While similar to Figure 7.33, of particular importance is the area shaded in yellow between the lines corresponding to $s = 6, m = 3000$ and $s = 3, m = 10,000$. In this area, a large document ($m = 1000$) with a small amount of identical sibling nodes ($s = 2$) yields better query performance than a small document with a large amount of identical sibling nodes. This reinforces our earlier observation that, as document size grows, the effect of s is more pronounced. With larger XML documents, the presence or absence of identical sibling nodes has a substantial impact on query

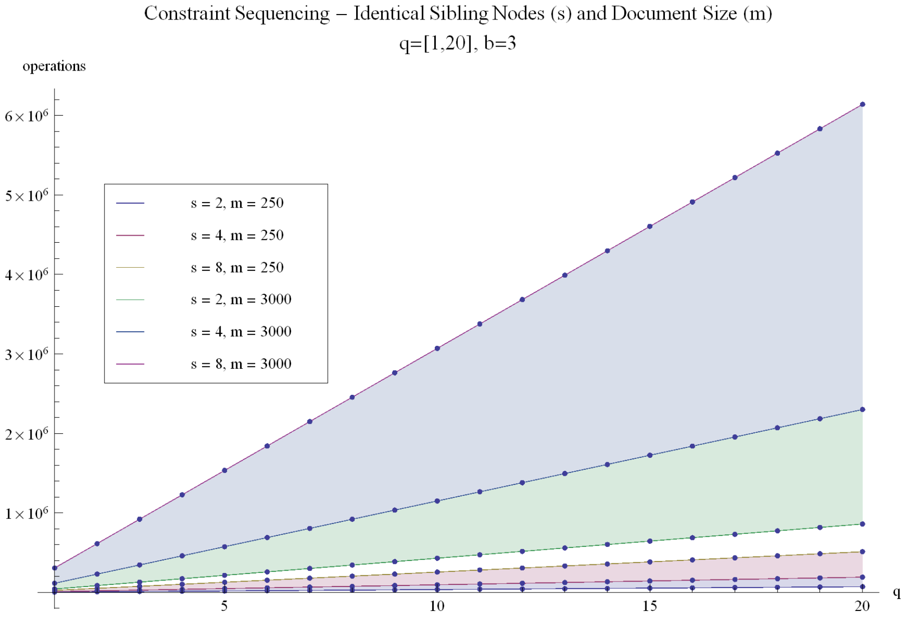


Figure 7.33: Constraint Sequencing, various document sizes (small) and identical sibling nodes (small)

performance.

7.2.4 Summary of Effects by Constraint Sequencing Parameters

The experimental results presented show that three parameters in Table 7.2 contribute to the overall performance of the algorithm. First, the number of query nodes q directly affects the performance of Constraint Sequencing in a linear manner. As query size grows, the number of operations required by Constraint Sequencing to answer the query increase. All of the results presented show that an increase in q always results in increased operations for Constraint Sequencing. Second, the effect of the document size m is a major contributing factor. When the number of identical sibling nodes is fixed, a larger document size results in an explosion of operations for Constraint Sequencing. Third, the frequency (amount) of identical sibling nodes s is another major factor in Constraint Sequencing. For smaller document sizes, the effect of an increased value for s is not as substantial as the effect of the document size itself. However, for large XML documents, the number of identical sibling nodes dramatically impacts query performance. In other words, as document size grows, the additional operations incurred by the presence of identical sibling nodes increases by an amount

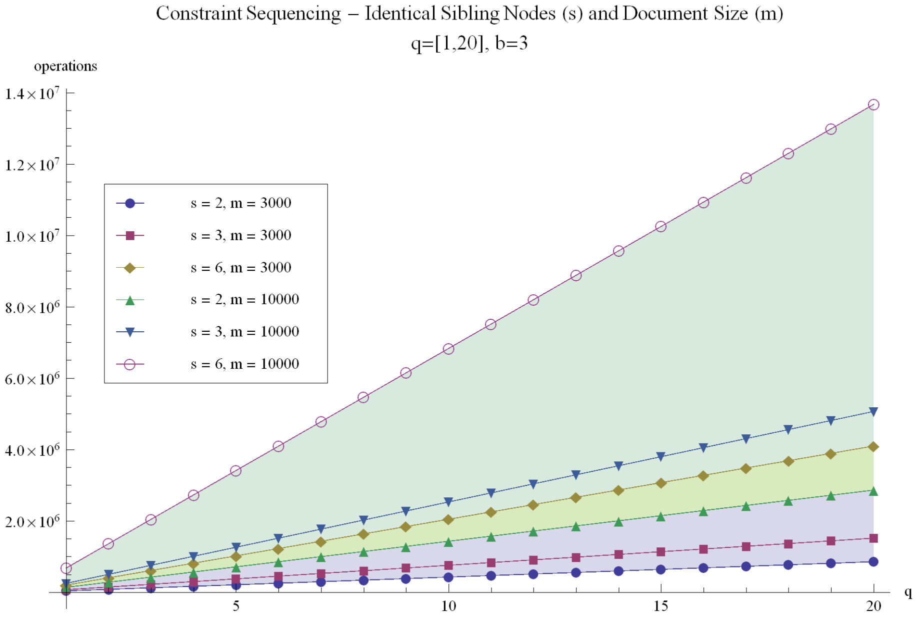


Figure 7.34: Constraint Sequencing, various document sizes (large) and identical sibling nodes (small)

that grows as s grows.

TwigStack and Constraint Sequencing represent two leading techniques for querying XML data in its native form. In order to better analyze their performance and propose a framework for efficient query processing, it is necessary to investigate non-native techniques as well. Chapter 8 presents the results of performance studies, similar to those in this chapter, for the SS-Join algorithm and our own technique, RDBQuery. Chapter 9 compares the two native techniques with each other, and the native technique that outperforms the other is compared to RDBQuery in Chapter 10.

Chapter 8

Analysis of Individual RDB Techniques

In a method similar to the techniques used in Chapter 7, this chapter considers two non-native techniques used to store and query XML data. Both techniques, SS-Join and RDBQuery, utilize a relational database management system (RDBMS) to store XML data and perform queries. SS-Join utilizes the relational cursor interface while RDBQuery uses standard relational operations, specifically the selection (σ) operator.

8.1 SS-Join

In Chapter 5, we present Equation 5.14 and Equation 5.15 based on the work by Shui et al. [SLFW05]. Table 8.1 presents the various parameters for the SS-Join algorithm that are used in Equation 8.1. This equation is repeated from Chapter 5, Equation 5.15. Refer to Chapter 5 for detailed explanations of the salient parameters listed in the table.

$$2|dSize||aSize| \times (\log_2(2^k - 2^{k-1}) \times (\log_2(dSize - dPos) + \log_2(aSize - aPos))) \tag{8.1}$$

Using the parameters in Table 8.1, we ran mathematical experiments to investigate the effects of the various parameters on overall query performance. The results of those experiments are

Parameter	*Name*	*Represents*
dSize	Descendant list	Quantity of nodes in descendant list
aSize	Ancestor list	Quantity of nodes in ancestor list
dPos	Descendant cursor position	Position of descendant cursor in relation
aPos	Ancestor cursor position	Position of ancestor cursor in relation
k	Breadth parameter	Used to define search space when skipping

Table 8.1: Parameters in the SS-Join Algorithm

discussed here with additional supporting material found in Appendix C. For some parameters, we use Equation 5.14 which represents the worst case scenario for the algorithm. This allows us to compare how parameters interact with each other with a simplified equation. However, when necessary, we utilize the more general form shown in Equation 8.1. By observing the behavior of SS-Join as given in the referenced equations, we postulate the following about its performance:

1. As the length of *dSize* and *aSize* increase, query performance will degrade.
2. The breadth parameter *k* will have a major impact on query performance since it controls the amount of skipping through the *Edge* relation (table).
3. The cursor positions *dPos* and *aPos* will have less of an impact on query performance than other parameters.
4. The relative sizes of *dSize* and *aSize* will affect the execution operations of SS-Join.

We make some general observations about XML documents that are shredded into RDB relations to narrow the search space for our study. The number of entries in the *Edge* table corresponds to the number of nodes in the XML document. In addition, we do not utilize the *Path* table that is proposed as an extension to the SS-Join algorithm by Shui et al. [SLFW05]. We also assume a constant (linear) time involved to move the relational cursor by a set amount g. Although SS-Join is divided into two separate algorithms (an ancestor and descendant portion, as shown in Section 5.3.1), we present the results for both algorithms. The other techniques, both native and non-native, consider the entire query and not the ancestor or descendant portion. For simplicity of notation, we refer to SS-Join as the entire algorithm (both SS Ancestor Join and SS Descendant Join).

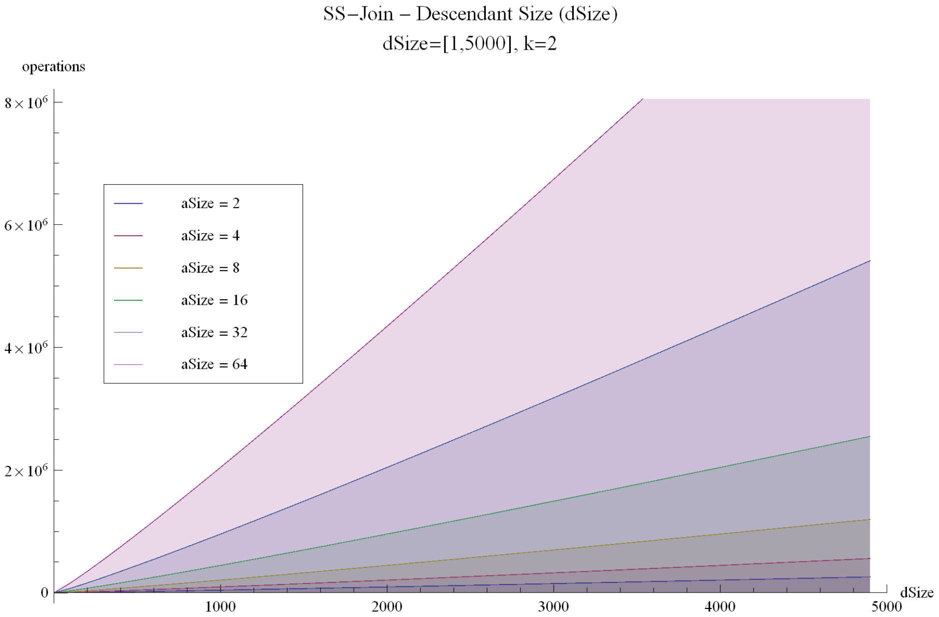


Figure 8.1: SS-Join, various descendant list sizes

8.1.1 Effect of *aSize* and *dSize*

The terms *aSize* and *dSize* refer to the size of the ancestor and descendant lists, respectively, that are passed to the SS-Join algorithm shown in Figure 5.1. These lists control the space where SS-Join must execute and search for answers to a given query. Our initial analysis focuses on a single parameter, *dSize*, and allows *aSize* to assume arbitrary values. The results are identical if the terms are reversed, and this behavior is observed in Equation 5.14. Figure 8.1 shows the results of SS-Join when *dSize* is varied from 1 to 5000 with an increasing *aSize* of $\log_2 n$. We follow the same labeling conventions as discussed in Section 7.1.1 for the graphs presented in this chapter. Of particular importance is how quickly the number of operations increases for a small value of *aSize* relative to *dSize*. Upon closer inspection, we observe that when *aSize* doubles, the number of operations also roughly doubles. The same observation could be made if the roles of *dSize* and *aSize* were reversed. For this experiment, the sizes of the two lists are intentionally disparate. Figure C.1 illustrates similar results with larger sizes for both lists. In order to compare with future experiments, note the scale on Figure C.1. SS-Join requires approximately 1.2×10^9 operations with *aSize* and *dSize* both equal to 5000. This also assumes that *aPos* and *dPos* are zero, and it illustrates a worst case scenario for SS-Join with the given parameters.

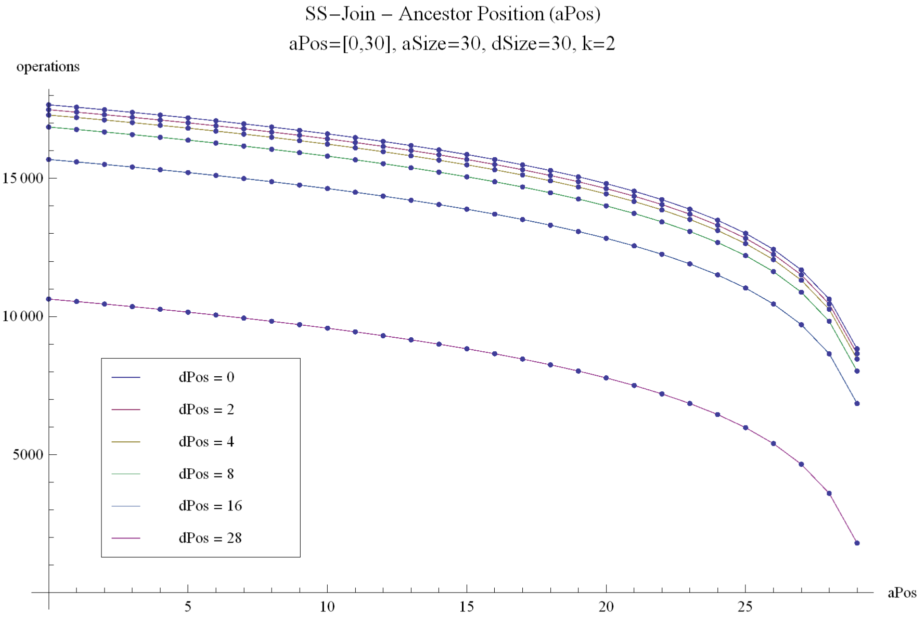


Figure 8.2: SS-Join, $aPos/dPos$ increasing (small lists)

8.1.2 Effect of $aPos$ and $dPos$

The moving cursors $aPos$ and $dPos$ are what allow SS-Join to skip past records that do not contribute to the final answer of a given query. In the worst case (Equation 5.14), SS-Join is unable to skip any records and both $aPos$ and $dPos$ are fixed at zero. Ignoring the effect of these cursors diminishes the performance of SS-Join, so we designed experiments that study the effect of moving the cursors throughout the two lists. As in Section 8.1.1, the results presented for $aPos$ and $dPos$ mirror the results if the values of the terms were reversed. In other words, $aPos$ and $dPos$ contribute the same amount and contribute in the same way to the overall complexity of SS-Join. This section utilizes the general form of SS-Join shown in Equation 8.1. In Figure 8.2, we begin with low sizes for the ancestor and descendant lists in order to observe a trend. As the position of the cursor increases, the number of operations decrease. This is to be expected from the SS-Join algorithm. If the $aPos$ cursor is moved further down the list, SS-Join need not examine the records in the list that occur before that cursor. This effectively narrows the search space for the algorithm. In Section 5.3.2, this is termed *index-free skipping* by the authors [SLFW05]. The results shown in Figure 8.2 mirror the results presented by Shui et al. in their original work [SLFW05], and this helps to substantiate the validity of our SS-Join model.

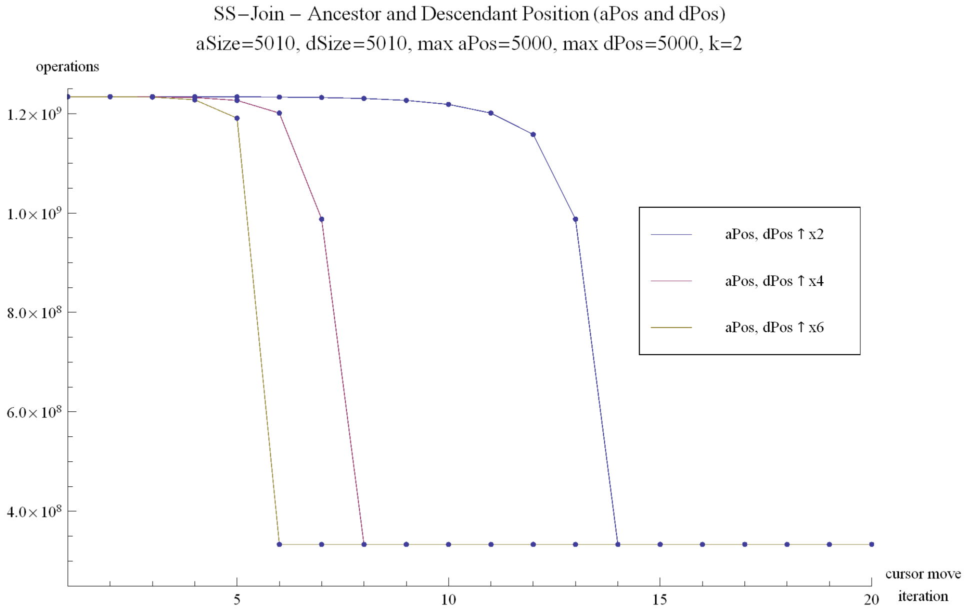


Figure 8.3: SS-Join, $aPos/dPos$ increasing

While the results shown in Figure 8.2 build intuition about SS-Join and confidence in our model, experiments with larger datasets are required to provide insight on how the algorithm performs in practice. Initially, we allow the size of $dSize$ and $aSize$ to be the same. Figure 8.3 shows execution of the SS-Join algorithm as we increase the cursors $aPos$ and $dPos$ by a constant factor (shown in the legend). The x-axis shows the iteration of the cursor as it advances. For example, at $aPos = 5$ on the x-axis, the blue line has moved both $aPos$ and $dPos$ forward by a factor of two for five times. Each iteration represents a doubling of the previous cursor position. Therefore, at position five, the cursor $aPos$ has increased from position 1 to position, or 2^{i-1} where i represents the iteration number. In general, if a line is increasing by a constant factor, the increase at position i will be f^{i-1} where f represents the factor. The behavior of the cursor is such that it cannot move past the size of the list. The yellow line corresponds to a factor of six, so at $i = 5$, the cursor $aPos$ is at position 1296 in the list $aSize$. Since the cursor cannot advance past the size of the list, we force the cursor to stop at a maximum position of the list size.

Upon initial inspection of Figure 8.3, it appears that increasing cursor positions has a dramatic impact on query performance. What is more interesting is that there is a limit, indicated by the yellow line parallel to the x-axis, to the benefits of an increased cursor position. The further SS-Join

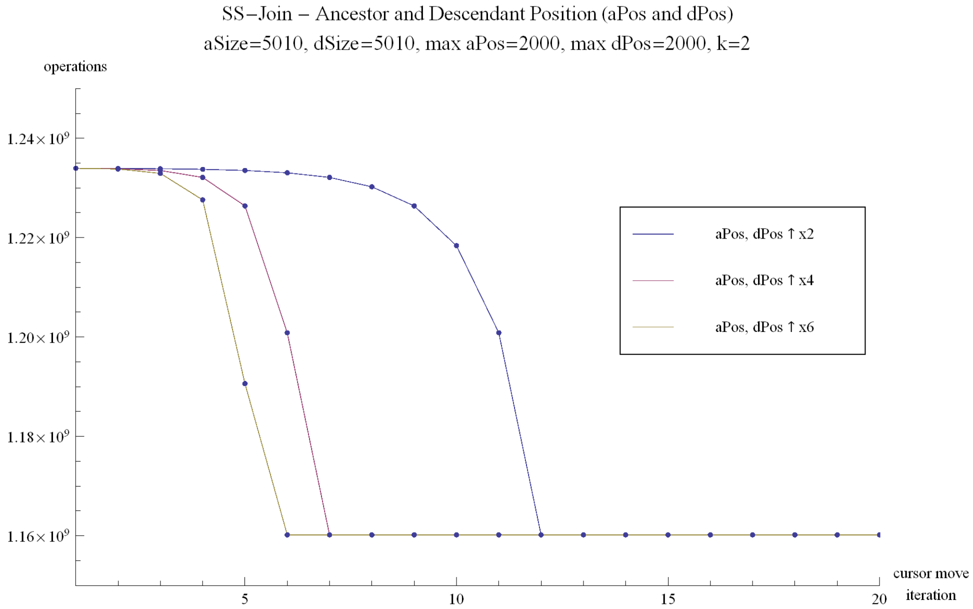


Figure 8.4: SS-Join, $aPos/dPos$ increasing (lower maximum position)

is able to advance the cursor, the better the performance of the algorithm. Figure 8.4 shows the results of a similar experiment, except the two cursors $aPos$ and $dPos$ were not allowed to advance past the midpoint of the list size. The performance gain lines still trend in the same direction, but the performance gains (reflected in the y-axis scale) are not as high. Figures C.2 and C.3 in Appendix C illustrate similar behavior for SS-Join but with cursor positions that increase by a constant amount rather than by a factor. As expected, gains in overall query performance are not as pronounced for queries that move the cursors a shorter distance or at a slower rate.

As mentioned in the introduction to this chapter, the relative sizes of $aSize$ and $dSize$ play an important role in the performance of SS-Join. If $aSize = dSize = c$, the algorithm takes longer to execute than in the case where $aSize + b = dSize - b = c$. Larger values of b result in increased performance gains for SS-Join. This behavior is shown in Figure 8.5. With $aSize$ allowed to be disproportionately large when compared with $dSize$, we observe a substantial performance gain when compared with the values illustrated in Figure 8.3. Note that while the effects of $aPos$ and $dPos$ remain similar to those previously observed, the y-axis scale is reduced by a factor of 250 (from $9.0{\times}10^8$ to $3.5{\times}10^6$). A slightly less exaggerated example is shown in Figure C.4. The performance gain is less that that shown in Figure 8.5, but there is a noticeable increase in

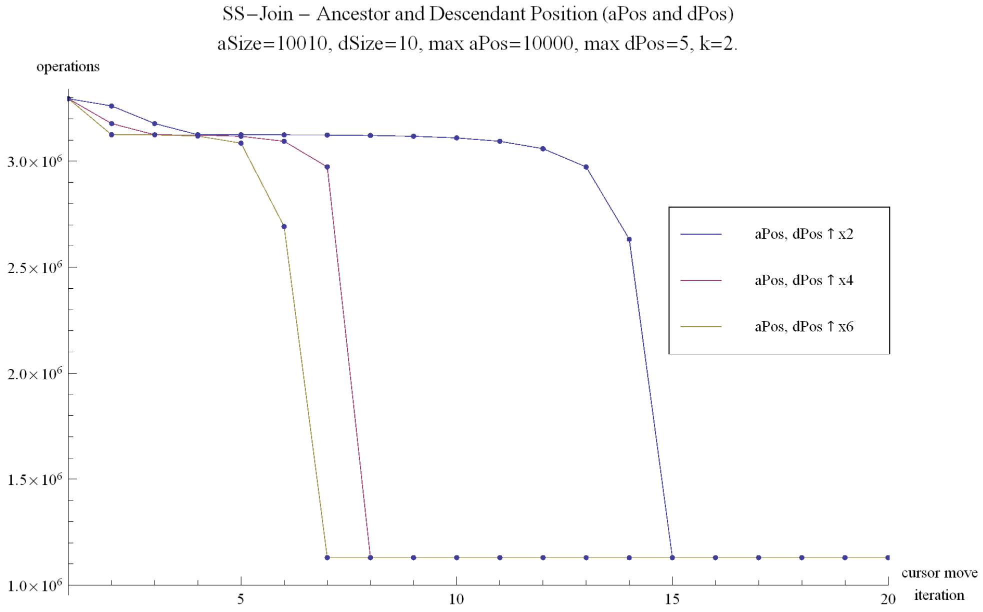


Figure 8.5: SS-Join, $aPos/dPos$ increasing (different size lists, high/low)

performance, especially as the cursors move through the lists.

We also explored the case where $aPos$ and $dPos$ increase in nonuniform (random) fashion. For these cases, single runs of SS-Join are illustrated in Figures C.5, C.6, C.7, and C.8 in Appendix C. Figure 8.6 shows the mean of 250 runs of the SS-Join algorithm with $aPos$ and $dPos$ increasing by a random amount within the range shown in the legend. We note that, for larger increases in $aPos$ and $dPos$, query performance increases. The bottom blue line shows a performance improvement of $2.0{\times}10^8$ after 10 cursor iterations if both $aPos$ and $dPos$ advance somewhere in the range from one to 5000. As we constrain the movement of the two cursors, the gains in query performance diminish. This is illustrated by the upper lines. Note how they cluster together with little improvement in SS-Join operations. Figure 8.7 demonstrates the increased performance gains as we continue to advance the two relational cursors. Since this behavior is also shown in Figure 8.6, we limit further analyses to a smaller number of cursor iterations to improve chart readability.

Figure 8.8 illustrates the effect of decreasing the range in which the cursors move. In this experiment, we place a higher minimum movement limit on the cursor while simultaneously decreasing the maximum range. Note that it is almost identical to Figure 8.6. The potential benefit of increasing the minimum advancement of the relational cursor is negated by taking the average

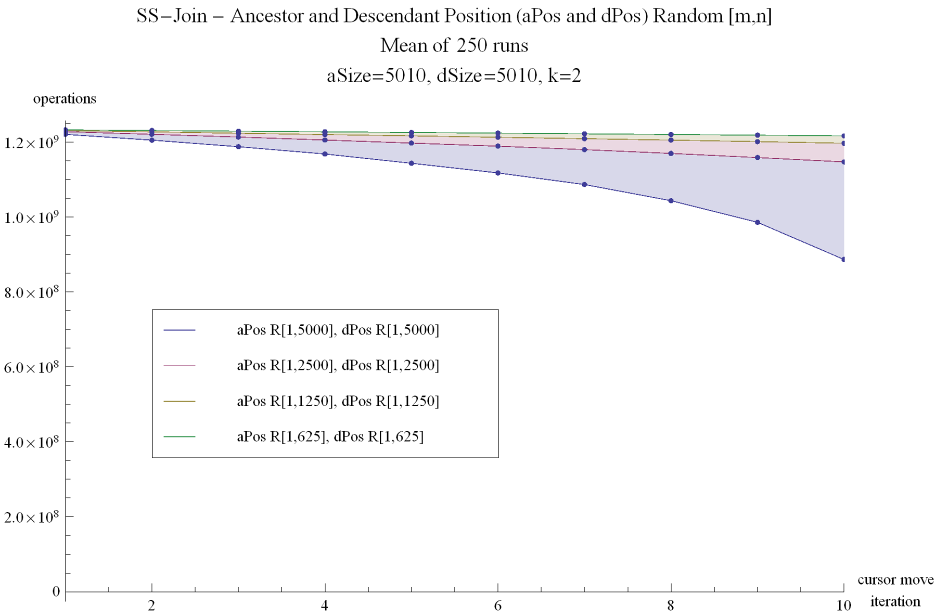


Figure 8.6: SS-Join, random increases to $aPos/dPos$ (large, identical ranges) - 250 runs

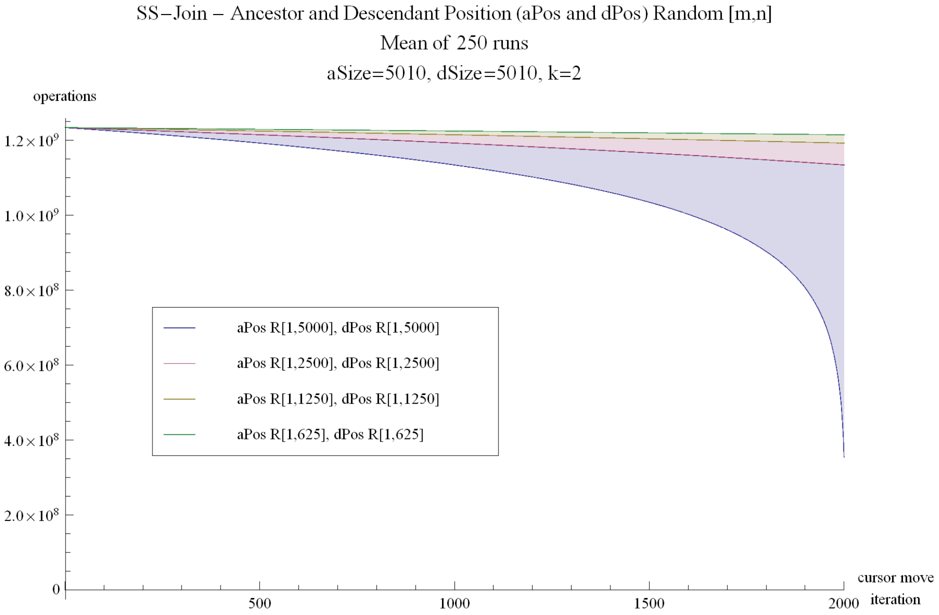


Figure 8.7: SS-Join, random increases (more iterations) to $aPos/dPos$ (large, identical ranges) - 250 runs

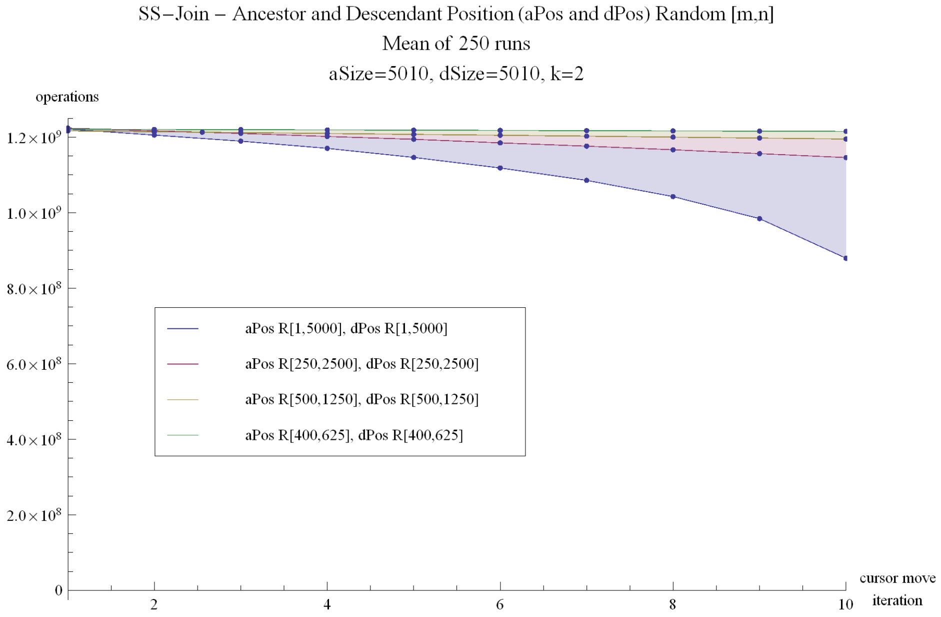


Figure 8.8: SS-Join, random increases to $aPos/dPos$ (narrowing, identical ranges) - 250 runs

across 250 runs. While some benefit may be experienced in such a case, the potential performance gains will average out over time as queries are processed. This is especially true of the database is volatile, resulting in different advancements of both cursors as new data is added or existing data removed. Figure 8.9 tells a similar story but with different parameters. In this experiment, we fixed the range for $aPos$ while increasing the range for $dPos$. Observe that when one range is relatively small (as is that for $aPos$), increasing the range of the other cursor does not have a significant impact on query performance. Figure 8.10 magnifies this result by decreasing the size of the two lists, $aSize$ and $dSize$, while fixing the range for $aPos$ relatively low and increasing the range for $dPos$. Observe the marginal performance gains exhibited by the top three lines. It is not until we reach the case where $dPos$ is allowed to advance from 80 to 990 (the bottom green line) where we notice a steady increase in SS-Join performance. Figure 8.11 is simply an enlargement of the interesting area from Figure 8.10. It is easier to observe the increased performance seen in the case where the $dPos$ cursor is allowed to advance over a larger range.

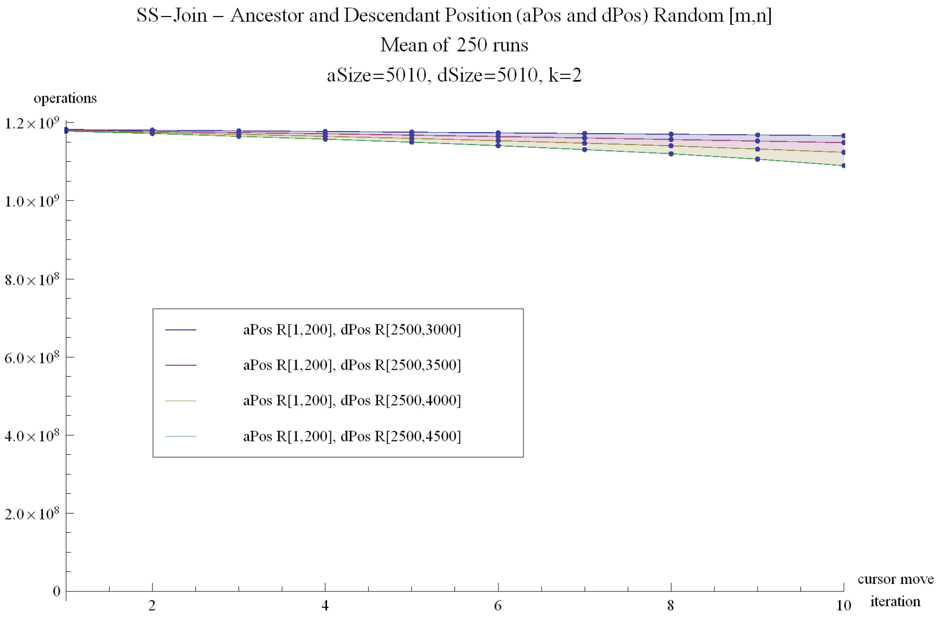


Figure 8.9: SS-Join, random increases to $aPos/dPos$ (one range fixed) - 250 runs

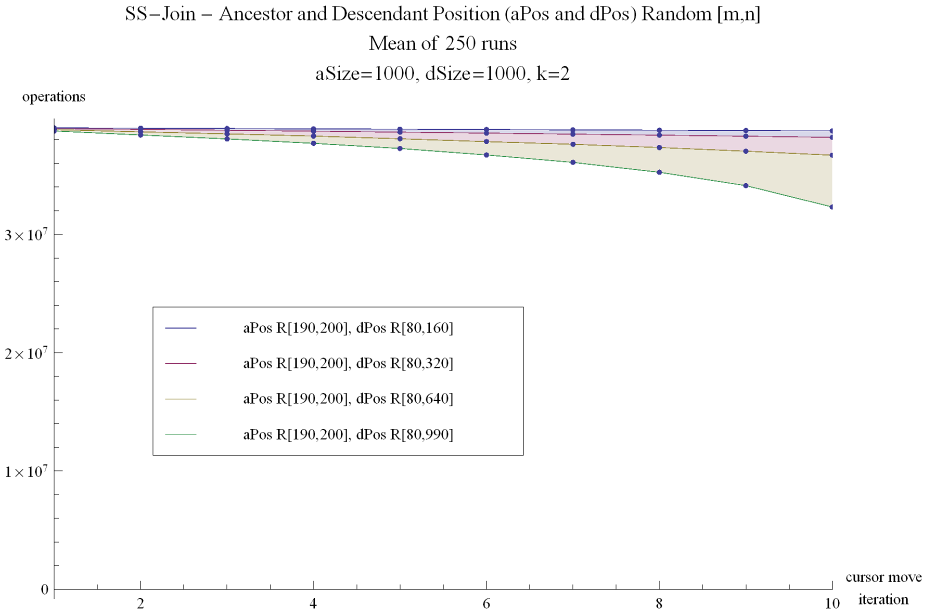


Figure 8.10: SS-Join, random increases to $aPos/dPos$ (one range fixed small) - 250 runs

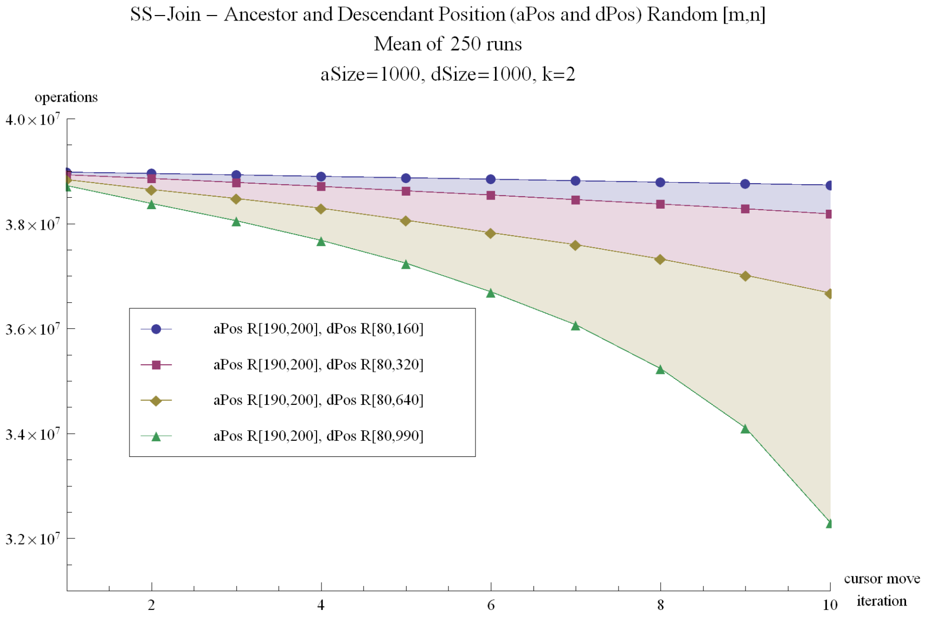


Figure 8.11: SS-Join, random increases to $aPos/dPos$ (one range fixed small) - 250 runs (Zoom)

8.1.3 Effect of k

One of the advantages of SS-Join over other algorithms is its ability to skip records that do not contribute to the final solution. It accomplishes this by moving the cursors $aPos$ and $dPos$ through the $Edge$ table. As the algorithm continues to skip, the skipping amount will trend towards increasing. The first skip will be smaller than the second skip if the second skip occurs directly after the first. In the worst case, SS-Join moves the cursor past the data it needs. It then must perform a binary search in the space $2^k - 2^{k-1}$. In the perfect case, the skip moves the cursor directly to the record it needs. In that case, k is one, and the value for $\log_2(2^1 - 2^0)$ becomes zero. This means that, if we happen to move the cursor directly to the correct node, we only incur a single cursor move (1 operation). This is not an interesting case since it always results in the same answer. In this section, we explore the effects of over-skipping the intended node by various sizes of k. After an XML document is shredded into a relation, it loses some of its native characteristics. While depth is somewhat reflected by the number of records, the XML document is flattened and breadth also factors into the size of the relation and number of records. For the purposes of our analysis, we allow k to reflect the breadth of the original XML tree. Recall from Chapter 5 that k refers to that amount of records that SS-Join skips to quickly move through the $Edge$ table. The purpose

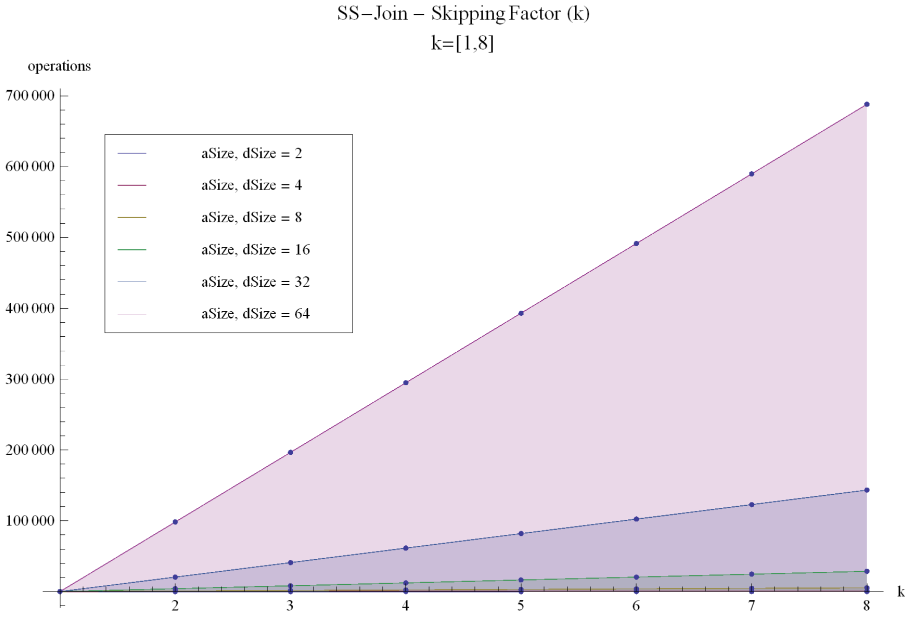


Figure 8.12: SS-Join, skipping factor increasing (small lists)

of this is to skip all entries in the ancestor and descendant lists that do not contribute to the final query result set. As a simple example, refer back to Figure 3.1 and consider a query that is only interested in `author` nodes with a `name` of `Peppard`. The SS-Join algorithm could skip all of the nodes that are in the ancestor list of node 23 since they do not contribute to the final result. By skipping these nodes, we have moved across the breadth of the XML tree.

The size *aSize* and *dSize* play an important impact when examining the effect of k. When the lists are small, as seen in Figure 8.12, as k increases, query performance significantly degrades. A larger value of k means that SS-Join has over-skipped the intended record by a larger amount and thus must search a larger set of records for the intended target. When k is eight, SS-Join must search an area equal to $2^8 - 2^{8-1}$, or 128 nodes. Figure 8.12 assumes the SS-Join algorithm always over-skips by the given value of k. Figure C.9 shows a similar explosion in the number of operations performed by SS-Join with larger *aSize* and *dSize* values. It is important to note that, regardless of the size of the two lists, increasing k has the same result. While the total number of operations will increase with a larger *aSize* and *dSize* values, the behavior of Figures 8.12 and C.9 are the same.

Figure 8.13 shows the results of varying the relational cursors *aPos* and *dPos* through the

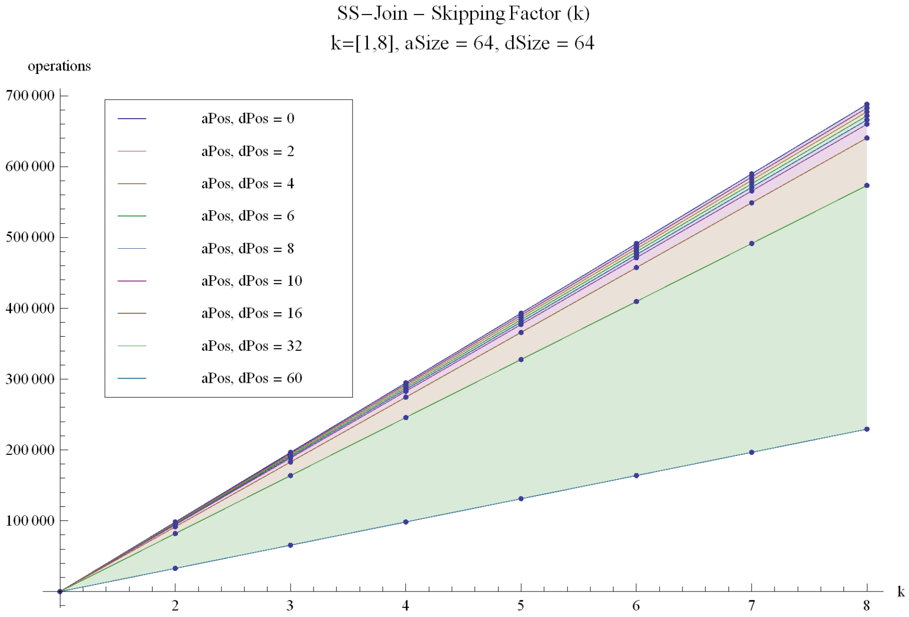


Figure 8.13: SS-Join, skipping factor and $aPos/dPos$ increasing (small lists)

table for different values of k. Note that as the cursors progress through the lists, the effect of k is diminished by varying amounts. Until the cursors reach half way through, there are nominal gains in SS-Join performance. However, the jump from half way through the list to near the end is dramatic (illustrated by the large green slice in the graph). When we consider the SS-Join algorithm, this type of behavior is justified. As the relational cursors progress through to the end of their respective lists, the algorithm is constrained from skipping in larger amounts. Therefore, as $dPos$ and $aPos$ increase, SS-Join cannot move the cursors forward by the same amount as when the algorithm initiated. Figures C.10, C.11, and C.12 in Appendix C demonstrate similar behavior. When the relational cursors pass the halfway mark in their respective lists, SS-Join begins to experience performance gains. These gains are marginally larger for increasing values of k. As a final experiment, we allow both $aSize$ and $dSize$ to increase in size to 5000 each. Figure 8.14 shows the results of SS-Join for $dPos$ progressing through the first half of the descendant list. Of importance is the marginal performance improvements regardless of the descendant list cursor. In Figure 8.15, $dPos$ moves through the second (last) half of the descendant list. Compare these results to those from the first half of the descendant list. Of particular importance is the large performance gains experienced when $dPos$ is close to the end of the list (close in value to $dSize$).

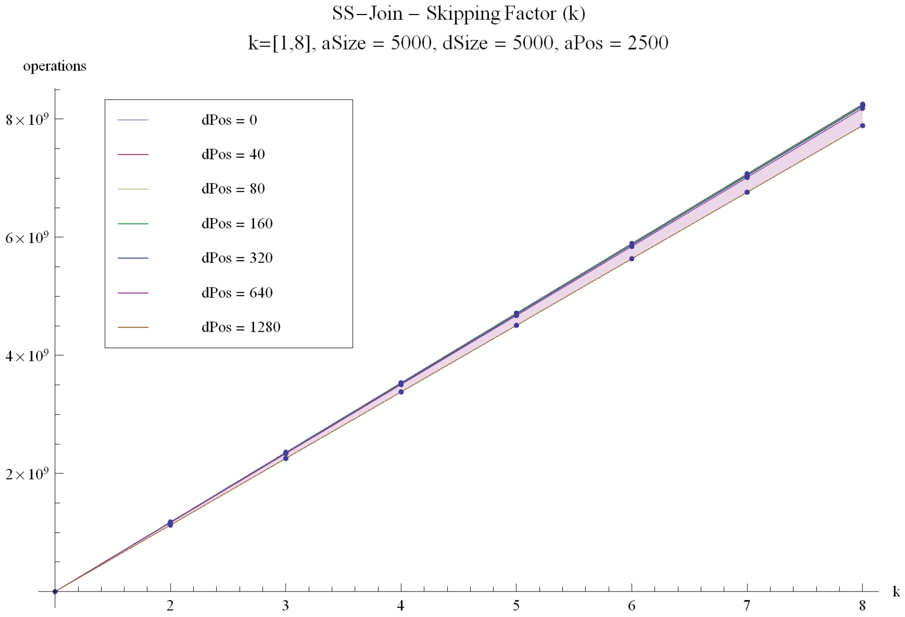


Figure 8.14: SS-Join, skipping factor increasing, *aPos* fixed, *dPos* moving through first half of list

This mirrors our previous observations about the effect of the relational cursor position when k increases.

8.1.4 Summary of Effects by SS-Join Parameters

The experimental results presented show that each of the parameters in Table 8.1 contribute to the overall performance of the SS-Join algorithm. As the size of the two lists, ancestor and descendant, grows, SS-Join must process more records (entries in the *Edge* table). Performance can further degrade if the algorithm skips a large section of the table (reflected by a large k value) and must backtrack and search inside that skipped region for the record of interest to continue processing. With regard to the relational cursor positions $aPos$ and $dPos$, our results consistently show that larger cursor positions (where the cursor is in the last half of the list) result in performance gains for SS-Join. The magnitude of these gains depends not only on the size of the list but also the number of consecutive times we move the cursor through the list. If SS-Join is able to move the cursor in increasing increments without over-skipping the necessary record, it can quickly process a lengthy list. While the effects of the relational cursor positions are not as profound as the effects of other parameters in the SS-Join algorithm, they still play a significant role in query performance

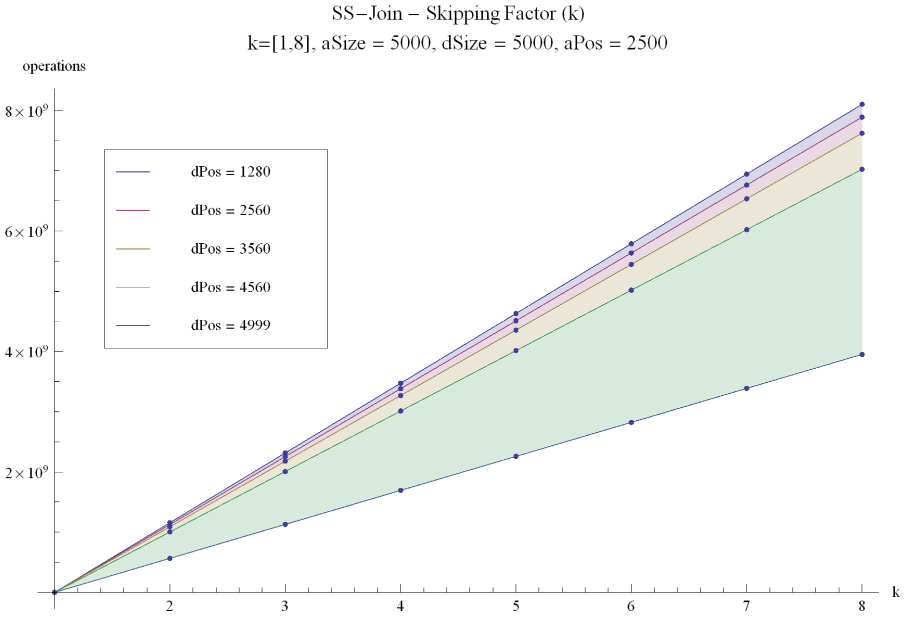


Figure 8.15: SS-Join, skipping factor increasing, *aPos* fixed, *dPos* moving through second half of list

and must be considered when analyzing SS-Join execution in all but the worst case.

8.2 RDBQuery

In Chapter 6, we present a new algorithm named RDBQuery to process XML data stored as a shredded relation in a relational database. Equation 8.3 is the result of our algorithmic analysis on RDBQuery (repeated for convenience from Equation 6.6), and Table 8.2 presents the various parameters used in the equation. Chapter 6 illustrates the salient parameters in Table 8.2 on a small example.

$$\phi_d\sigma_d + \phi_c\sigma_c \tag{8.2}$$

$$\phi_d\left[3x + \left\lceil\left(\frac{r}{d \times bfr}\right)\right\rceil + \frac{b + b_{I1}}{2} + \frac{r}{2}\right] + \phi_c\left[2x + \left\lceil\left(\frac{r}{d \times bfr}\right)\right\rceil + \frac{b_{I1}}{2} + \frac{r}{2}\right] \tag{8.3}$$

Using the parameters in Table 8.2, we ran mathematical experiments to investigate the effects of the parameters on overall query performance. The results of those experiments are discussed here

Parameter	*Name*	*Represents*
ϕ_d	Descendant edges	Number of descendant edges in the query
ϕ_c	Child edges	Number of child edges in the query
r	Records	Number of records in relation
d	Distinct values	Number of distinct values in relation
x	Index levels	Number of levels in the index
bfr	Blocking factor	Number of records in a block
b	Index blocks	Number of index blocks needed
b_{I1}	First level index blocks	Number of first level index blocks needed

Table 8.2: Parameters in the RDBQuery Algorithm

with additional supporting material found in Appendix D. Before we began the experiments, we made some general observations about the algorithm that allow us to significantly narrow our test space. When compared to the other parameters, the effects of b and b_{I1} can be ignored, especially for relations with any significant size. In addition, the blocking factor bfr typically remains constant and can be excluded from the analysis. The term x that represents the number of index levels can also be ignored. When compared to the effects of d, r, ϕ_d, and ϕ_c, it does not make a significant enough contribution to query performance to necessitate a full study. One parameter that is not listed in the table is the tuning parameter, also known as selectivity. In Equation 8.3, this is represented by the value $r/2$. As the denominator of this term changes from two and increases to other values, selectivity increases. The number of records RDBQuery selects from the *Edge* table decreases as it becomes more selective. By observing the behavior of RDBQuery, we postulate the following about its performance:

1. The effect of ϕ_d and ϕ_c will be similar if not the same.

2. Record size r will be the dominating parameter for the algorithm.

3. The number of distinct values d will have a significant impact up to a point.

4. The selectivity of records denoted by r/n will have a significant effect on query performance. As selectivity increases (n increases), RDBQuery performance will also increase.

Since RDBQuery uses the same relational database setup as SS-Join, the same observations made in Section 8.2 about XML documents that are shred into RDB relations also apply to this technique.

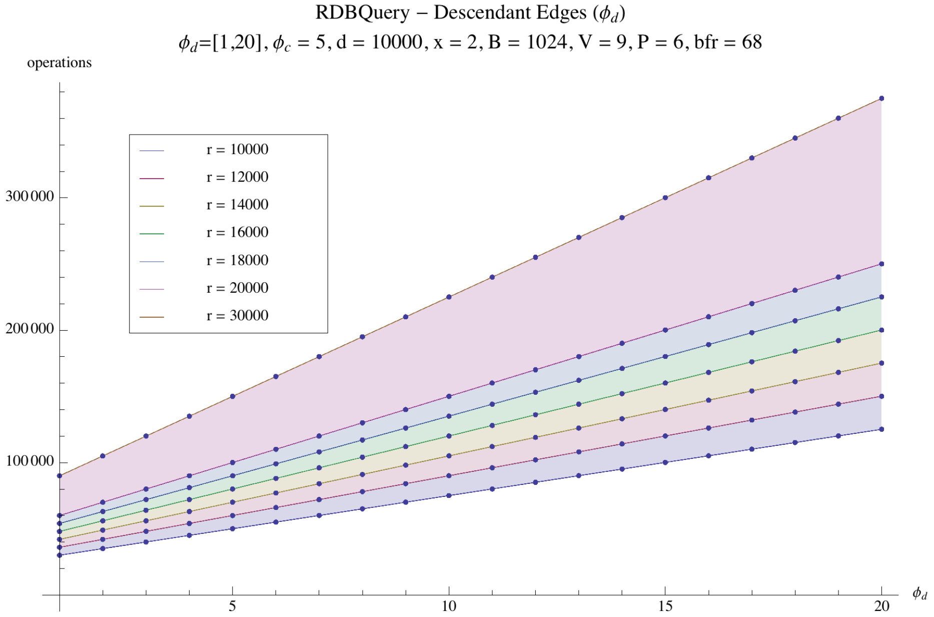


Figure 8.16: RDBQuery, record size increasing

In this case, we assume that the XML document is appropriate shredded into a single relational (the *Edge* table) as outlined in Section 5.2.1. In addition, the blocking factor *bfr* is calculated as $\lfloor B/R_i \rfloor$ where B is the block size and R_i is the sum of V, the file size, and P, the block pointer size [EN00].

8.2.1 Effect of r, ϕ_d, and ϕ_c

The term r refers to the size of the *Edge* table as represented by a single relation. As the number of records increase, the time necessary for RDBQuery to execute also increases. Figure 8.16 illustrates several runs of RDBQuery with various record sizes. Along the x-axis is the number of descendant edges (ϕ_d), and the number of child edges (ϕ_c) remains fixed at five. The lines correspond to tables with various number of records. As the number of records increases, the effect of increasing the query size ϕ_d is increased in a linear fashion. A similar result is achieved with a smaller number of records (Figure D.1). We also examine the case where there are no child nodes in the query, and this result is shown in Figure D.2, and the same trend from Figure 8.16 is observed. We know from the RDBQuery equation that the number of records will impact query performance, but we need to know how much of a factor record size is when the number of query edges (ψ_d and ψ_c) vary.

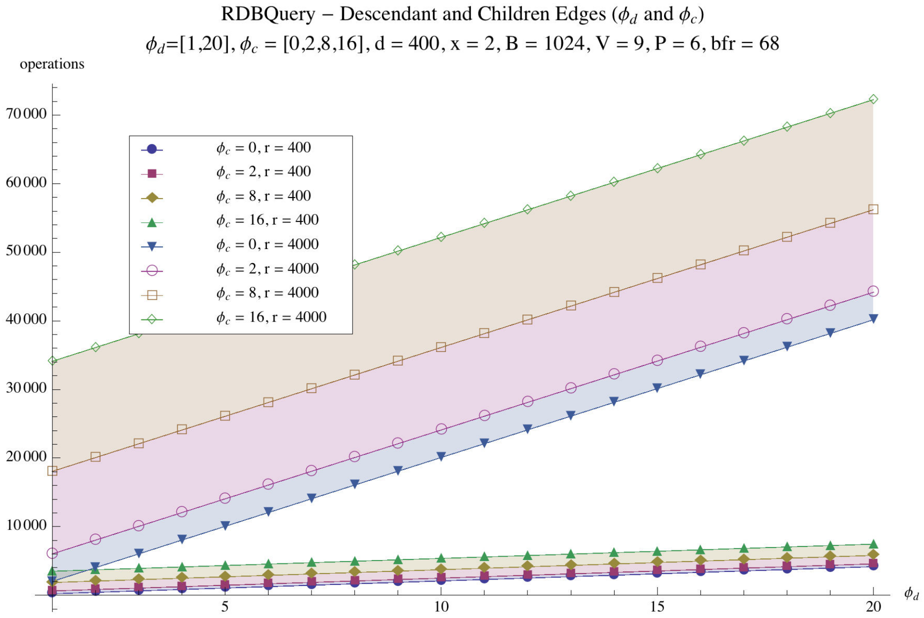


Figure 8.17: RDBQuery, descendant/child edges increasing

In order to better analyze the performance of RDBQuery, we designed an experiment that varies the record size, number of descendant edges, and number of child edges. Figure 8.17 illustrates the results of RDBQuery under these constraints. The larger shaded region that increases quickly represents a record size of 4000 while the lower, thinner region represents a record size of 400. As the number of edges in the query increase, the performance of RDBQuery degrades more quickly with a larger record size. It is not immediately clear why this should be the case, but upon closer inspection of the RDBQuery equation (Equation 8.3), we notice that the factor $r/2$ represents a tuning parameter (selectivity) in the relation. In short, we are allowing RDBQuery to operate with 50% selectivity (a high selectivity ratio). This led us to design a series of experiments that vary the selectivity parameter r/n and observe the impact on query performance.

To begin, we fixed the number of child edges and varied the selectivity from $r/2$ to $r/16$. Figure 8.18 shows that, as we increase the denominator from two to 16, selectivity increases (as r/n decreases), and RDBQuery finds fewer records in the *Edge* table that satisfy the complex selection statements shown in Algorithm 6.1, the RDBQuery algorithm. In other words, a value of $r/2$ represents a lower selectivity than $r/16$ since more records will be returned. The algorithm is being less selective. When selectivity increases, an increase in the number of query edges results in a

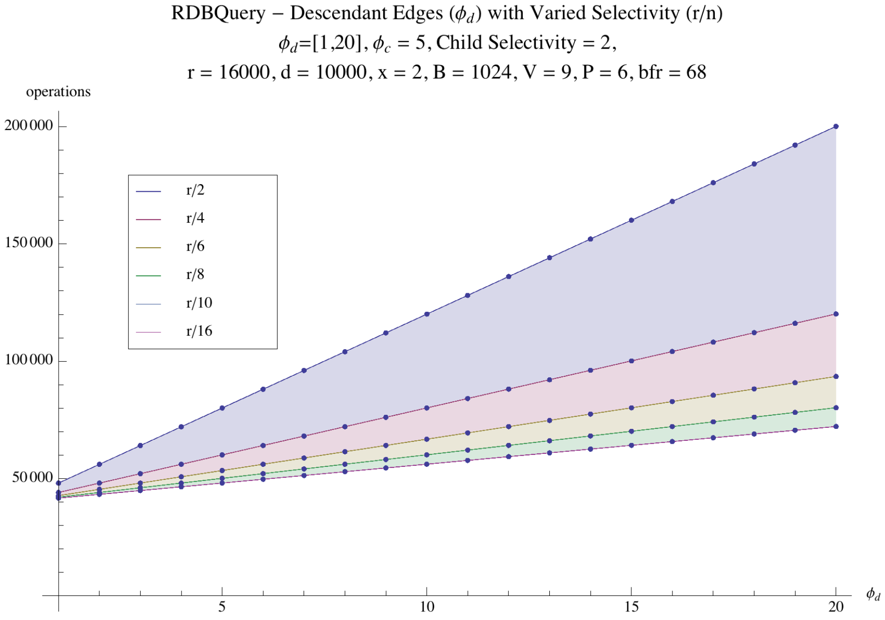


Figure 8.18: RDBQuery, selectivity increasing

smaller degradation in overall query performance. Compare the top line (blue, selectivity $r/2$) with the bottom line (red, selectivity $r/16$). The top line increases more quickly than the bottom for the same number of query edges. This is a factor of the selectivity.

A logical extension of this experiment is to allow the selectivity and number of all edges (ϕ_d and ϕ_c) to vary. Figures D.3, D.4, D.5, and D.6 represent the intermediate steps necessary to achieve the results shown in Figure 8.19. In this graph, we retain only the minimum and maximum selectivity and ϕ_c values. We observe the same behavior as seen in Figure 8.18. The selectivity r/n represents a larger contributing parameter to RDBQuery than the number of query edges. An additional method to study the effect of selectivity on query performance is to fix the number of query edges and vary the record size r. We allow selectivity to increase by a function r/n, where n increases by a factor. Figure 8.20 shows the result of RDBQuery run with a varying amount of records and $\phi_d = \phi_c = 32$. It is important to note that the greatest benefit to the algorithm is when selectivity increases from a factor of two to a factor of three, denoted by the top lines, respectively, in the graph. After that point, selectivity can increase by large amounts, but there appears to be little additional benefit to the overall execution of the algorithm. This leads us to conclude that, while selectivity is important for query performance, there is a point where higher

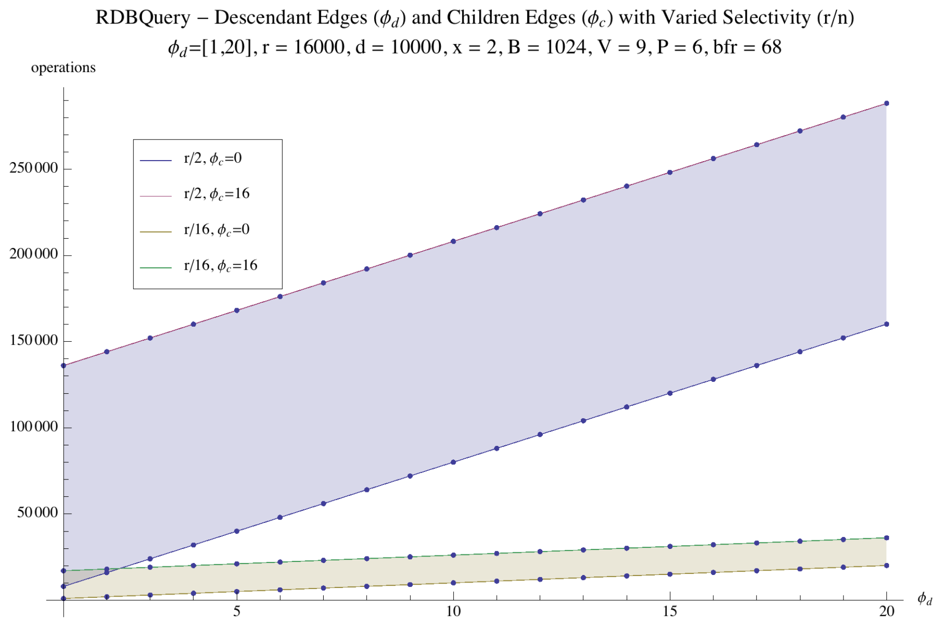


Figure 8.19: RDBQuery, selectivity and descendant/child edges increasing (min/max values shown)

selectivity results in minimal performance gains. A similar result is shown in Figure D.7 but for a smaller query size.

8.2.2 Effect of d

The term d reflects the number of distinct values in the *Edge* table for a shredded XML document. As the number of distinct values increases, query performance should also increase. However, there is a limit to those performance gains. As shown in Equation 8.3, d is involved in the term with r and bfr. The term, given as $\left\lceil\left(\frac{r}{d\times bfr}\right)\right\rceil$, states that as d increases, the value of the entire term will decrease. The ceiling function will prove to dampen the benefits of d slightly.

We begin our study of the parameter d in Figure 8.21 by fixing both the number of query edges (shown in the legend) and the selectivity ($r/10$). We notice an early drop in the number of operations followed by a seeming leveling-out of the plot. However, upon further investigation, we see that there is actually a slight decrease in the results. Figure 8.22 shows the same experiment but with a smaller range of distinct values. Here we clearly note the sharp decline at the beginning of the run and then a seemingly flat (constant) number of operations. The plots are actually slowly decreasing, but the amount is too small to observe on the graph.

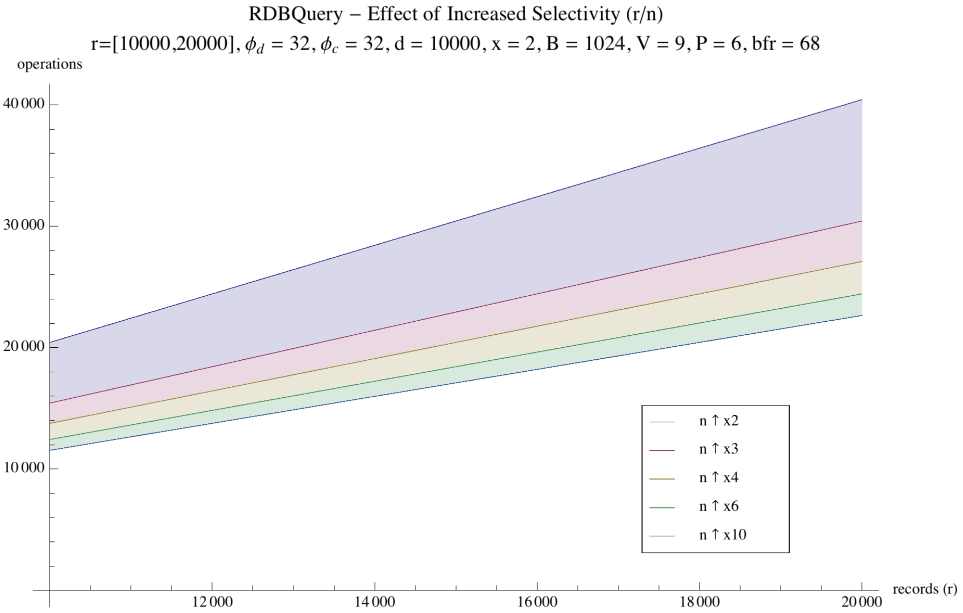


Figure 8.20: RDBQuery, selectivity increasing by constant factors

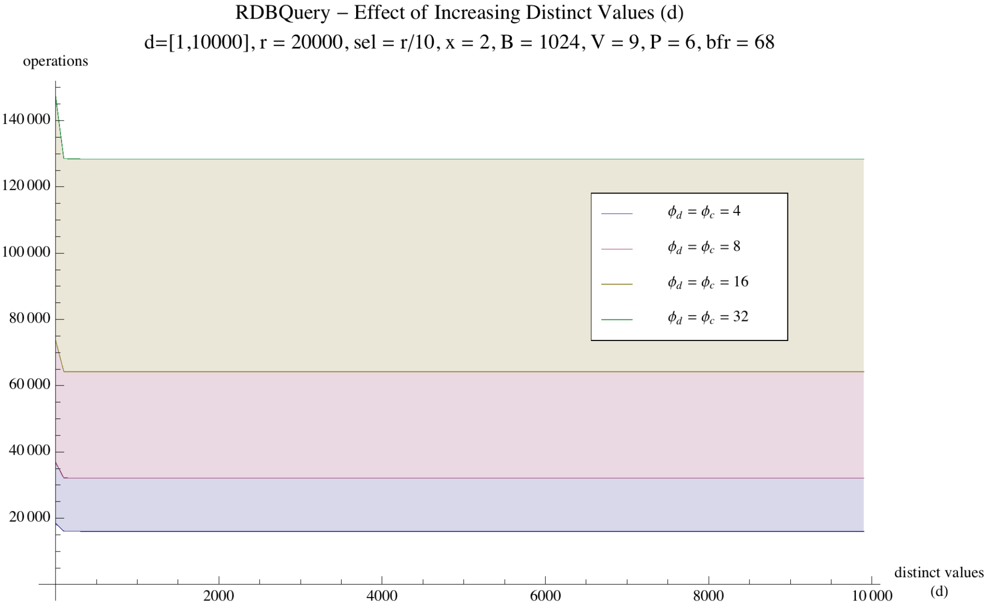


Figure 8.21: RDBQuery, distinct values increasing (fixed selectivity)

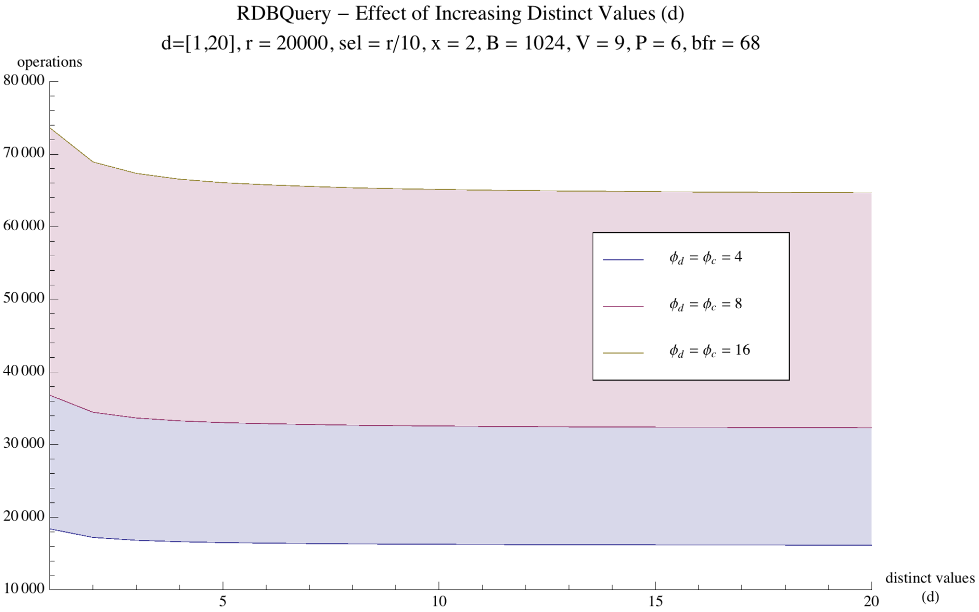


Figure 8.22: RDBQuery, distinct values increasing (fixed selectivity) - Zoom

Upon closer inspection of the term $\left\lceil \left(\frac{r}{d \times bfr} \right) \right\rceil$, we note that the ceiling function combined with the *bfr* causes it to quickly decline for small values of d but then see increasingly smaller changes as d increases. Using the same values as in Figure 8.22, Table 8.3 presents the first 10 results of that ceiling function with d values from one to 10. It is clear from the table that, as d increases, the value of the ceiling function in RDBQuery decreases by smaller amounts. This explains the sharp fall in Figure 8.22 followed by the seemingly level number of operations. Similar results are shown in Figures D.8 and D.9. Of the two graphs, Figure D.9 shows results most similar to those presented here with a constant selectivity size. In addition, Figure D.10 shows how RDBQuery reacts with a number of distinct values from one to 50, which represents 0.25% of the total records in the table. As d increases, RDBQuery performance increases. When all records are distinct, the term that involves d becomes extremely small.

8.2.3 Summary of Effects by RDBQuery Parameters

The experimental results presented show that some parameters of RDBQuery listed in Table 8.2 contribute more to the overall performance of the algorithm than others. The two dominant factors are the number of records r and the selectivity of the two selection operators used in the algorithm.

Value for d	$\left\lceil\left(\frac{r}{d\times bfr}\right)\right\rceil$
1	295
2	148
3	99
4	74
5	59
6	50
7	43
8	37
9	33
10	30

Table 8.3: Results of ceiling function in RDBQuery with d values from 1 to 10 ($r = 20000$, bfr= 68)

For a higher number of records, the algorithm must search a larger space for tuples that match the selection conditions. In addition, RDBQuery suffers from the same performance issues as relational databases. When selectivity is low (meaning that many results are returned), there is more work required to answer a query. However, when selectivity is high, the benefits of using an optimized RDB query processor are obvious. The number of distinct values d in the relation also contributes to the query performance, but its effect is neglegable when compared to other parameters. The most obvious parameters that impact RDBQuery performance are the number of child edges (ϕ_c) and descendant edges (ϕ_d) in the query. More edges translate to more nodes, and a longer query requires additional selection operations to be performed.

8.3 Overall Conclusions

While SS-Join and RDBQuery are both non-native approaches to query XML documents, they differ in their ability to perform queries of different lengths and styles. As noted in Section 5.5, SS-Join is only able to process a query of size two, does not perform on twig queries, is unable to successfully execute on documents with recursive nodes, and does not differentiate between ancestor/descendant queries and parent/child queries. These limitations make a true comparative analysis between SS-Join and RDBQuery highly limited, and SS-Join does not compete on a similar level as TwigStack, Constraint Sequencing, or RDBQuery. From the experiments presented in this chapter, we observe some general scenarios where SS-Join can outperform RDBQuery. Figure

8.2 illustrates the effect of increasing the ancestor and descendant positions in small lists. This represents a document that is more wide than deep (has a higher degree of fan-out). In this case, the number of ancestor nodes and descendant nodes are limited due to the limited/constrained depth of the document. We observe a steep decrease in the number of operations for SS-Join in this scenario. As we will demonstrate in Chapter 10, selectivity in the relational database is low for a wide/similar document. Even with a low selectivity (Figure 8.18), we note that, from the number of operations on the y-axis, SS-Join outperforms RDBQuery. For this reason, we include SS-Join in our final framework presented in Chapter 11, but we dismiss the necessity of an exhaustive comparison with RDBQuery.

In this chapter and Chapter 7, we presented a detailed analysis of the major contributing factors for the SS-Join, RDBQuery, TwigStack, and Constraint Sequencing algorithms. In Chapter 9, we examine the two native techniques, TwigStack and Constraint Sequencing, together and analyze their behavior for similar queries and XML documents. From these experiments, we select a technique that proves to outperform the other in the majority of situations that use synthetic data and real-world parameters and compare that technique with RDBQuery in Chapter 10.

Chapter 9

Comparative Analysis of Native Techniques

Now that we have identified the salient features from the four individual XML query processing techniques, we now shift our focus to a comparative analysis between the two native techniques, TwigStack and Constraint Sequencing. In this chapter, we present the results of experiments that compare the performance of these two techniques when applied to similar scenarios. We investigate the effects of querying over a deep XML tree with small fan-out, a shallow tree with a high degree of fan-out, and a tree that lies somewhere between the two extremes (similar depth and breadth).

9.1 Overview

In the following experiments, we create small, synthetic datasets using Mathematica. This simplifies the analysis and make performance numbers more readable. To ensure our methodology scales appropriately when applied to a larger, more representative XML document, we utilize the DBLP XML dataset [dbl09] when appropriate. Although XML documents can, in theory, appear in any configuration, their practical shape skews towards a shallow tree with a large breadth (high degree of fan-out). The XML Data Repository at the University of Washington [xml09] houses many XML datasets, and all of them are more broad than deep. However, while an extremely broad tree tends to be more common, we investigate other representative cases in the interest of generating a complete framework and model. Tables 7.1 and 7.2 in Chapter 7 illustrate the various parameters used for TwigStack and Constraint Sequencing, respectively, throughout the analyses that follow. For Constraint Sequencing, we introduce two variations of the term m. In previous chapters, m referred to both the document size and the amount of the document that was sequenced (a worst case scenario). In order to better compare with other techniques, m must be divided into two separate parameters, m_{doc} and m_{seq}. The value of m_{doc} is the size of the XML document while the value of m_{seq} refers to the portion of the document that is subjected to sequencing. Since the value of m_{seq} will always be less than (or in the worst case, equal to) the size of the document, it is appropriate to view the size of m_{seq} as a window into the document m_{doc} that is subject to sequencing. Refer to Equation 4.10 in Chapter 4 for more detailed information about this behavior. As a final overview note, when we state that an XML document or tree has a small breadth, this refers to the low degree of fan-out in the document.

9.2 Deep Tree, Low Breadth (Deep)

We first present the case where an XML document is deep and has a small breadth. In TwigStack, the rate at which T_i decreases represents the breadth of the document. If we force T_i to decrease quickly, it means that there are few nodes that are in each stream T_i. To represent a deep tree in Constraint Sequencing, we allow the branching factor b to remain low. The term ψ_x in TwigStack refers to the local breadth of the query, not the breadth (fan-out) of the original XML document. For that reason, we fix ψ_x and later investigate the effects of increasing this value. To increase readability and simplify terminology, we allow the term *deep* to refer to a tree that is deep in terms of node recursion (many levels) but low in breadth.

9.2.1 Experimental Results

Figure 9.1 illustrates the effect of various values for m_{seq} in Constraint Sequencing. At the top of the figure, we note that this experiment is for query sizes q from one to 50. Unless otherwise stated, it is assumed that query sizes are discrete values in single-step increments. The second line refers to the parameters used in Constraint Sequencing, and the third line lists parameters for TwigStack. When an equality is shown, it is read as a static value for the given parameter. For example, the values of b, s, and ψ_x are all static and equal two. If a parameter is allowed to assume a random value, that is noted by the capital letter R followed by the range for the random values. In Figure

9.1, the value for $S_{parent(x)}$ is allowed to assume a random value between two and five, inclusive. Unless otherwise stated, all random data is the result of a mean of 250 executions. If an arrow is shown, it is interpreted as an increase or decrease in the value of the term. In the legend of Figure 9.1, the value of m_{seq} is shown with a downward facing arrow followed by a number. The second legend entry means that the value of m_{seq} is decreasing (denoted by the down arrow) by a constant rate (denoted by the x) of 1.1. In other words, for successive iterations of the Constraint Sequencing algorithm, the value of m_{seq} is decreased by 1.1 (10%). This same notation is used throughout this chapter and the other comparative analyses in the following chapters. Figure 9.1 also demonstrates how our query performance studies are visualized. Two techniques are represented, one on each axis. In this case, TwigStack is on the x-axis while Constraint Sequencing is on the y-axis. The line $y = x$ that runs through the graph divides it into two triangle sections. The points represent queries of various sizes, and smaller queries are closer to the origin. Points that fall above the diagonal line represent queries that perform better for the technique on the x-axis while those that fall below the line correspond to queries that perform better for the technique on the y-axis. In Figure 9.1, since all points fall above the diagonal line, all queries perform better when run using TwigStack.

In Figure 9.1, we observe the effect of varying m_{seq} in Constraint Sequencing. This experiment is run in the presence of few identical sibling nodes (s) and a low value for $S_{parent(x)}$. As m_{seq} decreases more rapidly, the performance of Constraint Sequencing approaches the performance of TwigStack. This is shown by the yellow and green lines that approach the dividing line. When we increase the value of s by one (E.1), Constraint Sequencing demonstrates its expected behavior with a decrease in performance. Similar results are shown in Figure E.2 when we allow $S_{parent(x)}$ to decrease by a factor of two. An interesting result is shown in Figure 9.2. When we decrease the number of identical sibling nodes, query performance is not only similar for the various values of m_{seq} but is also the same for each technique. As we will observe later in the chapter, this is not the case for a shallow tree with a higher breadth. We note that, as m_{seq} decreases more quickly, the performance of Constraint Sequencing relative to TwigStack improves. We choose a representative behavior for m_{seq} as decreasing by a factor of two for future experiments. Wang and Meng also illustrate this behavior by their algorithm [WM05].

Since the behavior of the streams T_i are dictated by the shape of the XML document, only two

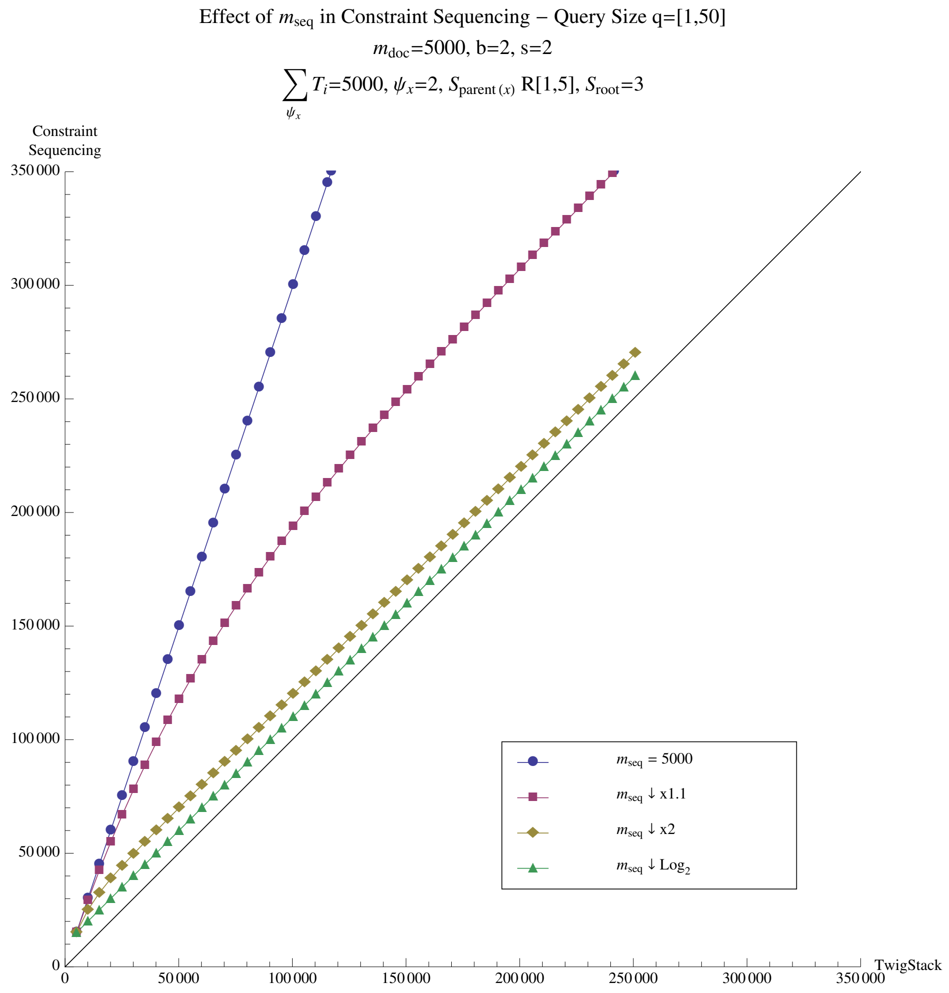


Figure 9.1: CS, vary sequence size (low random $S_{parent(x)}$) - Deep

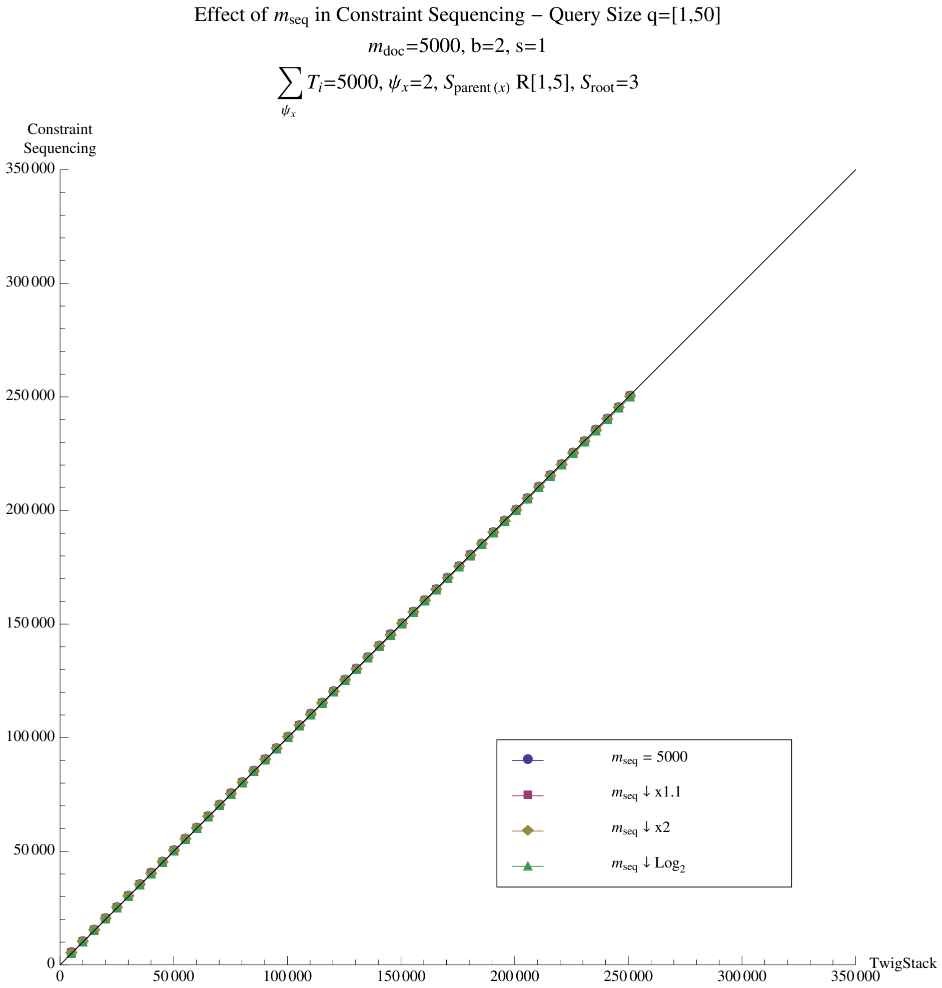


Figure 9.2: CS, vary sequence size (low random $S_{parent(x)}$, low s) - Deep

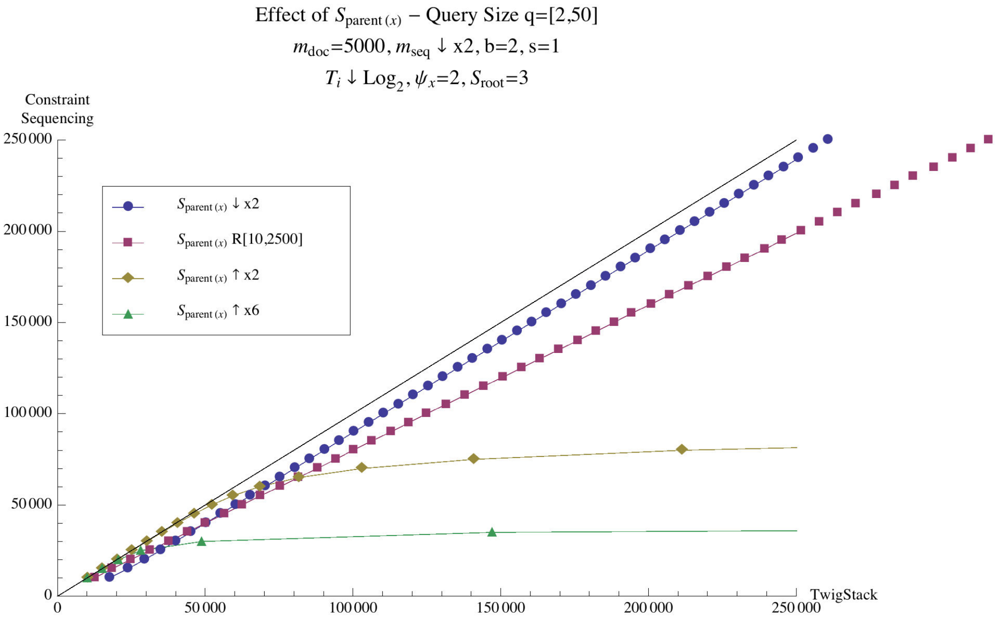


Figure 9.3: TS, vary stack size (low s) - Deep

parameters from TwigStack, $S_{parent(x)}$ and ψ_x, need be investigated. In addition, the branching factor b and behavior of m_{seq} remain fixed, so the remaining parameter from Constraint Sequencing, s, also requires analysis. Figure 9.3 illustrates the behaviors of TwigStack and Constraint Sequencing with various $S_{parent(x)}$ behaviors. Similar results are shown in Figures E.3 and E.4. Without many identical sibling nodes, Constraint Sequencing outperforms TwigStack for all behaviors of $S_{parent(x)}$ with all query sizes. Recall that the sizes of stacks in TwigStack reflect the number of results generated by the algorithm before pruning. A lower value for $S_{parent(x)}$, or a behavior that decreases, models relatively few intermediate results. As the stack sizes increase, more and more intermediate results are produced. This affects the outer summation of the TwigStack equation. If use the same parameters in Figure 9.3 but increase the number of identical sibling nodes to three, we see a dramatic shift in query performance as shown in Figure 9.4. With more identical sibling nodes, performance of Constraint Sequencing degrades. Since identical sibling nodes do not affect TwigStack, this pushes the performance for queries with a size less than 37 above the line. Therefore, if there are identical sibling nodes, small to medium sized queries perform better with TwigStack than Constraint Sequencing. However, all of this is still dependent on the behavior of the stack sizes. If stack sizes steadily increase, performance quickly shifts in favor of Constraint

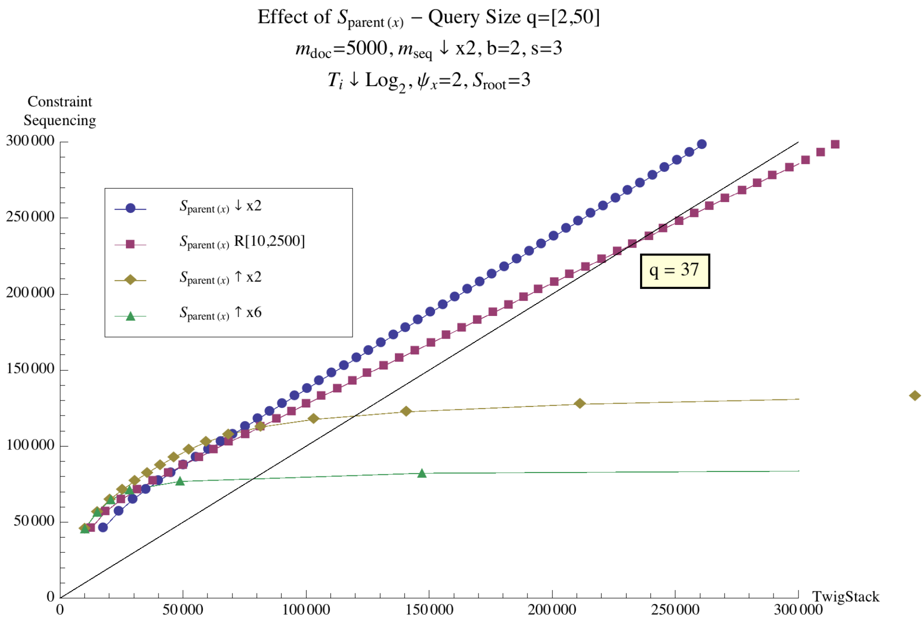


Figure 9.4: TS, vary stack size (increased s) - Deep

Sequencing. If we allow the number of sibling nodes to fluctuate between two and 10, results are pushed even further in favor of TwigStack (Figure E.5). If we continue the trend shown in this figure by increasing the maximum query size to 200, we observe that at a query size of 155, performance shifts in favor of Constraint Sequencing if the size of TwigStack stacks do not decrease rapidly. This is illustrated in Figure 9.5. As a point of reference, if a value of q is shown on a graph in a yellow box, this refers to the query length where the performance shifts in favor to the other technique. This notation is used on future graphs when necessary. By continuing to increase the number of identical sibling nodes, we shift performance in favor of TwigStack (Figure E.6).

For deep XML documents, the experiments above show that there is a distinct advantage to using Constraint Sequencing, provided there are few or no identical sibling nodes. However, once the number of identical sibling nodes begins to increase, the performance of smaller queries is better when using TwigStack. These results correspond to the experimental results of the constraint sequence authors illustrated in the original work [WM05].

The final parameter of interest, ψ_x, only affects the TwigStack algorithm. When we reference the TwigStack equation (Equation 3.5), we note that ψ_x only directly affects the value of the streams T. In order to model a deep tree with a shallow breadth, we must force the all streams T_i

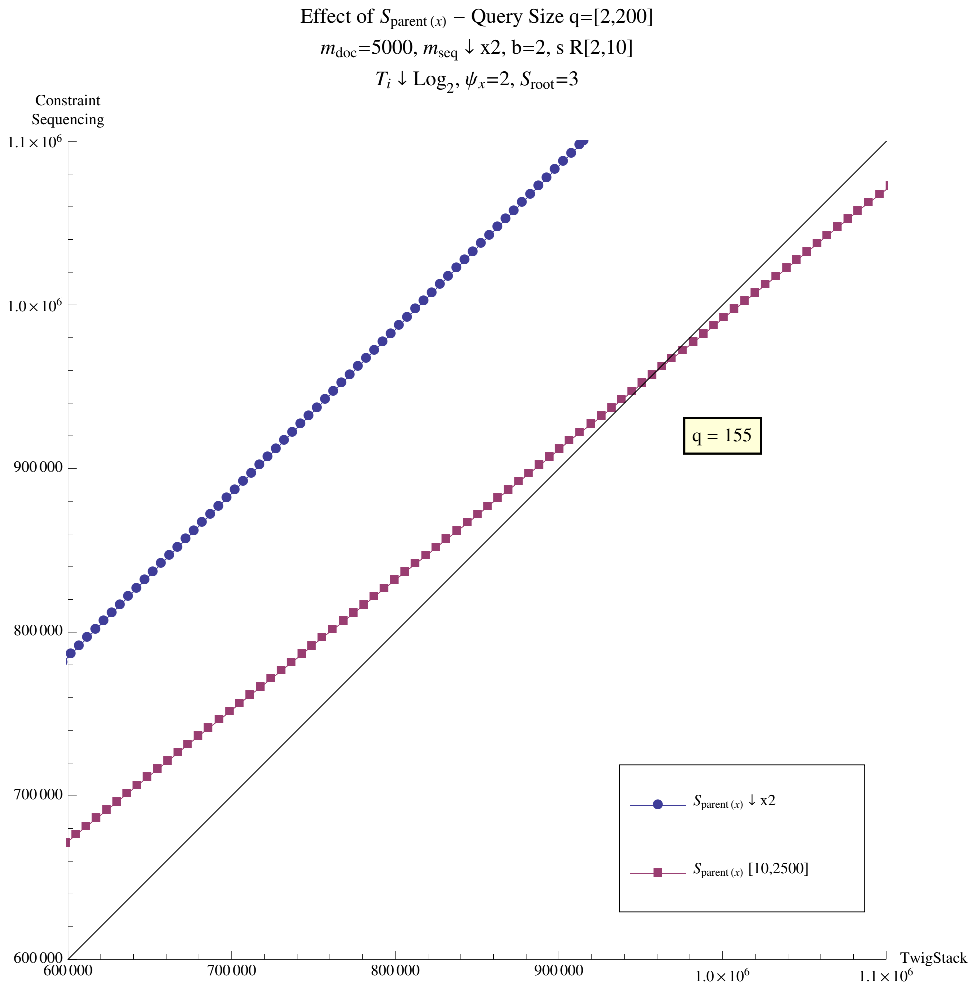


Figure 9.5: TS, vary stack size (random s, larger query) - Deep

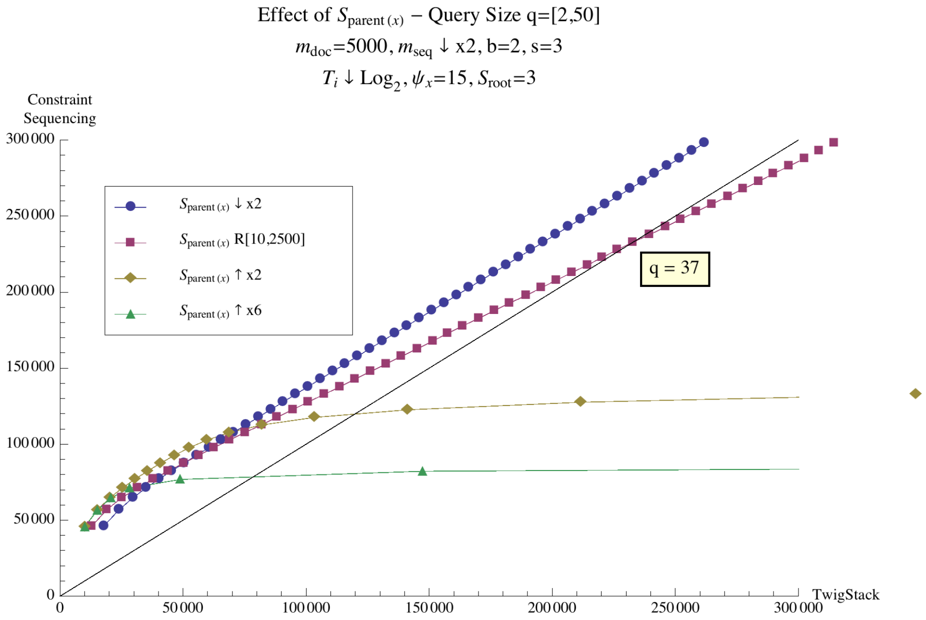


Figure 9.6: TS, vary stack size (increased s, high ψ_x) - Deep

to either be small or decrease rapidly. Therefore, all values of T_i will be small with respect to a tree that has a higher degree of fan-out. The result is that the shape of the TwigStack query (reflected by ψ_x) has little impact on query performance. This is validated by results shown in Figures 9.6 and E.7. Figure 9.6 represents the same data shown in Figure 9.4 but with an increased value for ψ_x. Note that the performance experienced little change. While the raw data that generated the two graphs slightly differs (better performance is shown with a smaller ψ_x), this is not reflected in the graph. In fact, query performance remains so similar that the shift in preference from TwigStack to Constraint Sequencing remains at a query size of 37. Figure E.7 illustrates similar behavior and corresponds to Figure E.3.

9.2.2 Conclusions

From the experimental results that show the relative performance of TwigStack and Constraint Sequencing on deep trees, several important conclusions are drawn. First, in the absence of identical sibling nodes, Constraint Sequencing outperforms TwigStack without regard to the size of the stacks. Second, as the number of identical sibling nodes increases, better performance shifts to TwigStack if the sizes of the stacks avoid being too large or consistently decrease. If stack sizes are extremely large or steadily increase, performance that favors TwigStack for small query sizes quickly shifts to Constraint Sequencing. The number of identical sibling nodes is what ultimately determines when and if performance shifts from TwigStack to Constraint Sequencing. Third, in deep trees, the shape of the query used in TwigStack has a negligible affect on performance. As is shown in the following section, this is not the case for a tree that is shallow in depth but large in breadth.

9.3 Shallow Tree, High Breadth (Wide)

As a complete contrast to a deep tree with a small breadth, we now consider the opposite case of an XML document that is shallow but extremely wide. For our purposes, the term *wide* refers to a shallow tree with a high degree of fan-out (increased breadth) and is used to simplify our terminology. To specify a wide tree, we force T_i to decrease slowly and increase the value of b to reflect a larger branching factor. Similar to the previous section, we fix ψ_x to a low value and later investigate the effects of a wider query shape for TwigStack.

9.3.1 Experimental Results

With a procedure similar to that in the previous section, we studied the performance of TwigStack and Constraint Sequencing on an XML document that, while similar in size, is opposite in terms of its shape. The analysis and presentation of graphs mirrors the previous section. When necessary, comparisons are made to the results for deep trees. Figure 9.7 illustrates the effect of m_{seq} when varied over a wide tree. Immediately we observe that there is a dramatic shift in favor of TwigStack when compared to Figure 9.1. Note that the parameter ψ_x remains two, and this signifies a small breadth for the query q in TwigStack. If we increase that factor to 10 and run the same experiment, we observe an equally dramatic shift in favor of Constraint Sequencing (Figure 9.8). By further modification of ψ_x, we notice another shift as illustrated in Figure E.8. This leads us to the conclusion that, while unimportant for deep trees, the shape of the query when performed on a wide XML document has important consequences to query performance. Figure E.9 shows how query performance continues to skew in favor of TwigStack, and this is to be expected with a decreasing $S_{parent(x)}$. Recall the interesting result shown in Figure 9.2 where TwigStack and Constraint Sequencing performed equally. Figure 9.9 shows the results of the same experiment but

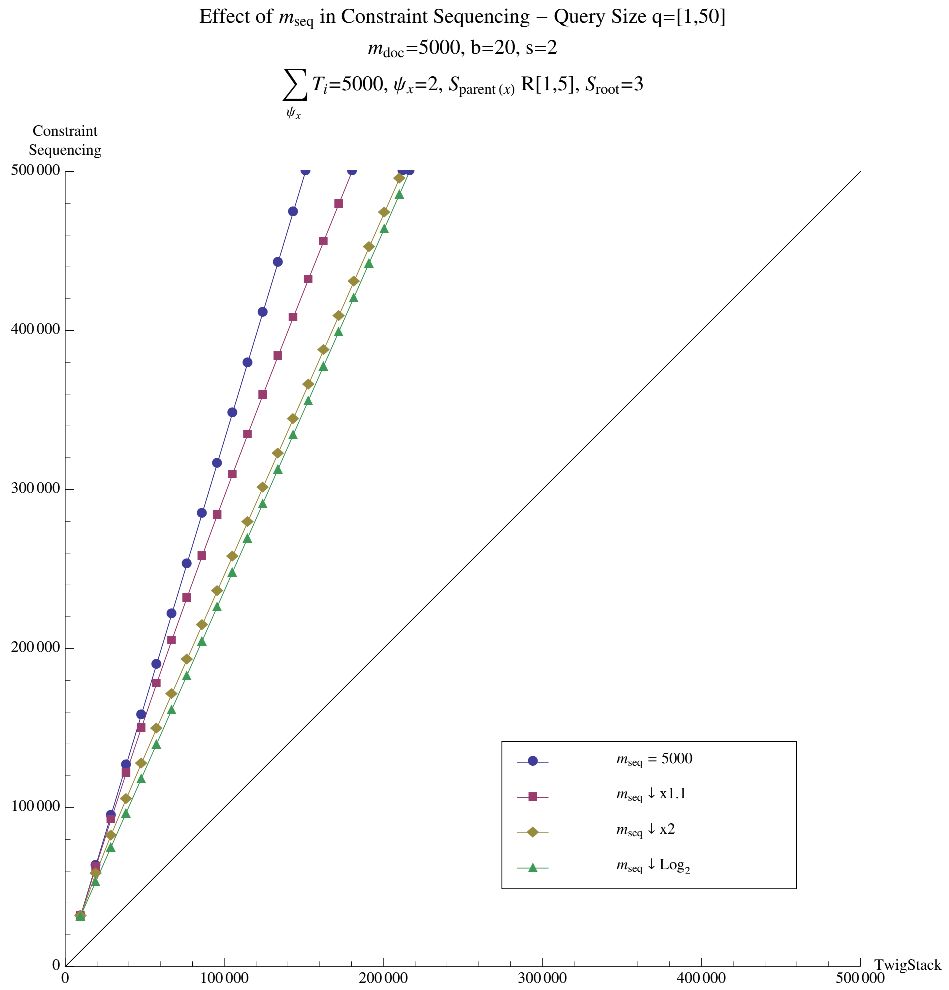


Figure 9.7: CS, vary sequence size (low $S_{parent(x)}$, low ψ_x) - Wide

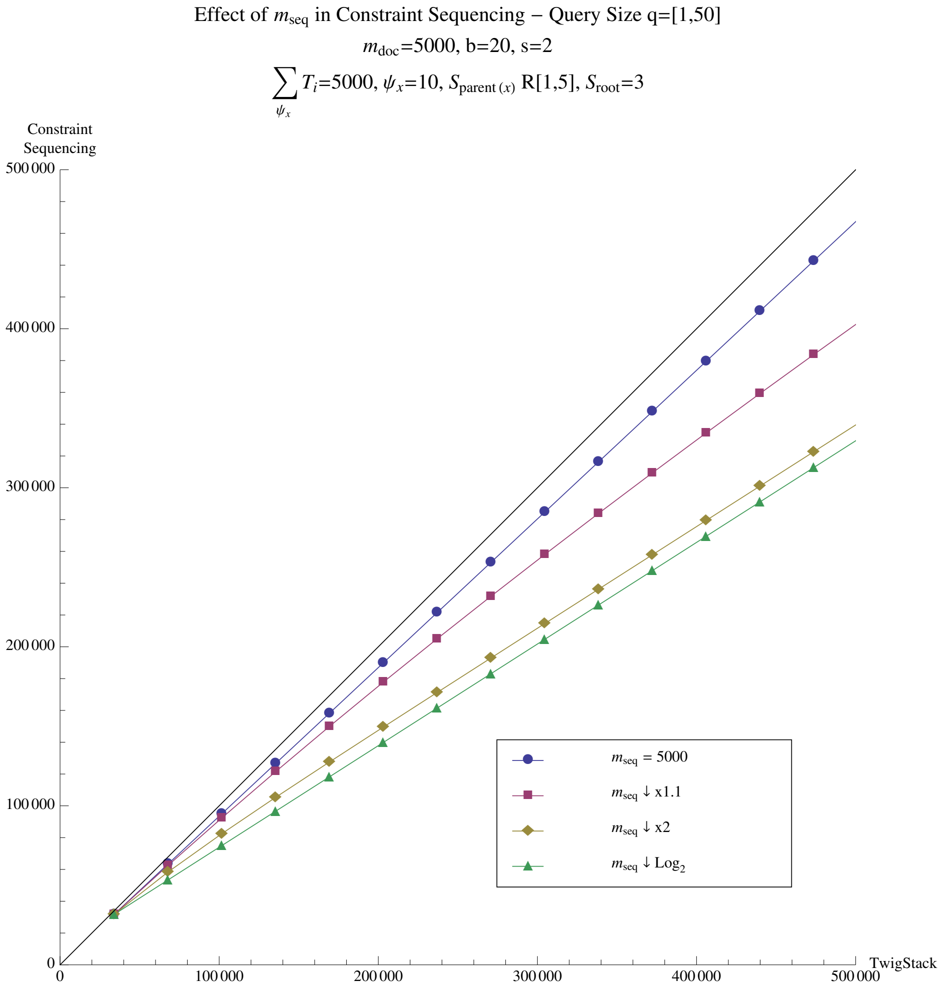


Figure 9.8: CS, vary sequence size (low $S_{parent(x)}$, high ψ_x) - Wide

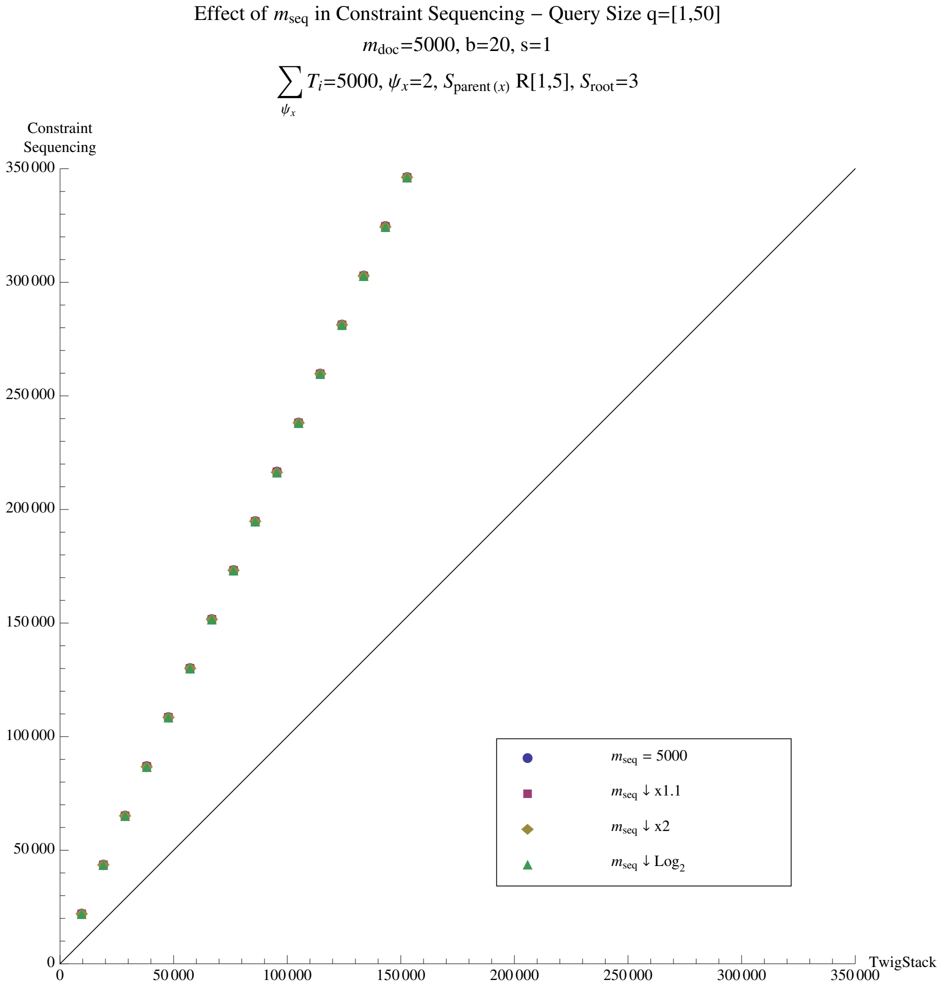


Figure 9.9: CS, vary sequence size (low s, low ψ_x) - Wide

performed on a wide document. Note that query performance now trends in favor of TwigStack. However, if we increase the breadth of the query by raising ψ_x, the reverse observation can be made (Figure E.10). As a final experiment into the effect of m_{seq} in wide documents, Figure E.11 shows that as the number identical sibling nodes increases, query performance favors TwigStack over Constraint Sequencing.

As in Section 9.2, we fix m_{seq} to decrease by a constant factor of two and investigate the parameters $S_{parent(x)}$, s, and ψ_x. In order to simulate a wide XML document, we force the streams T_i to decrease at a much slower rate than in Section 9.2. In addition, we increase the branching factor b to 10 as this allows Constraint Sequencing to be accurately compared to TwigStack. For

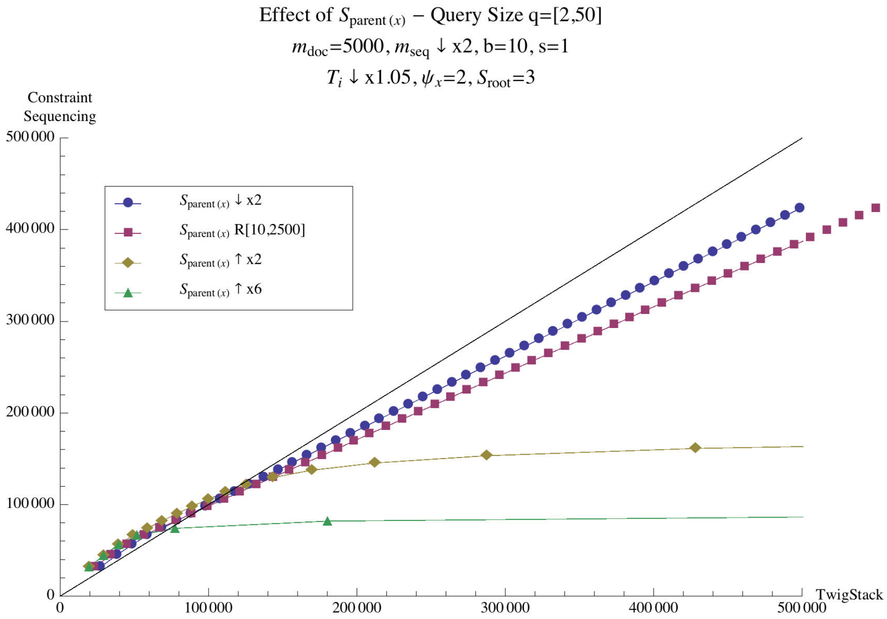


Figure 9.10: TS, vary stack size (low s) - Wide

the time being, we fix ψ_x to a constant value of two. This is the same value used in the previous section, and it allows us to compare our results and draw valid conclusions. Later in this section we investigate the ramifications to query performance of an increased ψ_x value. Figure 9.10 shows the results of queries over a wide document with a small number of identical sibling nodes. Figures E.12 and E.13 in Appendix E illustrate similar behavior. When compared to Figure 9.3, we notice a slight bump in TwigStack performance for small query sizes regardless of the $S_{parent(x)}$ behavior. However, unlike the results for deep trees, TwigStack performance degrades more quickly when performance does shift back in favor of Constraint Sequencing. This occurs for queries greater than 10 in length. Figure 9.11 illustrates the same experiment but with an increase to the number of identical sibling nodes. We observe the same behavior as displayed in the case for deep XML documents; as the number of identical sibling nodes increases, query performance shifts in favor of TwigStack. In Figure 9.11, we note that for queries longer than 33 nodes (items), Constraint Sequencing outperforms TwigStack. The effect of $S_{parent(x)}$ is reflected more for decreasing values or a random sampling. In cases where the stack sizes are strictly increasing, performance of TwigStack performs similarly on both deep and wide XML documents. If we allow s to increase further, we observe an increased shift towards TwigStack (Figure E.15). By increasing the maximum query

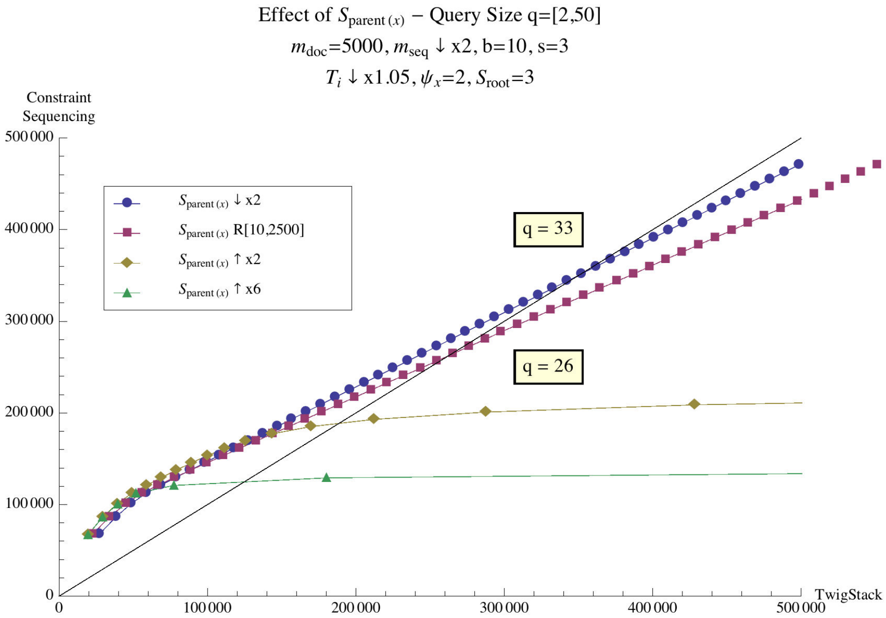


Figure 9.11: TS, vary stack size (increased s) - Wide

size and isolating our view to where performance shifts in favor of Constraint Sequencing, we see in Figure 9.12 that this performance shift occurs at query length 92 (for random $S_{parent(x)}$ values) and query length 101 (for consistently decreasing stack sizes). Contrast this to the results shown in Figure 9.5. In the case for deep trees, query performance remains in favor of TwigStack for longer query sizes (up to a length of 155), and, in the case of a consistently decreasing stack size, performance never favors Constraint Sequencing. This illustrates that TwigStack is influenced more by the breadth of an XML document than by its depth. A continued increase in s is shown in Figure E.16. Query performance is heavily skewed in favor of TwigStack. Although the blue and red lines eventually intersect the diagonal divider, this does not occur until the query length exceeds 400 items.

In Figure 9.13, we allow the branching factor b to start high then quickly diminish to a low value. This models a tree that has a high degree of fan-out at the first level (after the root node) then becomes a deep tree with little or no fan-out. In this case, for small values of $S_{parent(x)}$, TwigStack outperforms Constraint Sequencing until the query length exceeds 12. However, when stack sizes increase, Constraint Sequencing is favored over TwigStack for queries larger in size than six. A similar result is shown in Figure E.14 but with an increase in the number of identical sibling

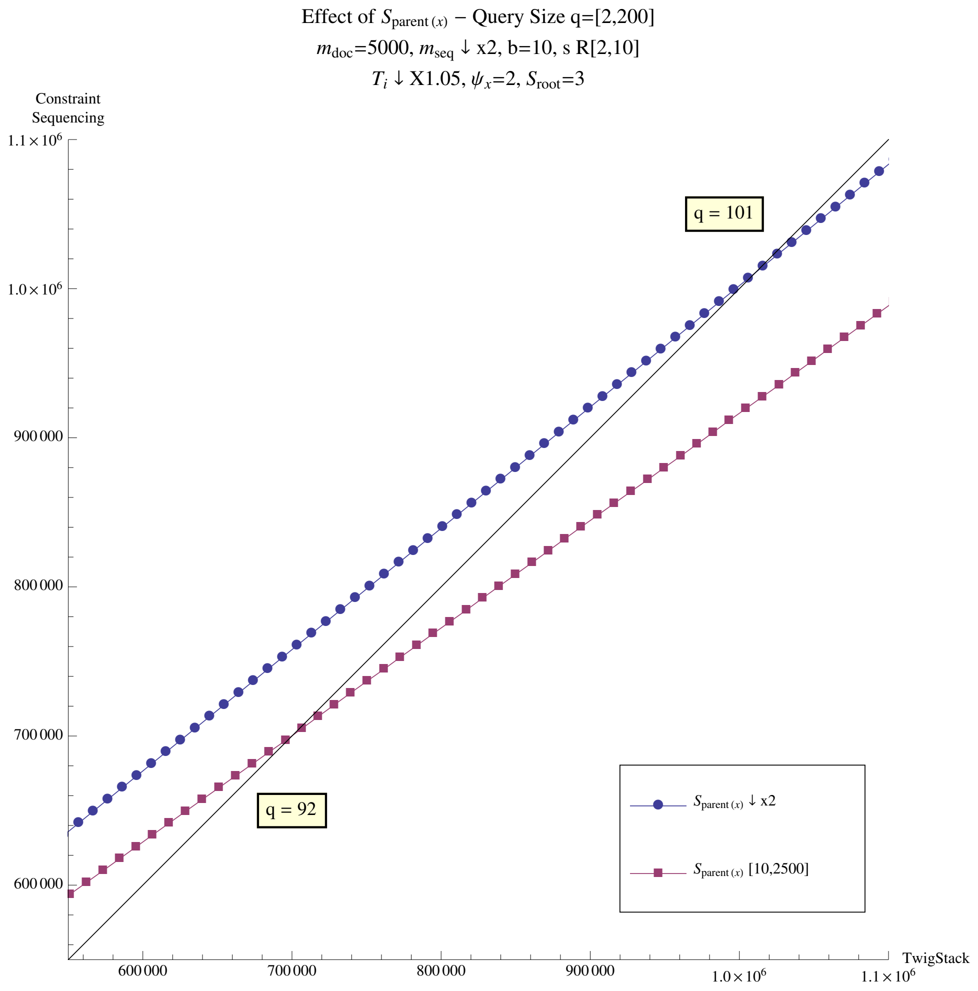


Figure 9.12: TS, vary stack size (random s, larger query) - Wide

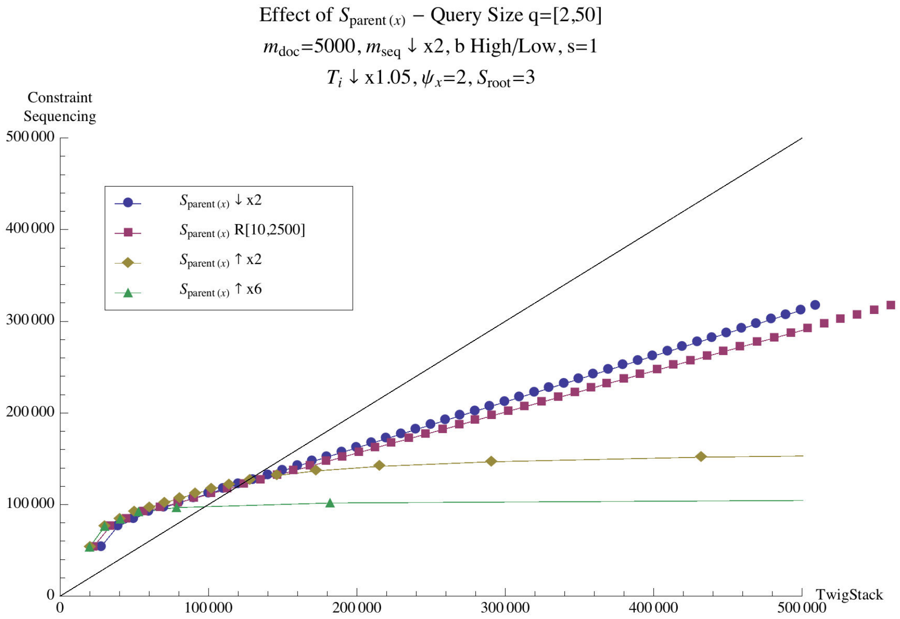


Figure 9.13: TS, vary stack size (b high/low) - Wide

nodes. As expected, performance is shifted in favor of TwigStack.

As mentioned earlier, the effect of the query shape (not the XML document shape) has a profound impact on TwigStack. By increasing the value of ψ_x, we force TwigStack to operate with a query that has a high degree of fan-out. Figure 9.14 shows the effect of increasing the breadth of the query. Since Constraint Sequencing does not have a term that reflects query breadth, it remains unaffected by this change. This is similar to the way that TwigStack is unaffected by identical sibling nodes. The results shown are dramatic and clearly indicate a strong bias towards Constraint Sequencing. By increasing the number of identical sibling nodes to three (Figure E.17) and then to a random value between two and 10 (Figure E.18), we continue to observe a clear preference for Constraint Sequencing over TwigStack. Figure 9.15 illustrates the case for queries that have fan-out (at various levels of the query) between two and 10. While query performance appears to track as a similar value for both TwigStack and Constraint Sequencing for extremely low query sizes, Constraint Sequencing quickly overtakes TwigStack in terms of performance for low, fixed values of s.

However, when the value of s is allowed to increase to a high, random range, there is a dramatic shift back towards an increased performance of TwigStack over Constraint Sequencing. Figure 9.16

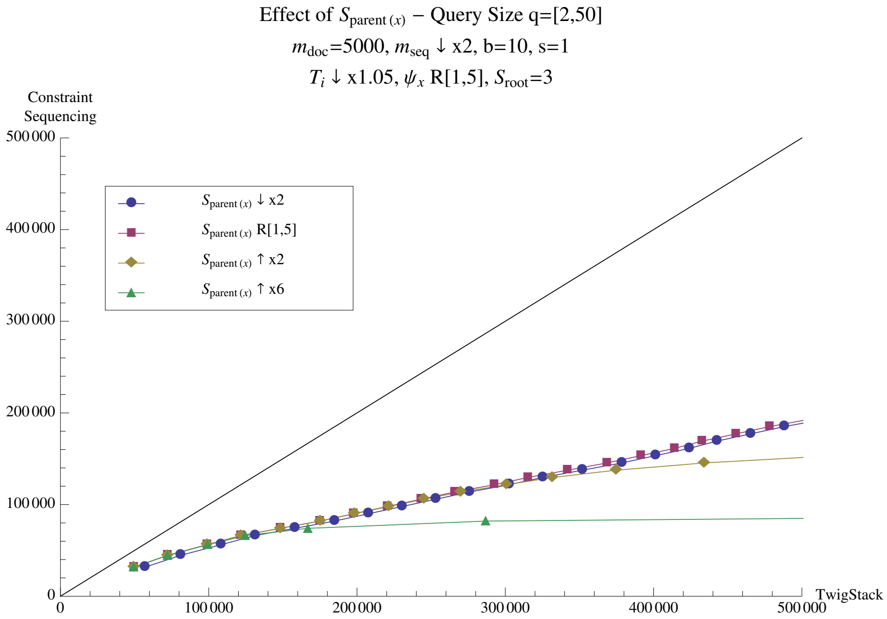


Figure 9.14: TS, vary stack size in small random range (low s, high ψ_x) - Wide

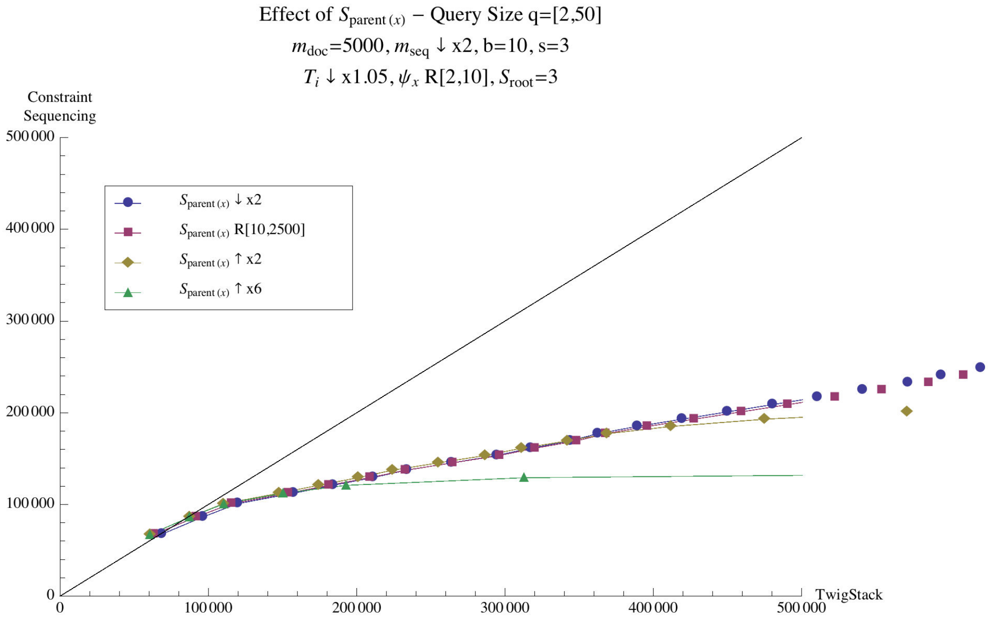


Figure 9.15: TS, vary stack size (increased s, random ψ_x) - Wide

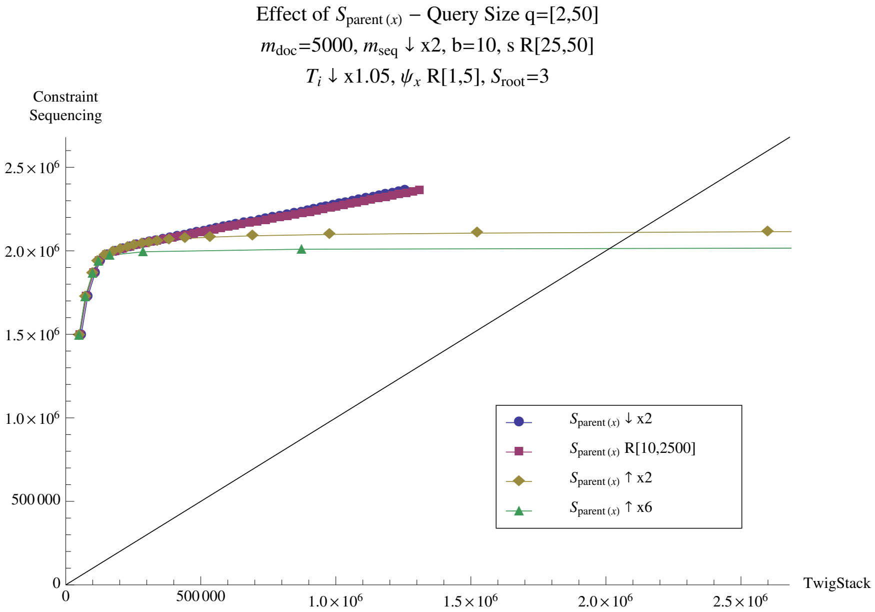


Figure 9.16: TS, vary stack size (high random s, high ψ_x) - Wide

shows that, for queries less than a length of 39, TwigStack outperforms Constraint Sequencing. As query size increases, TwigStack performance degrades while Constraint Sequencing performance decreases slightly. This is because, with a query that exhibits a high degree of fan-out, TwigStack must so more work in the inner summation across all ψ_x. For a deep tree, the relatively low fan-out keeps this summation to a low number of iterations.

9.3.2 Conclusions

From the experimental results that show the relative performance of TwigStack and Constraint Sequencing on wide trees, several important conclusions are drawn. First, in contrast to the corresponding case with deep trees, in the absence of identical sibling nodes, TwigStack outperforms Constraint Sequencing for small query sizes with a low query fan-out (low ψ_x). Once the query fan-out increases, Constraint Sequencing is the preferred method. Second, as was also shown for deep trees, the presence of identical sibling nodes dramatically reduces the effectiveness of Constraint Sequencing. The query shape is important when using TwigStack, and a query with high fan-out performs better using Constraint Sequencing unless the number of identical sibling nodes are high. This leads us to conclude that, unlike the case for deep trees, the shape of the query

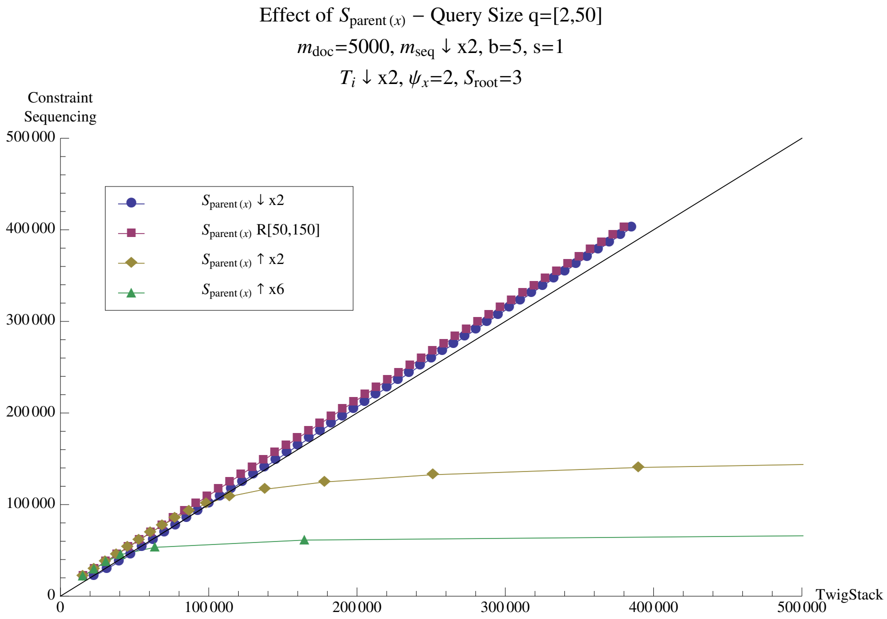


Figure 9.17: TS, vary stack size in medium random range (low s) - Similar Depth/Breadth

plays an important role when considering wide XML documents.

9.4 Trees with Similar Depth and Breadth

The performance results presented above were achieved by running the same set of experiments on XML documents that are very deep (Section 9.2) and very wide (Section 9.3). A subset of these experiments was performed on XML documents that have similar depth and breadth. The majority of the results are shown in Appendix F. As a whole, the results tend to slightly favor TwigStack over Constraint Sequencing. Figure 9.17 illustrates performance of the two techniques using the same experiment shown in Figures E.4 and E.13. While those figures showed a slight bias towards Constraint Sequencing, the result of the same experiment performed on a tree with similar depth and breadth gives a slight advantage to TwigStack for decreasing and relatively small stack sizes. When the stack size is increased to a random range or strictly increases in size, performance shifts back to Constraint Sequencing. Similar results, where TwigStack slightly outperforms Constraint Sequencing when it previously did not, are displayed in the remaining figures in Appendix F. As mentioned earlier in this chapter, the majority of available XML datasets have a much higher

```
<inproceedings key="conf/icpr/Little00">
  <author>James J. Little</author>
  <title>Deforming Surface Features Lines in Intrinsic
      Coordinates.</title>
  <pages>1291-1294</pages>
  <year>2000</year>
  <booktitle>ICPR</booktitle>
  <ee>http://computer.org/proceedings/icpr/0750/Volume%201/
      07501291abs.htm</ee>
  <url>db/conf/icpr/icpr2000-1.html#Little00</url>
</inproceedings>

<article key="tr/ibm/RJ1318">
  <author>Raymond F. Boyce</author>
  <author>Donald D. Chamberlin</author>
  <title>Using a Structured English Query Language as a Data Definition
         Facility.</title>
  <journal>IBM Research Report</journal>
  <volume>RJ1318</volume>
  <month>December</month>
  <year>1973</year>
  <ee>db/labs/ibm/RJ1318.html</ee>
  <cdrom>ibmTR/rj1318.pdf</cdrom>
</article>
```

Figure 9.18: Sample from DBLP XML Dataset

breadth than depth. For that reason, we place less emphasis on the results of the experiments on XML documents that are equally deep and wide when making our conclusions.

9.5 DBPL XML Dataset

All of the previous experiments in this chapter were performed on a relatively small synthetic dataset. To ensure that our equations scale correctly and our experiments continue to provide valid results for a large XML document, we modeled the DBLP XML dataset in Mathematica [dbl09]. As of this writing, the DBLP dataset had 3,332,130 elements with an average depth of 2.90228 (maximum depth of 10). This corresponds to an extremely wide tree with relatively shallow depth. A small sample of the XML document is shown in Figure 9.18. DBLP is a computer science bibliography database, and two entries in the database are shown in the figure. We modeled this dataset in Mathematica and performed the same experiments as described in Section 9.3 since this is a wide XML document. The results of these experiments are contained in Appendix G, and the results scale with the results shown and discussed in Section 9.3. When comparing these graphs to those created using a smaller dataset, the difference is the scale of the number of operations shown on the x- and y-axis. Figure G.13 shows the same experiment as that shown in Figure 9.12

in Section 9.3. Note that the red line crosses the boundary between TwigStack and Constraint Sequencing at almost the exact same length of q. The slight difference can be attributed to a slight variation when compiling the random data used for the number of identical sibling nodes.

9.6 Overall Conclusions

A definitive answer of which technique, TwigStack or Constraint Sequencing, outperforms the other in every situation is impossible to provide. Variables such as the number of intermediate results (which influences the sizes of the stacks in TwigStack) and the number of identical sibling nodes (which influences the performance of Constraint Sequencing) can change from one XML document to another and are highly dependent on the overall design of the document. However, most XML documents we surveyed represented a wide tree. We found one dataset, the Treebank project from the University of Pennsylvania, that has an average depth of 7.87279 (maximum depth of 36) and 2,437,666 elements [tre09], but this is still a tree with a high degree of fan-out. In addition, such trees typically display a lack of identical sibling nodes, and this fact was also noted by Wang and Meng [WM05]. The number of intermediate results produced by such trees tend to be large [BKS02], and this results in larger stack sizes for TwigStack. For those reasons, we select Constraint Sequencing as the technique that outperforms TwigStack for the majority of real-world cases. In Chapter 10, we perform a similar set of experiments and analyses on Constraint Sequencing and our algorithm that operates on XML data in a relational database, RDBQuery.

Chapter 10

Comparative Analysis of Constraint Sequencing and RDBQuery

In Chapter 9, we presented the results of a performance study between the two native XML query techniques, TwigStack and Constraint Sequencing. From that data and from a time/operations standpoint, we determined that Constraint Sequencing is the preferred technique for querying XML documents in their native form. In this chapter, we present the results and observations of a similar study, this time between Constraint Sequencing and our own technique, RDBQuery. We investigate the effects of querying over a deep XML tree with small fan-out, a shallow tree (wide) with a high degree of fan-out, and a tree the is in the middle of the two extremes.

10.1 Overview

Similar to the technique employed in Chapter 9, we used Mathematica to create small, synthetic datasets. We also continue to follow the assumption that most real-world XML documents appear as shallow trees with a high degree of fan-out. For our purposes, we label these trees as wide to differentiate them from deep trees. Table 7.2 and Table 8.2 show the parameters for Constraint Sequencing and RDBQuery, respectively. For the purposes of this comparison, we assume that the indices, both primary and secondary, that are utilized by RDBQuery are in memory. This allows us to compare RDBQuery on an equal basis with Constraint Sequencing. If the relational database indices were not in memory, we would need to account for disk access. We leverage the information obtained from Chapter 7, Chapter 8, and Chapter 9 to further reduce the search space for our experiments in this chapter. We observed that the presence of identical sibling nodes in Constraint Sequencing significantly reduces the effectiveness of the technique, so this chapter omits an in-depth analysis of the effects of the parameter s in Constraint Sequencing. For RDBQuery, we noted that the number of distinct values d has little impact on query performance. Therefore, it is unnecessary to perform detailed experiments that vary the number of distinct values. We fix the number of distinct values to be 30% of the number of records in the majority of our test cases. An increase in the number of distinct values will only improve the performance of RDBQuery.

10.2 Deep Tree, Low Breadth (Deep)

We first present the case where an XML document is deep and has a small breadth. In Constraint Sequencing, we force the branching factor b to remain low. If we keep the number of total nodes (length) of the document m_{doc} constant, this results in a deep tree. The methodology to force a deep tree structure in RDBQuery is slightly more complex. Since the XML document is flattened into a single relation (the *Edge* table), the internal structure of the document is lost. However, upon close examination of the RDBQuery algorithm, we note that the selectivity of the two relational queries helps determine the tree shape. With a deep tree, the two selections (lines 5 and 15 of Algorithm 6.1) return more tuples as results. The selectivity of the descendant edge query (line 5) is especially low, and a large number of tuples may be needed to satisfy the selection condition. When we visualize a deep tree with limited breadth, this appears intuitive. Consider a node at level two of an extremely deep tree. This node has more descendants than a similar node in a wide tree. As was shown in Section 8.2.1, the performance of RDBQuery relies heavily on the selectivity of the nodes in the query. If the selectivity is low, a larger number of results are returned and performance degrades. In the opposite case, few results are returned and performance is improved. While, in general, selectivity is lower for deep trees, we acknowledge the possibility that a particular XML document may display high selectivity in this case. For that reason, we study the effects of selectivity through the spectrum in both the deep and wide test cases. To conserve space, selectivity is abbreviated as *sel* on our graphs. Unless otherwise noted, graphs that include random values are the result of a mean of 250 executions. The terminology on the graphs is the same as that used in Chapter 9. As a final note before we present our results, we use the term *low selectivity* to

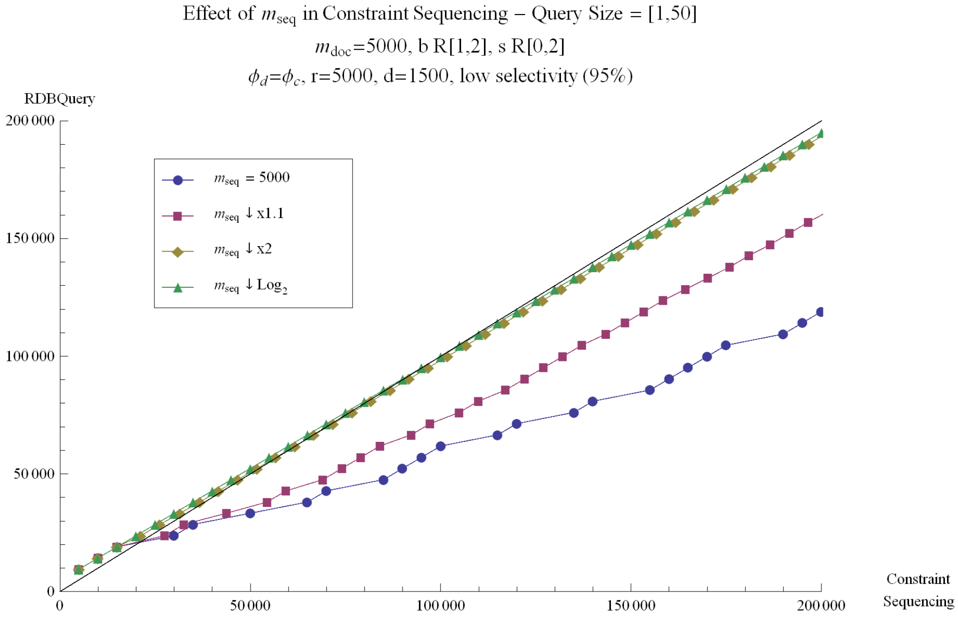


Figure 10.1: CS, vary sequence size (low selectivity) - Deep

describe a query that is not selective. In this case, a large number of tuples are returned from the database. The percentage shown in the graphs next to the selectivity indicates how many records are returned from the selection.

10.2.1 Experimental Results

Figure 10.1 illustrates the effect of various values for m_{seq} in Constraint Sequencing when RDBQuery is forced to have low selectivity. In keeping with our analyses for deep trees, the branching factor b is constrained to a low range, and the number of identical sibling nodes s are also constrained. As the document that requires sequencing (m_{seq}) decreases more rapidly, the performance of Constraint Sequencing improves. For the two most aggressive m_{seq} values, Constraint Sequencing outperforms RDBQuery until query length/size exceeds 18. As was the case in Chapter 9, we select a representative behavior for m_{seq} and run future experiments with this as a fixed parameter. Similar results are shown in Figures H.1, H.2, and H.3, which illustrate the same experiment with high selectivity and an increased number of identical sibling nodes. Of particular interest is the case where s is extremely low (one at the most), and this is shown in Figure 10.2. When the number of identical sibling nodes is low, the effect of m_{seq} can be ignored, and Constraint Sequencing

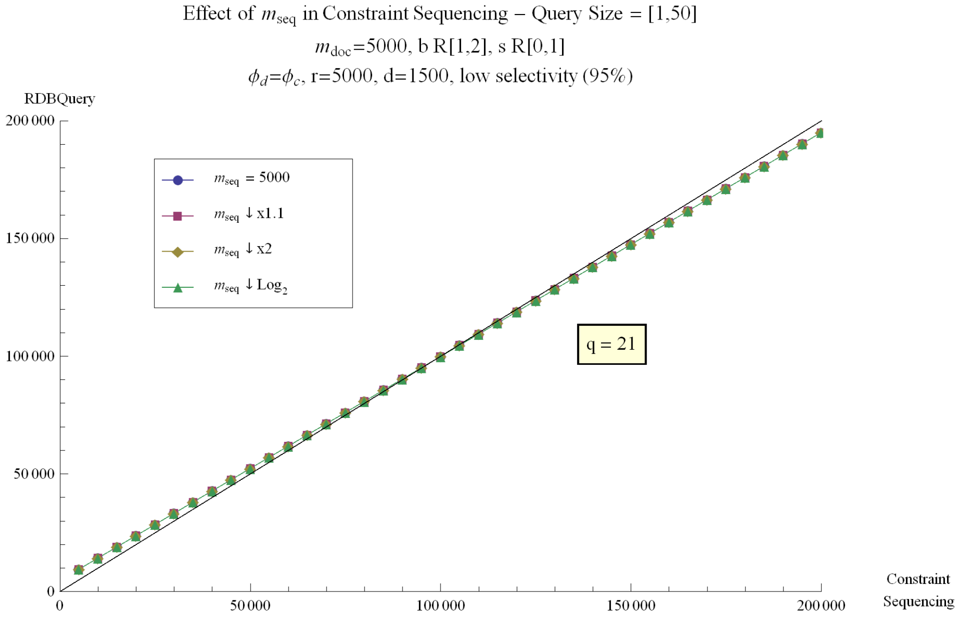


Figure 10.2: CS, vary sequence size (low selectivity, decreased s) - Deep

outperforms RDBQuery for an increased query size (up to length 21). Due to the structure of a deep tree, the possible number of identical sibling nodes is low. We limit most of our experiments for deep trees to a low number of identical sibling nodes, but we do present some outlying cases where a deep tree may have a large number of identical sibling nodes. In general, we conclude that Constraint Sequencing outperforms RDBQuery for deep trees where m_{seq} is decreasing quickly and the selectivity encountered by RDBQuery is low. This is exactly the scenario expected from a query over a deep XML document.

While the number of identical sibling nodes in a deep tree may be low, it is worthwhile to investigate their impact on Constraint Sequencing to confirm our previous results in Chapters 7 and 9. Figure 10.3 shows results when the number of identical sibling nodes in constraint sequence vary randomly in the range shown in the legend. For a low value of s, Constraint Sequencing outperforms RDBQuery. With a slight increase in s, the opposite is true. Preference does shift back to constraint sequence in the presence of one or two identical sibling nodes, but this does not occur until the query length is greater than 50. Figure 10.4 illustrates the effect of increasing selectivity in RDBQuery. When compared to Figure 10.3, we observe that the trend is the same, but the overall results are shifted in favor of RDBQuery. The blue line, which completely favored

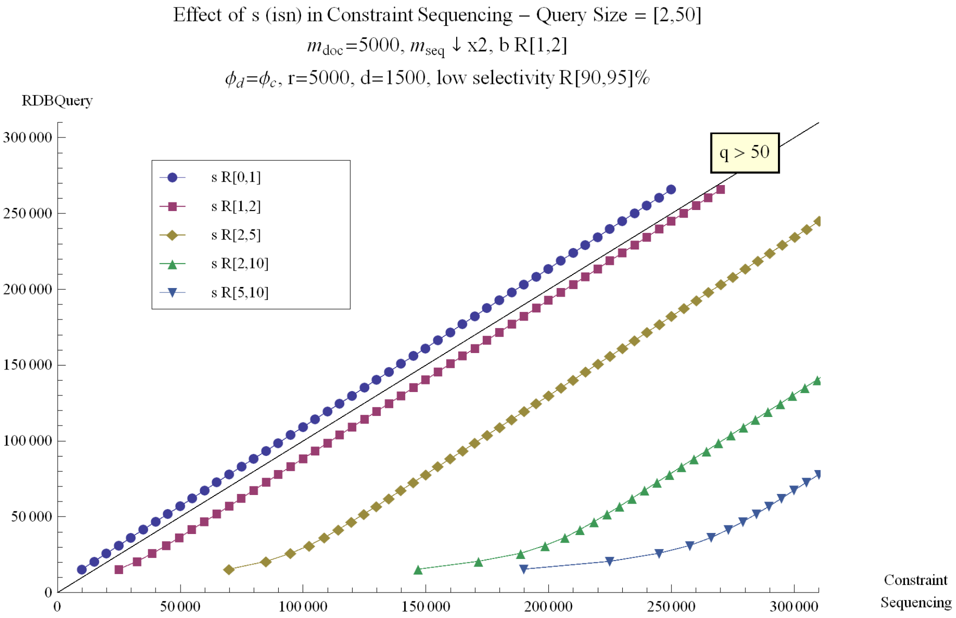


Figure 10.3: CS, random identical sibling nodes (low selectivity) - Deep

Constraint Sequencing in Figure 10.3, now shifts to prefer RDBQuery for queries greater than 21 in length. This is an expected result and displays the same trend as previously identified. Additional results are shown in Figures H.4 and H.5 where the selectivity of RDBQuery is increased, and these illustrate a continued shift in favor of RDBQuery as selectivity increases.

As mentioned at the opening of this chapter, the number of distinct values d encountered by RDBQuery does not affect overall performance enough to warrant an in-depth study of this parameter. To illustrate that claim, we present Figure 10.5. This experiment shows how query performance changes in RDBQuery based on a changing percent of distinct tuples in the database. Note that all seven trend lines overlap. This means that, regardless of the percentage of distinct values, RDBQuery performs the same. Figures H.6 and H.7 display this same trend but in the presence of more identical sibling nodes and an increased selectivity. Since we observe little if any difference in performance given a wide range of distinct tuples, we return the number of distinct values to a fixed 30% for the remainder of this section.

In Chapter 8, we observed that the number of descendant and child edges in an RDBQuery query affect the performance of the technique. We also acknowledged that the selectivity encountered by RDBQuery also affects query performance; a higher selectivity yields better performance and vice

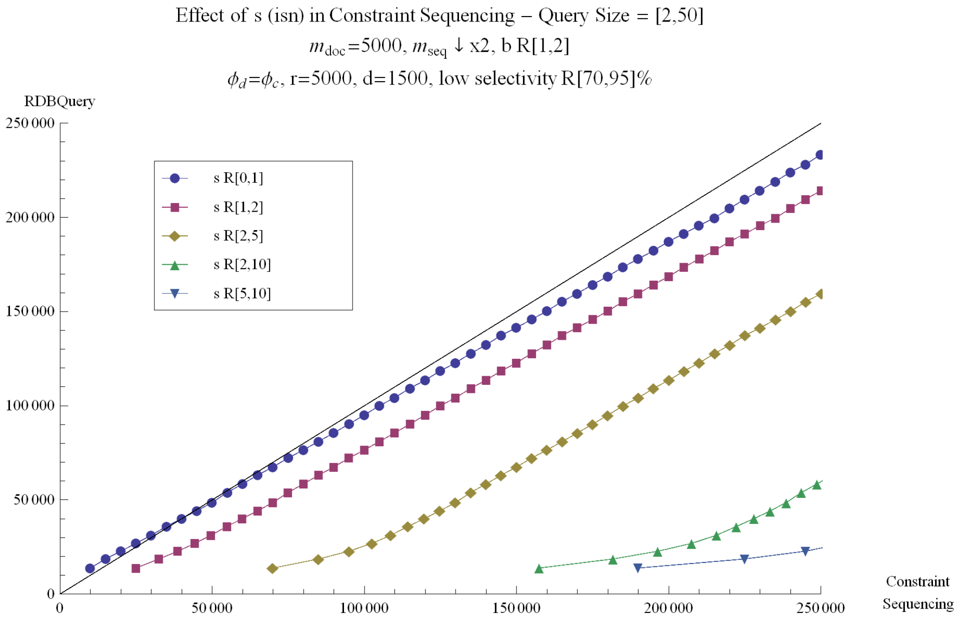


Figure 10.4: CS, random identical sibling nodes (increased low selectivity) - Deep

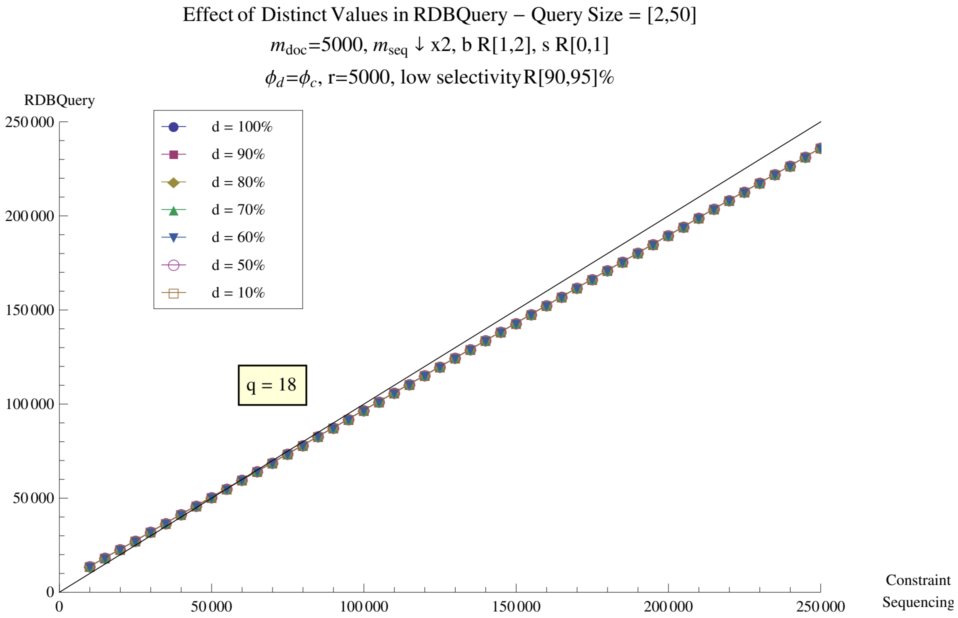


Figure 10.5: RDBQuery, vary distinct values (low selectivity, low s) - Deep

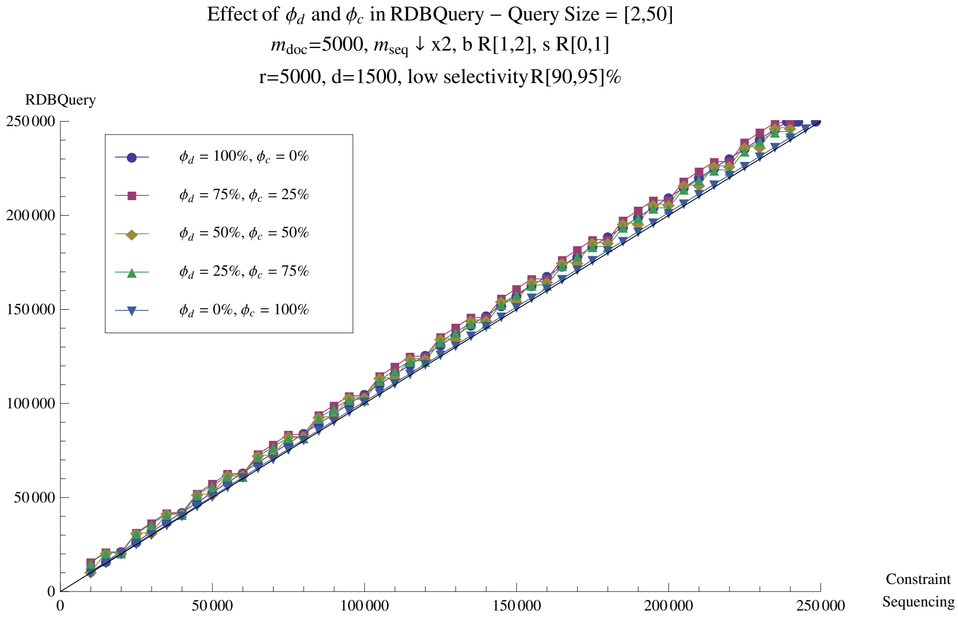


Figure 10.6: RDBQuery, query edge distribution (low selectivity) - Deep

versa. Our previous experiments in this chapter fixed the number of descendant and child edges to be the same ($\phi_d = \phi_c$). We now present a series of experiments that vary the occurrence of these edges while maintaining a constant selectivity. Figure 10.6 illustrates the change in RDBQuery performance by altering the percentage of descendant and child edges, and Figure 10.7 shows the same data but with some lines removed and in a closer view. As expected, Constraint Sequencing outperforms RDBQuery in all cases.

By increasing the selectivity and running the same experiment, we note a shift of performance to favor RDBQuery in Figure 10.8. While query size remains small ($q < 15$), performance of RDBQuery is approximately equal to that of Constraint Sequencing. However, as query length increases, performance shifts to favor RDBQuery. As query length increases, it becomes increasingly closer to the depth of the original XML document (now represented in an *Edge* relation/table). This means that there are continually fewer descendant nodes that could possibly satisfy the selection in RDBQuery. If we were to continue this to the extreme, our query would be equal in length to the depth of the document. As the algorithm progresses down the query tree, it must do less work when executing the descendant selection. This allows each successive selection on the database to return fewer results, and thus we observe the increased performance for RDBQuery, when compared with

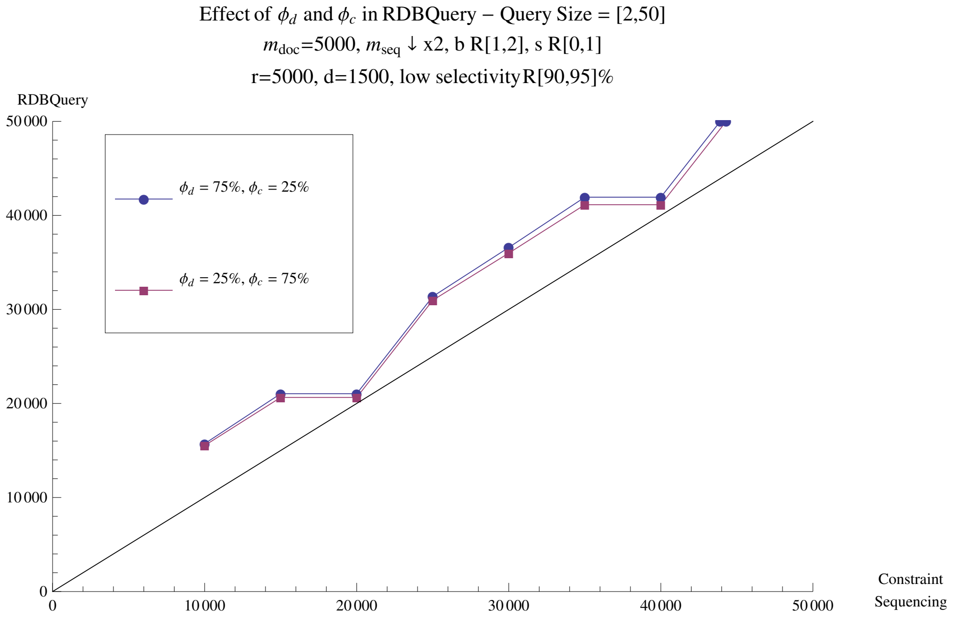


Figure 10.7: RDBQuery, query edge distribution (low selectivity) - Deep (Zoom)

Constraint Sequencing, as query length increases. Figures H.8 and H.9 in Appendix H demonstrate this same trend but for increased selectivity and in the presence of more identical sibling nodes.

While the previous set of experiments show that selectivity and the distribution of descendant and child edges factor into the performance of RDBQuery, they do not show the entire picture. For completeness, we must now separate the selectivity of the descendant edges from that of the child edges. Figures 10.9 and 10.10 show the effects of various descendant and child edge distributions. In Figure 10.9, the selectivity of descendant edges is lower than that of child edges, and these selectivity values are reversed in Figure 10.10. Our first observation is that the selectivity of the descendant edges, sel(ϕ_d), impacts RDBQuery more than the selectivity of child edges, sel(ϕ_d). Both graphs display a wedge shape, but that shape in Figure 10.9 widens to a greater extent than the shape in Figure 10.10. In addition, we note that RDBQuery performance is better when the number of descendant edges is low compared to the number of child edges. This effect is magnified when selectivity for ϕ_d is lower than that for ϕ_c, and that is what the larger and smaller wedge slices in the two figures tell us. Upon a closer examination of Figure 10.10, it should be noted that the order of the five lines shown in the legend are flipped when compared to Figure 10.9. The topmost line in Figure 10.10, which runs on top of the dividing line $y = x$, tells us that performance for

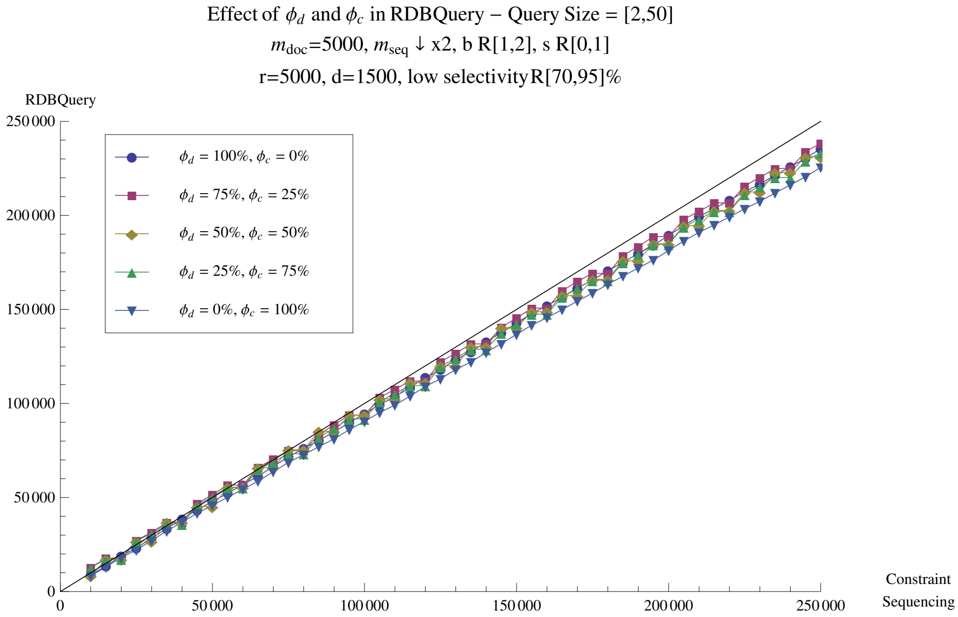


Figure 10.8: RDBQuery, query edge distribution (increased low selectivity) - Deep

RDBQuery and Constraint Sequencing is similar with 100% child edges with the given selectivity values. In Figure 10.9, we notice that the top line, which corresponds to 100% descendant edges, is above the dividing line. This illustrates that, with a higher number of descendant edges and a lower descendant selectivity, RDBQuery performs worse than Constraint Sequencing. In the opposite case, where the number of child edges is 100% and they exhibit the same low selectivity, RDBQuery performs as well as Constraint Sequencing but not better. Figures H.10, H.11, and H.12 demonstrate the same trend with varying selectivity values for descendant and child edges.

10.2.2 Conclusions

From the experimental results that show the relative performance of Constraint Sequencing and RDBQuery on deep trees, several important conclusions are drawn. First, in the absence of identical sibling nodes, Constraint Sequencing outperforms RDBQuery for small query sizes when selectivity is low. Like was the case in Chapter 9, an increase in identical sibling nodes shifts preference from Constraint Sequencing to RDBQuery regardless of the selectivity. Second, selectivity is the most important parameter when considering the performance of RDBQuery and comparing it to competing techniques. In particular, the selectivity encountered by the descendant edge queries of

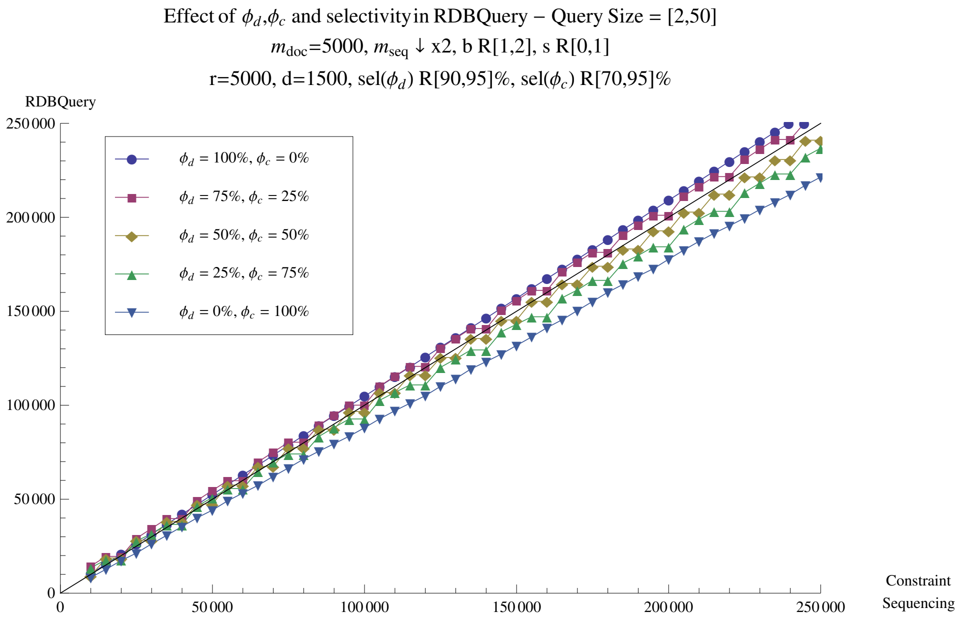


Figure 10.9: RDBQuery, sel(ϕ_d) < sel(ϕ_c) (low selectivity) - Deep

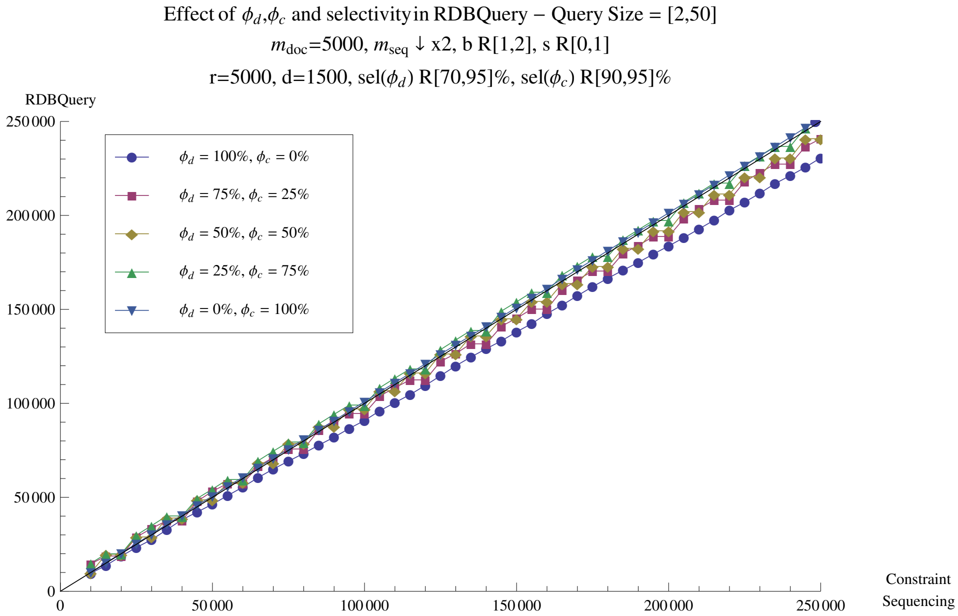


Figure 10.10: RDBQuery, sel(ϕ_d) > sel(ϕ_c) (low selectivity) - Deep

RDBQuery plays the most important role in overall performance of the technique. As selectivity increases, the performance of RDBQuery relative to Constraint Sequencing improves. While it is impossible to group all deep XML trees into a single category, they typically have a low occurrence of identical sibling nodes and reflect a low selectivity for small queries. For these reasons, Constraint Sequencing is the preferred technique when running a small query over a deep XML document. With results that mirror those in Chapter 9, we present similar experiments run on wide trees in the next section.

10.3 Shallow Tree, High Breadth (Wide)

As a contrast to the experiments in the previous section, we now run similar experiments on a wide tree. We use the term wide when referring to a tree with low depth (shallow) that has a high degree of fan-out (high breadth). In Constraint Sequencing, we simulate a wide tree by increasing the branching factor b. With RDBQuery, application of reasoning similar to that used in Section 10.2 tells us that a wide tree exhibits a higher potential selectivity for its nodes. When we consider the shape of a wide tree, it is important to note that, for any given node, the number of descendant nodes is typically less than the number of descendant nodes in a deep tree. This is a direct result of the tree shape. However, to ensure completeness, we do investigate representative cases for selectivity as it changes from high to low.

10.3.1 Experimental Results

In Figure 10.11, we investigate the effect of sequence length m_{seq} in Constraint Sequencing. Note that the branching factor b is increased from that used in Section 10.2 to reflect the increase fan-out of the XML document. It is clear that RDBQuery outperforms Constraint Sequencing regardless of sequence size. Figures H.13, H.14, H.15, and H.16 display a similar trend for various increases and decreases in selectivity and identical sibling nodes. In general, the increase to the branching factor b of Constraint Sequencing causes the technique to perform worse than RDBQuery even when selectivity is low.

A more interesting case is shown in Figure 10.12. For this experiment, we force the branching factor to be large for the first level under a root node and then decrease rapidly (within a single level) to one. This results in an XML document where there is an explosion in breadth then the rest

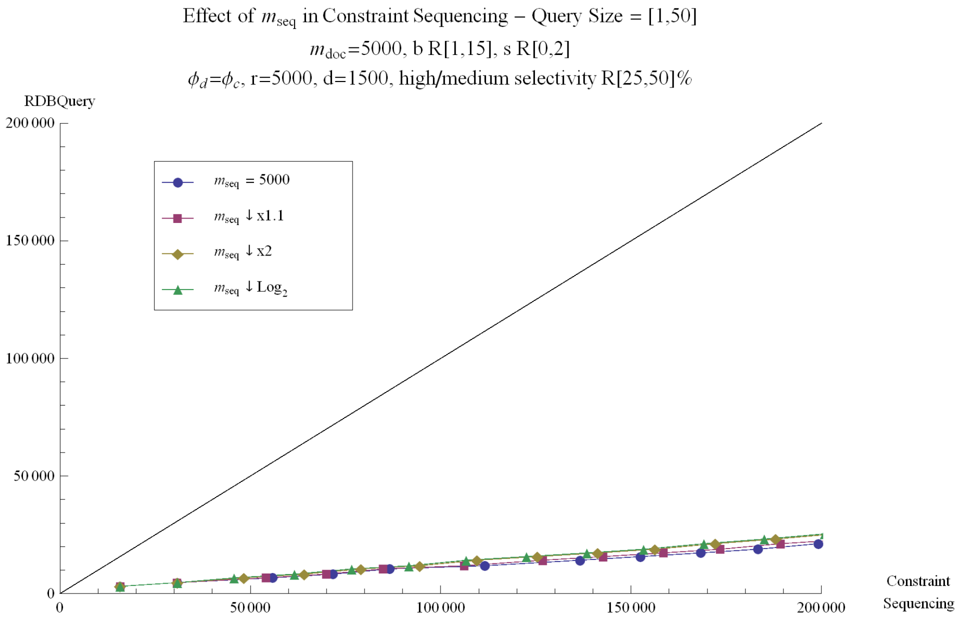


Figure 10.11: CS, vary sequence size (medium/high selectivity) - Wide

of the tree exhibits a deep structure with little increase in width. We note an immediate explosion in the number of operations for Constraint Sequencing, then it quickly exhibits behavior shown for deep trees. However, by the time this occurs (with a larger query size), RDBQuery substantially outperforms Constraint Sequencing to the point where preference for sequencing cannot occur. The same behavior is shown with a higher selectivity in Figure H.17. As we progress through our analysis of wide trees, we continue to use the high/low terminology to refer to an XML document that exhibits this type of structure. Our experimental results with m_{seq} demonstrate the same behavior observed for deep trees. As m_{seq} decreases faster, Constraint Sequencing performance improves, albeit to a lesser extent than seen in deep trees.

Due to the structure of a wide XML document, identical sibling nodes could occur with a greater frequency and in a greater number than with a deep document. Figure 10.13 illustrates the effect of identical sibling nodes on Constraint Sequencing when RDBQuery encounters low selectivity. As expected, Constraint Sequencing performs better with lower s values, but at no time does it outperform RDBQuery. Figures H.18, H.19, H.20, and H.21 in Appendix H exhibit a similar behavior. Even when the branching factor is decreased (Figure H.21), RDBQuery continues to outperform Constraint Sequencing on wider documents. When we allow b to start high then

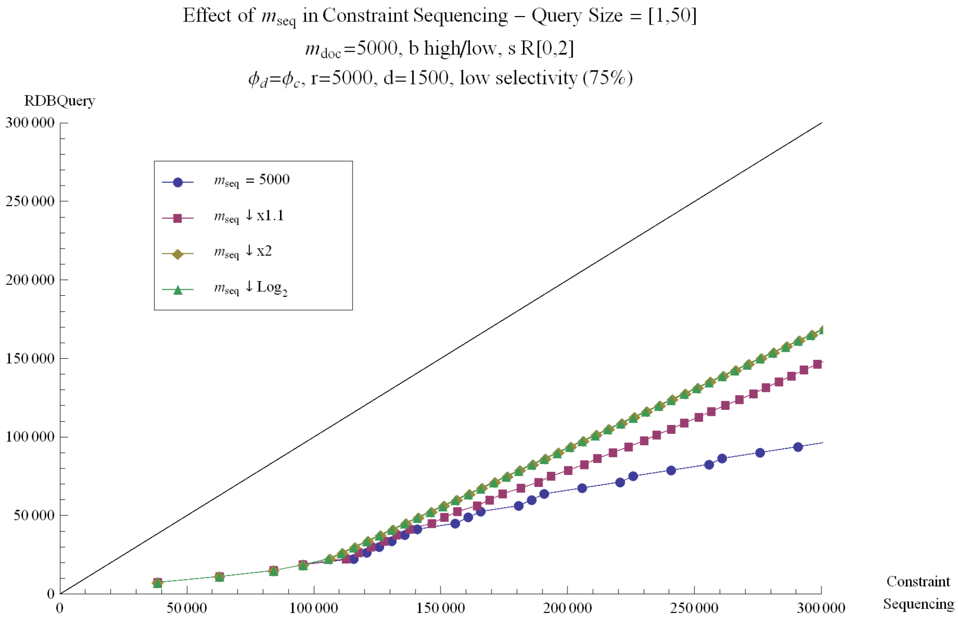


Figure 10.12: CS, vary sequence size (low selectivity, b high/low) - Wide

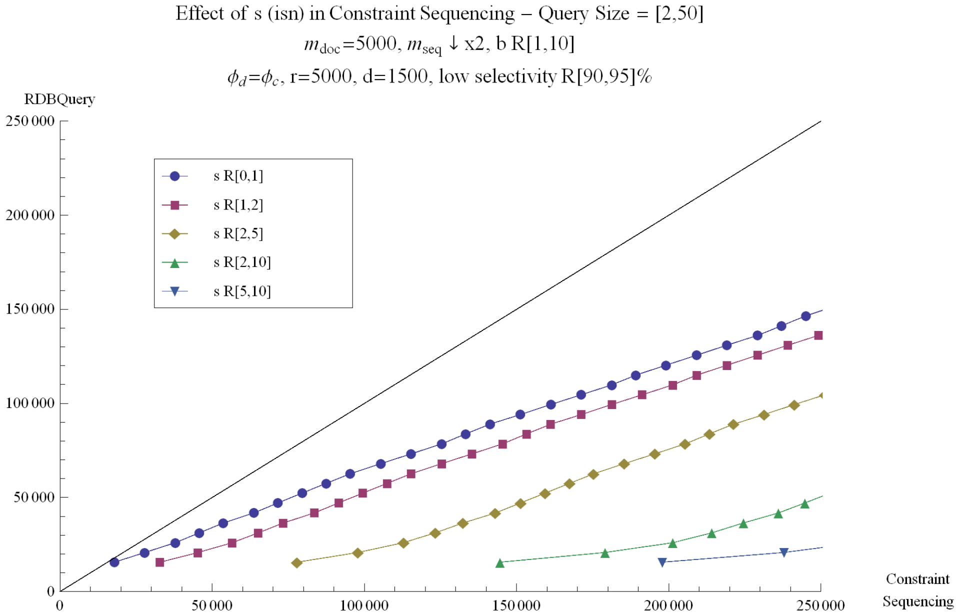


Figure 10.13: CS, random identical sibling nodes (low selectivity) - Wide

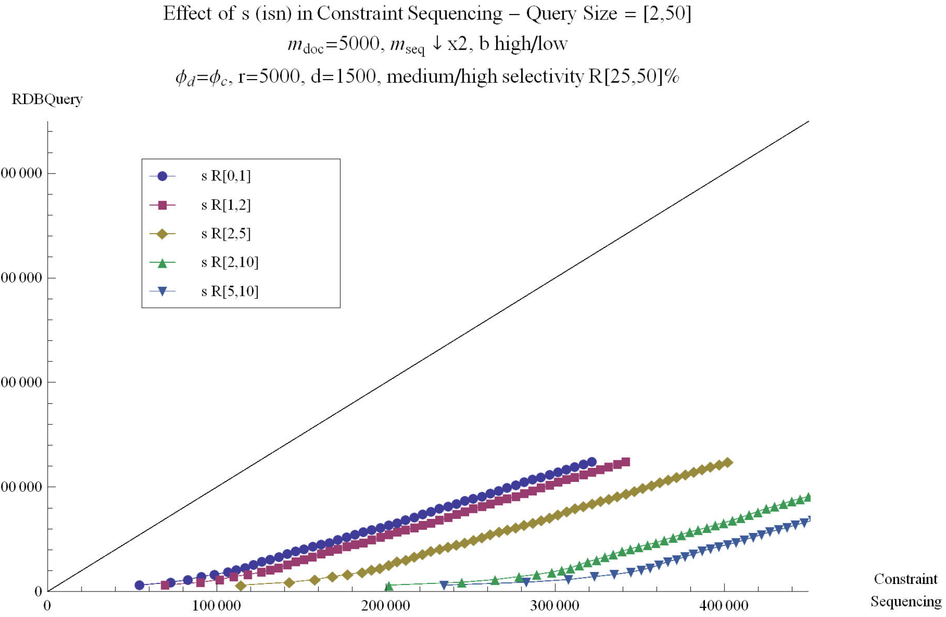


Figure 10.14: CS, random identical sibling nodes (medium/high selectivity, b high/low) - Wide

quickly become low (equal to one), we observe the behavior shown in Figure 10.14. Note that by the time the query size equals seven for a low s value, Constraint Sequencing performance improves significantly. Similar behavior is shown in Figures H.22 and H.23. In Figure H.23, we observe that performance will eventually switch to favor Constraint Sequencing over RDBQuery. This does not happen until the query size exceeds 50. The overall trend in these graphs is that, as s increases, Constraint Sequencing performance decreases. In addition, when it encounters an XML document that exhibits a high fan-out followed by a deep structure, Constraint Sequencing performance does improve but not at a rate high enough to overtake RDBQuery. As was the case for deep trees, the number of distinct values d in RDBQuery has little impact on query performance. These graphs are shown in Appendix H, Figures H.24, H.25, and H.26. The results show a preference for RDBQuery in each case.

Since RDBQuery execution relies heavily on the number of descendant and child edges, we now discuss the effect of their distribution to one other while maintaining a fixed selectivity. Figure 10.16 shows the effects of changing the distribution of ϕ_d and ϕ_c in RDBQuery by the amounts shown in the legend. In contrast to Figure 10.6, we observe that query execution is better for RDBQuery than Constraint Sequencing regardless of the edge distribution in the query. Figure

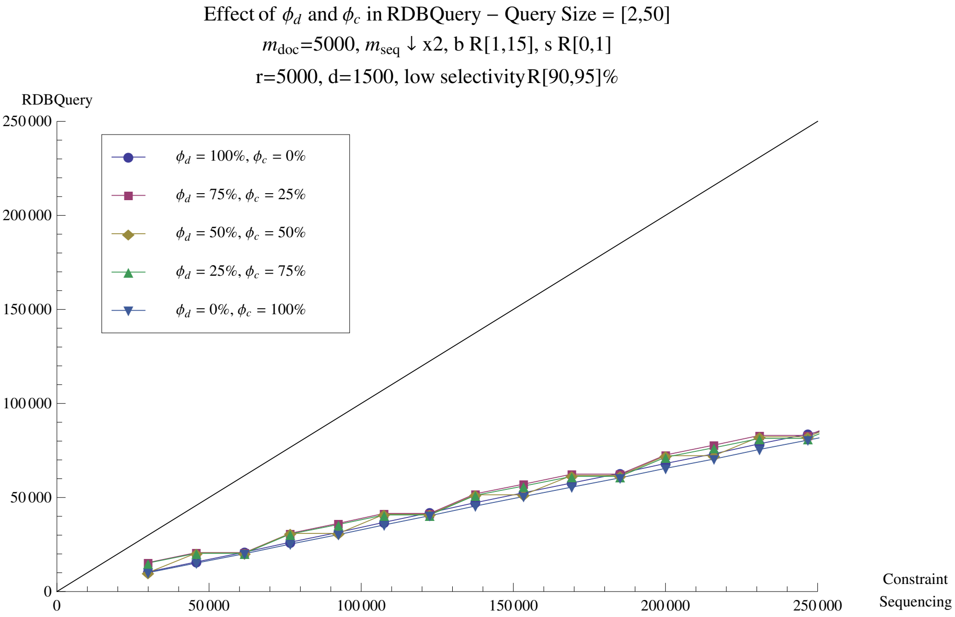


Figure 10.15: RDBQuery, query edge distribution (low selectivity) - Wide

10.16 shows similar behavior and is comparable to Figure 10.8. Increasing the selectivity to a medium/high range results in Figure 10.17. The shift towards the x-axis indicates that RDBQuery performance increases (improves), while the absence of an upward turn in the lines indicate that Constraint Sequencing continues to perform poorly in this situation.

When compared to similar experiments with deep trees, the distribution of descendant and child nodes in a wide tree does not alter query performance to the same extent when selectivity is fixed. Performance is skewed in favor of RDBQuery. Our next set of experiments separates selectivity for the two types of edges, descendant and child.

Figure 10.18 illustrates the same experiment shown for deep trees in Figure 10.9. While the lines trend together, they are pushed much closer to the x-axis and away from the central dividing line. Even for low selectivity, RDBQuery outperforms Constraint Sequencing. The same trend is shown in Figure H.27 but with a more narrow (lower fan-out) XML document. As mentioned earlier, an interesting case presents itself when we have a tree that exhibits a high degree of fan-out at the first level after the root node but then quickly reduces to a deep structure. In Figure 10.19. Notice the initial jump from $q = 2$ to $q = 6$ for all lines. This illustrates the behavior of Constraint Sequencing on an extremely wide document. After that point, performance quickly improves, and increases

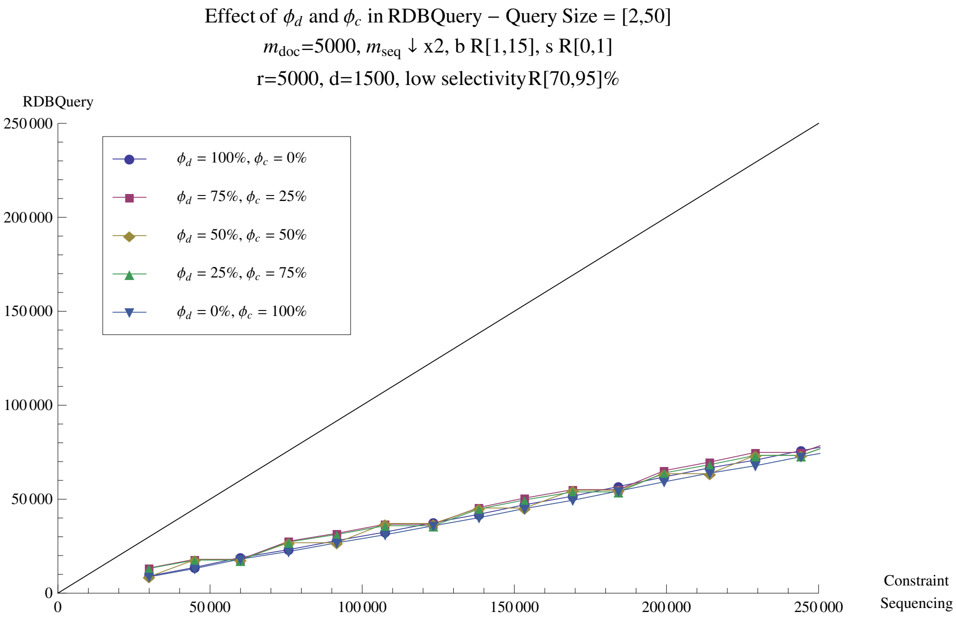


Figure 10.16: RDBQuery, query edge distribution (increased low selectivity) - Wide

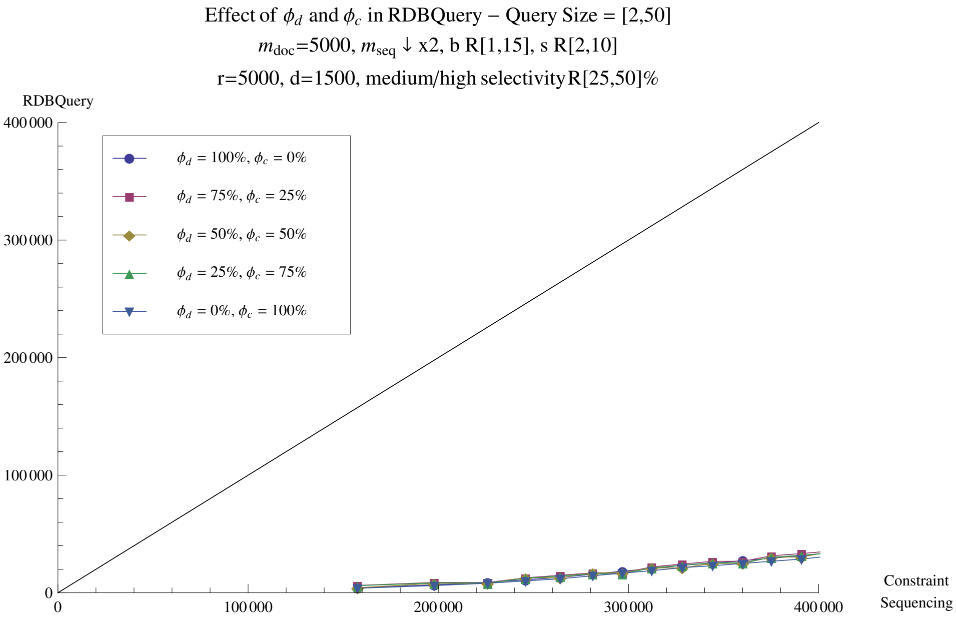


Figure 10.17: RDBQuery, query edge distribution (medium/high selectivity) - Wide

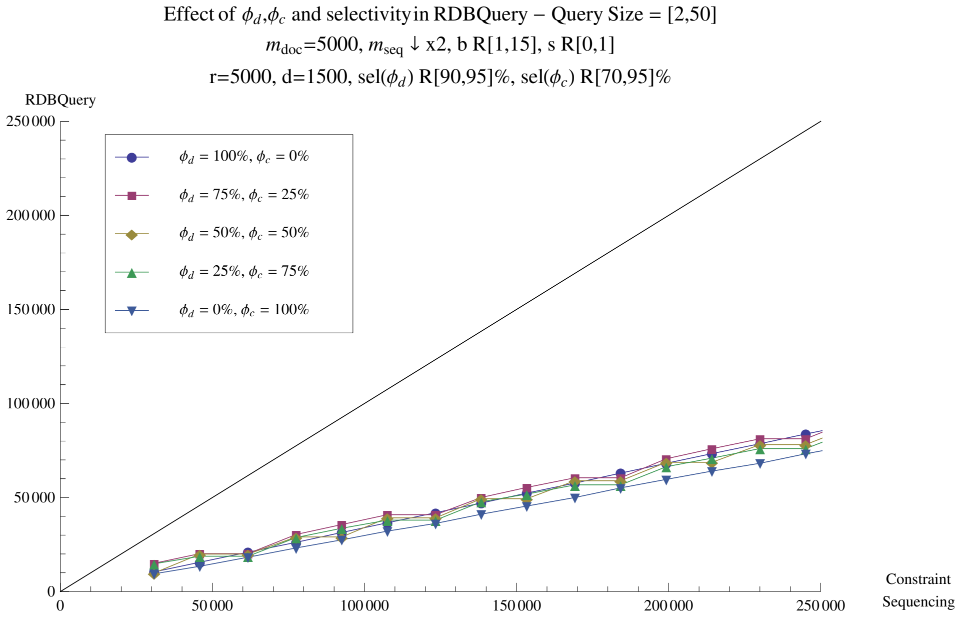


Figure 10.18: RDBQuery, sel(ϕ_d) < sel(ϕ_c) (low selectivity) - Wide

to query size q result in less of an increase in the number of operations Constraint Sequencing must perform. A similar trend is observed throughout the rest of these experiments. Figure 10.20 demonstrates the effects of an increase in selectivity of child edges (ϕ_c). The selectivities for ϕ_d and ϕ_c are reversed in Figure H.28, and results are similar to those shown in Figure 10.18. A similar plot for a lower value of b is shown in Figure H.29, and the case where b is high then immediately low is shown in Figure 10.21. When b is allowed to be high/low as seen in Figure 10.21, we again note the dramatic increase in Constraint Sequencing performance after $q = 5$. Observe that, as selectivity increases, performance continues to shift towards the x-axis (in favor of RDBQuery over Constraint Sequencing). Further examples of this trend are shown in Figures H.30, H.31, H.32, and H.33 in Appendix H.

As a final test scenario, we allow b to take extreme opposite values. At the first level of the document, the branching factor is such that 2/3 (66.6%) of the document nodes are a result of the fan-out from the root node to the next level. After the second level, the document is strictly a deep tree ($b = 1$). This gives us an extremely wide tree that, after its first level, behaves like a deep tree. Figure 10.22 illustrates what happens when Constraint Sequencing and RDBQuery are run over such a document. Note that, while Constraint Sequencing performance starts high (due to

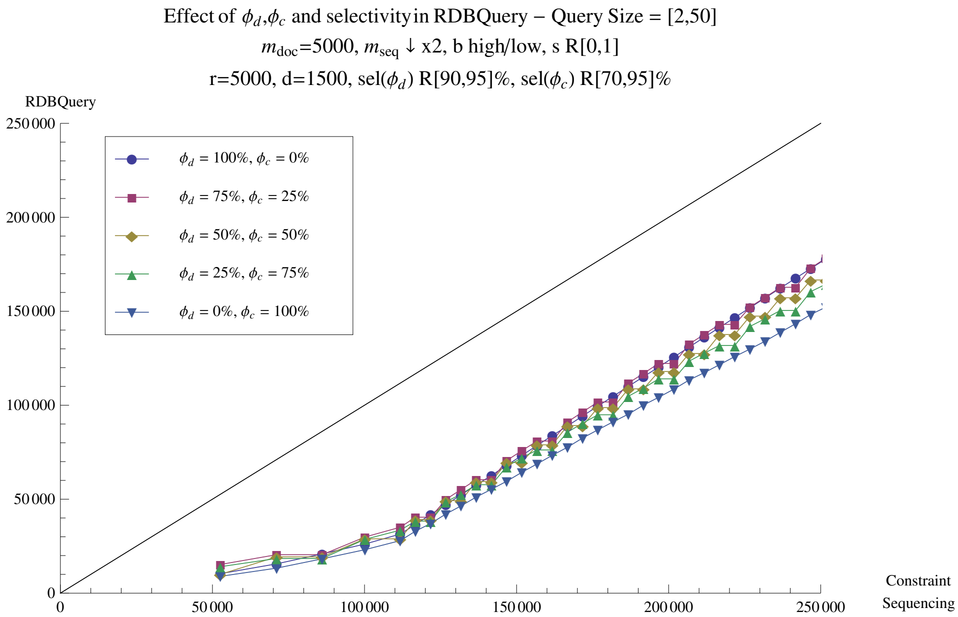


Figure 10.19: RDBQuery, sel(ϕ_d) < sel(ϕ_c) (low selectivity, b high/low) - Wide

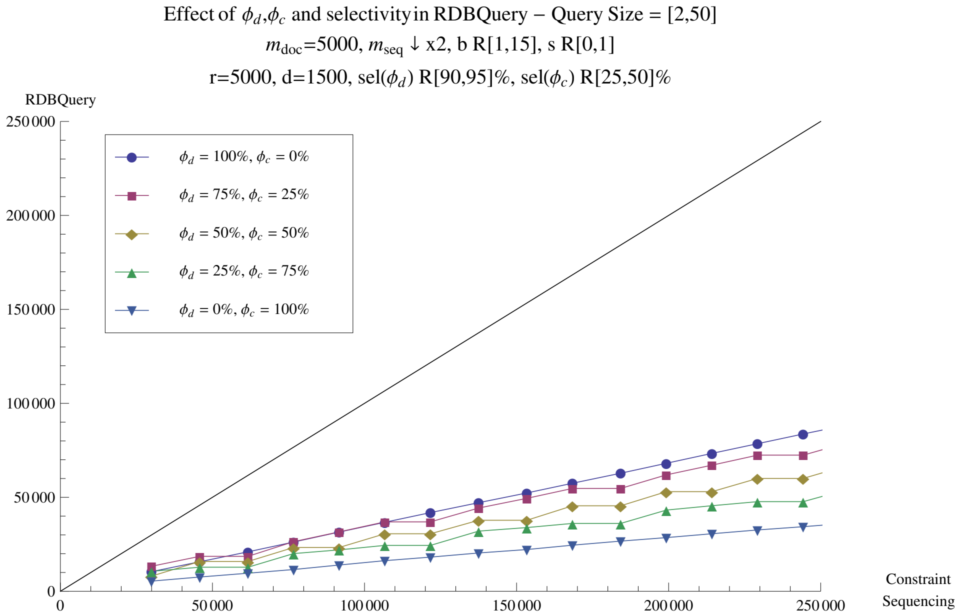


Figure 10.20: RDBQuery, low sel(ϕ_d) < medium/high sel(ϕ_c) - Wide

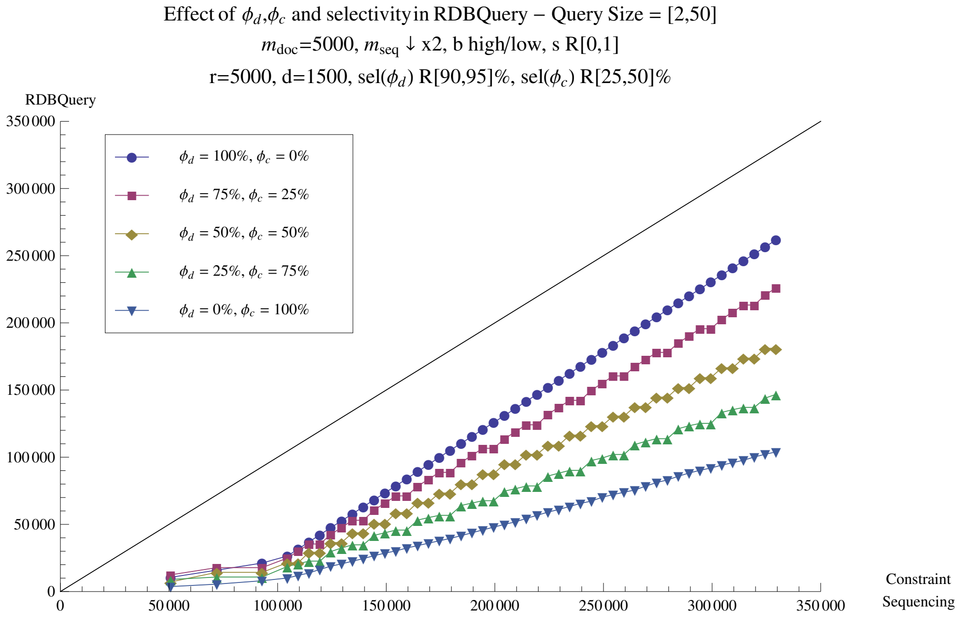


Figure 10.21: RDBQuery, low sel(ϕ_d) < medium/high sel(ϕ_c) (b high/low) - Wide

the high value of b at the first level of the XML document), it increases gradually from that point on. Conversely, RDBQuery starts small and then increases in operations at a much faster rate than Constraint Sequencing. As better seen in Figure 10.23, preference for Constraint Sequencing occurs when query size exceeds a length of 12.

By itself, this is a substantial result and a potential limitation of RDBQuery. However, note that the number of identical sibling nodes in Figures 10.22 and 10.23 is extremely low (randomly generated between zero and one). If an XML document exhibits this type of explosive behavior at the root node, there is the potential for a large number of identical sibling nodes to exist as a result. In Figure 10.24, we increase the number of identical sibling nodes to a maximum of 10 and observe the dramatic shift in Constraint Sequencing performance. The plot only shows results up through $q = 50$, so to notice an increase in performance from RDBQuery to Constraint Sequencing, the query size would need exceed this amount. Since the performance of RDBQuery remains unaffected by identical sibling nodes, preference for RDBQuery over Constraint Sequencing persists.

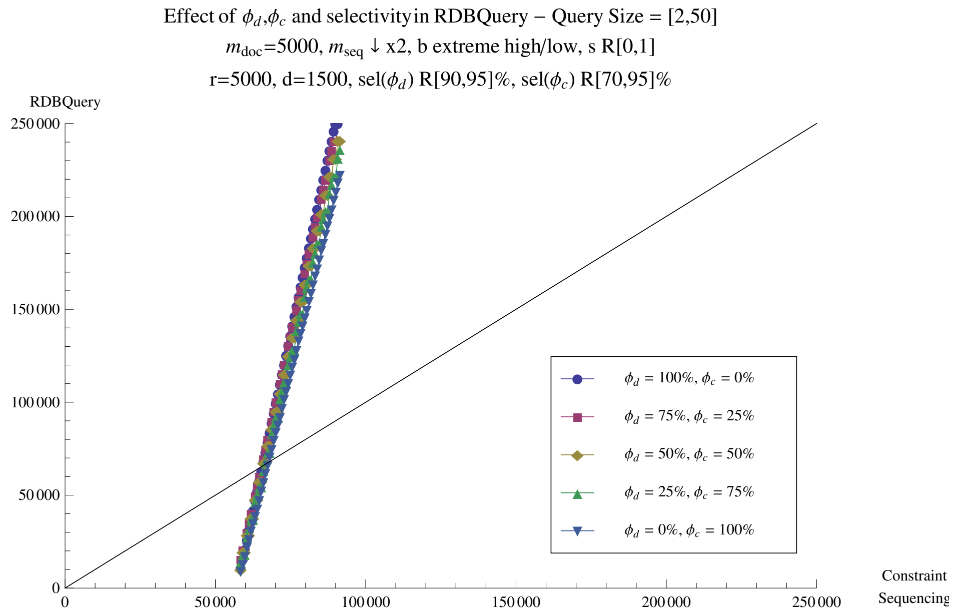


Figure 10.22: RDBQuery, low sel(ϕ_d) < low sel(ϕ_c) (b extreme high/low) - Wide

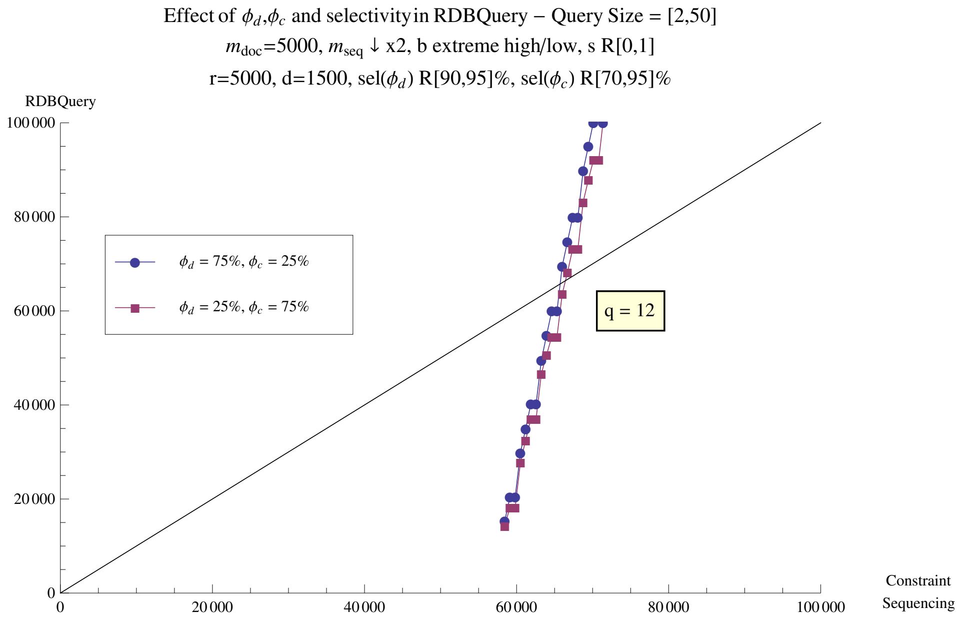


Figure 10.23: RDBQuery, low sel(ϕ_d) < low sel(ϕ_c) (b extreme high/low) - Wide (Zoom)

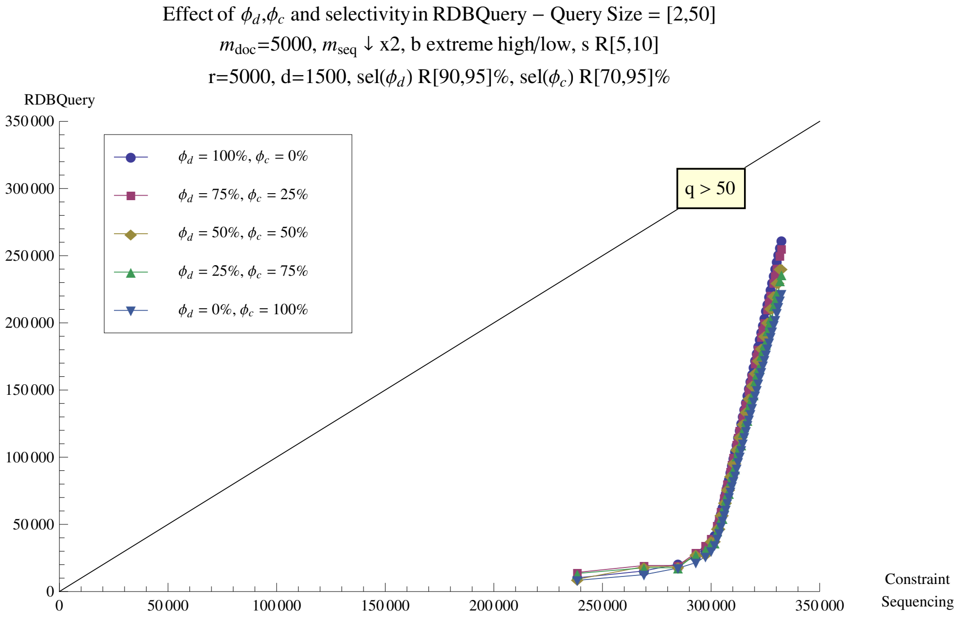


Figure 10.24: RDBQuery, low sel(ϕ_d) < low sel(ϕ_c) (b extreme high/low, increased s) - Wide

10.3.2 Conclusions

From our experimental results that show the performance of Constraint Sequencing and RDBQuery on wide trees, several important conclusions are made. First, the presence of identical sibling nodes continues to be a stumbling block for Constraint Sequencing that can drastically affect query performance. When s increases by a small amount, the ramifications for Constraint Sequencing are substantial. Second, when querying across a wide XML document, RDBQuery outperforms Constraint Sequencing for at least smaller queries. Unless the tree exhibits an extreme branching followed by an immediate switch to an increase in depth only, RDBQuery is preferred over Constraint Sequencing. If, on the other hand, a large query needs to be executed on this same type of tree, Constraint Sequencing outperforms RDBQuery unless the XML document exhibits a slight increase in the number of identical sibling nodes. In that case, RDBQuery, unaffected by identical sibling nodes, performs the query with a smaller number of operations.

10.4 Trees with Similar Depth and Breadth

The performance results presented above were achieved by running a similar set of experiments on XML documents that are relatively deep (Section 10.2) and wide (Section 10.3). As we progressed throughout those sections, some results, such as those with medium selectivity for RDBQuery, partially model a tree that has a similar depth and breadth. To complete our analysis, we perform a small subset of these same experiments on XML documents that have similar depth and breadth. The majority of these results are shown in Appendix I. As a whole, these results favor RDBQuery over Constraint Sequencing, and the experimental results fall between the results for deep trees and those for wide trees. In other words, while query performance is skewed in favor of RDBQuery, the results are closer to the dividing line $y = x$ than those for wide trees.

10.5 DBLP XML Dataset

All of the previous experiments in this chapter were performed on a relatively small synthetic dataset. As in Chapter 9, we utilized the same model of the DBLP dataset [dbl09] in Mathematica. As shown in Section 9.5, Constraint Sequencing scales appropriately when performed on a larger dataset. The same is asserted for RDBQuery. Appendix J contains the results of a limited subset of the experiments found in Section 10.3. While some slight variation may exist in the data, especially in the exact spot where lines cross the $y = x$ dividing line, this can be attributed to minute differences when compiling the random data used for the number of identical sibling nodes, selectivity, and branching factor. The results illustrated throughout Appendix J show that our cost model scales appropriately for larger datasets.

10.6 Overall Conclusions

As was the case in Section 9.6, a definitive answer on if Constraint Sequencing outperforms RDBQuery in all cases is impossible to answer. As we observed, neither technique is preferred in all scenarios with all XML documents. The parameter that detracts the most from Constraint Sequencing's performance, s, can vary widely depending on a specific XML document. Likewise, the selectivity of tuples in the *Edge* table, used by RDBQuery, can be difficult to estimate. However, we observed that RDBQuery outperforms Constraint Sequencing in most cases, and it performs substantially better than Constraint Sequencing on wide XML documents. When Constraint Sequencing is preferred, the margin by which it outperforms RDBQuery is small. This is shown by the graphs in Section 10.2. Also, RDBQuery is shown to scale equal to or better than Constraint Sequencing in most cases, the exception to this being when an XML document exhibits an extreme shift from a high degree of fan-out to no fan-out whatsoever. If we permit the prevalence of wide XML documents over deep XML documents in use as of this writing to influence our decision, RDBQuery is a better choice over Constraint Sequencing. However, to use this technique the structure of the original XML document is lost when it is shred into the *Edge* table. The original document can be reconstructed from the *Edge* table, but this requires additional processing not considered in our analyses. Typically, when the XML document is shred, the original document is no longer updated and otherwise maintained. We also encounter the necessary overhead of a relational database management system. However, the clear dominance of RDBQuery over Constraint Sequencing in the vast majority of test cases presented allows us to choose it as the technique that outperforms Constraint Sequencing.

With our cost models and analyses complete, we present our conclusions and a framework for selecting an optimal XML query technique in Chapter 11. At the end of that same chapter, we discuss future work and open questions.

Chapter 11

Conclusions and Future Work

In this chapter, we summarize our observations and results from Chapters 7 through 10 as a framework for cost-based query optimization. We also discuss areas of future work and opportunities for further investigation.

11.1 Conclusions

The cost models we developed for TwigStack [BKS02], Constraint Sequencing [WM05], SS-Join [SLFW05], and our own technique, RDBQuery, allowed us to create and execute an extensive performance study across all parameters of each technique. In order to develop those models, we unified the techniques with a previously absent common terminology. The results of our study are visualized in Figure 11.1 as a decision framework graph. This framework allows us to arrive at an optimal XML query technique (shown at the bottom of the figure) by making decisions about features of the XML document, XML query, and/or shredded relation (in the case of non-native techniques). In Figure 11.1, nodes (with the exception of the bottom row) represent aspects upon which a decision is made. Directed lines leaving those nodes are labeled with an answer that either leads to another decision or a query technique. The nodes shown at the bottom of the figure (with no outgoing arcs) represent the four XML query techniques studied in this dissertation. As we discuss the decision graph, we use the terms *top* and *bottom* to refer to the appropriate areas of the graph when visualized in its current form (as shown). In addition, the term *user* refers to a human user or an automated system based on the framework. We continue to use the terms *wide* and *deep*

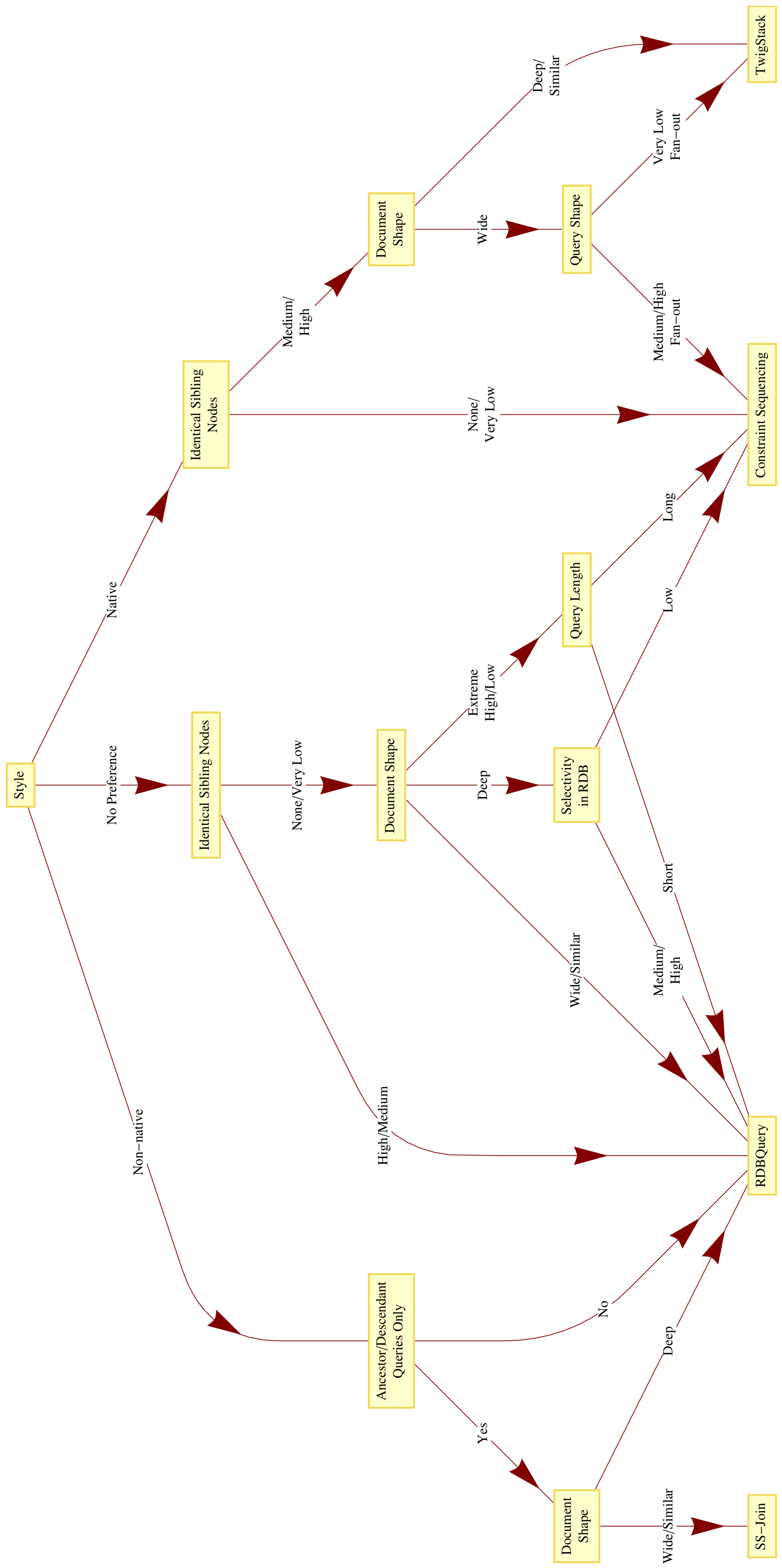


Figure 11.1: XML Cost-based Optimization Framework

to describe trees/documents that have a high degree of fan-out and little fan-out, respectively. The term *similar* is used to describe a structure that is similarly deep and broad. At the top of Figure 11.1, the user is presented with a choice of techniques. If they prefer or need a native technique (TwigStack and Constraint Sequencing) or non-native technique (SS-Join and RDBQuery), they proceed down the appropriate path. If they have no preference, the user proceeds down the center. Each of these choices is discussed in their own section.

11.1.1 Non-Native Preference

If the user prefers a non-native technique, they are then asked if they will run ancestor/descendant queries exclusively. If this is the case, then SS-Join is preferred over RDBQuery for wide/similar document shapes. The justification for this is found in the analysis of RDB techniques found in Chapter 8. If the user wishes to perform any other type of queries, such as a twig query, then RDBQuery must be used since SS-Join does not support these types of queries. Then, as discussed in Chapter 8, SS-Join is preferred over RDBQuery when the size of the ancestor and descendant lists are small. This is reflected in by a wide or similar document shape. For deep document structures, RDBQuery is preferred over SS-Join, regardless of the type of query performed.

11.1.2 Native Preference

The path to the right from the *Style* node illustrates the necessary process if the user prefers a native query technique. As shown in Chapters 7 and 9, Constraint Sequencing outperforms TwigStack in the absence of identical sibling nodes. Constraint Sequencing also is preferred if there exists a very low amount of these sibling nodes. A hard number is difficult to specify, but the number of identical sibling nodes should represent less than 1% of the total nodes in the document. If more identical sibling nodes exist, then we must consider the shape of the document. For deep and similar documents, TwigStack is the preferred method. In the case of wide documents, the shape of the query is important in the determination of a preferred query method. For queries that exhibit very low fan-out, TwigStack outperforms Constraint Sequencing. This is illustrated by the experiments shown in Section 9.3. To be considered very low, the average fan-out of a query should be less than three. In the case of a simple twig query, the fan-out averages less than two, so TwigStack would be preferred. If the query fan-out increases, Constraint Sequencing outperforms

TwigStack.

11.1.3 No User Preference

The center path is selected when the user presents no preference to query technique style (native or non-native). The result of this decision is the selection of either RDBQuery or Constraint Sequencing as the preferred query technique. The options along this path are supported by the experiments shown in Chapter 10. As expected, if the number of identical sibling nodes are medium or high, RDBQuery is preferred over Constraint Sequencing. The same rough guide of no more than 1% of the total nodes as identical sibling nodes can be applied in this case. If there are few or no identical sibling nodes, then we must examine the document shape. If the document is wide or similar, RDBQuery is preferred. However, there is an important special case for when the document exhibits an extreme high/low structure. As an example, if a document has an extremely high fan-out after the root node but then scales back to no fan-out, the document would be classified as extreme high/low. If this accurately describes the original document, then we must examine the average query length. If it is short (less than 15 nodes in length), then RDBQuery outperforms Constraint Sequencing. Conversely, long queries favor Constraint Sequencing. The extreme high/low case could also be modeled as a special case of the wide/similar document shape, but for clarity we allow it to have its own path. For a deep document, we need to know something about the selectivity in the relational database. Recall from Section 10.2 that if selectivity is low, RDBQuery returns larger results and performance shifts in favor of Constraint Sequencing. If selectivity will be medium or high, then RDBQuery performs as well as Constraint Sequencing and will outperform it as selectivity increases.

11.1.4 Contributions

The decision framework shown in Figure 11.1 serves as a visual representation of our unique contributions in this dissertation. We began by performing a literature survey for applicable XML query techniques. After the search, we classified the techniques according to their style of operation and selected a representative technique from the two main categories, native and non-native, that performed as well as or better other techniques in the area. All of the techniques we surveyed exhibited different terminology and were not compared to other representative techniques from other areas. We provided a unifying terminology for comparing XML query techniques and developed a mathematical cost model for each. In the process, we noted the limitations of the representative relational database technique, SS-Join, and used this as motivation to create our own technique, RDBQuery, that leverages some of the ideas in SS-Join and builds upon them.

We used our four cost models to conduct a performance study of each individual technique across all of its salient parameters. During this process, we noted factors that significantly impact query performance and used those to determine which technique from each area outperformed the other. For the non-native techniques, we selected RDBQuery since it does not suffer from the query style limitations that SS-Join has. We then employed a similar technique to compare the representative native technique, Constraint Sequencing, to the representative non-native technique, RDBQuery. Using all of our results, we created a framework for cost-based optimization and presented the results as a framework that a user, human or machine, could employ. Our framework is based on 322 experiments created and executed in Mathematica, and 270 of those unique experiments are included in this dissertation.

11.2 Future Work

An immediate area for future work is to implement RDBQuery and test the technique with real data that has not been mathematically modeled. This would allow us to further verify the validity of our experimental results. This not only requires a significant amount of coding to implement, but decisions would need to be made regarding the relational database management system used and how to include descendant/child edge information within the query. One possible solution would use something similar to a graphical query designer found in most commercial relational database packages to construct the query. The user could then specify a descendant or child edge when creating the query.

We focus our efforts on query execution and the number of operations required to perform the XML query. Another consideration is one of space requirements for each of the four techniques presented here. In TwigStack, we would require space to hold the entire XML document, a similar amount of space to hold the streams T that act as an index into the document for similar nodes, a small amount of space to hold the query, and space for the stacks. The space necessary for the stacks would depend on the number of results, both final and intermediate, returned by TwigStack.

For a larger result set, more space would be required since the stacks would be larger. In Constraint Sequencing, we require space for the XML document, a small amount of space for the query itself, and space for an index that is used for identical sibling nodes. The space required for the sibling node index could be small if the index was implemented using an efficient structure such as a B^+-tree. From our rough estimates, Constraint Sequencing appears to require less space than TwigStack, and this is substantiated by the original authors of the work [WM05]. For the non-native techniques SS-Join and RDBQuery, space for the relational database is required. This could be potentially large (at least as large as the number of nodes in the document). Both techniques would also require a small amount of space to hold the query. Since SS-Join does not make use of the RDB index, it is not required for this technique. However, it may be difficult to locate a commercial RDBMS package that allows indexing as an option to turn off. RDBQuery requires these indices and makes frequent use of them. In a study similar to that presented in this dissertation, cost models for space requirements could be formulated and a similar performance study completed. The results of that study, combined with our own results, would provide a finer granularity of decision-making in selecting the appropriate XML query technique. Once an appropriate space cost-model framework is established, it could be used in conjunction with our results to create a tool that fully automates XML query technique selection.

An additional area for future work considers the impact of database updates (modifications to the original XML document). As the techniques exist in their current form, TwigStack and Constraint Sequencing do not provide an efficient method for updating (relabeling) the XML document. In both cases, the entire document would need to be relabeled if a sufficient amount of new nodes were inserted. The two non-native techniques, SS-Join and RDBQuery, allow for an efficient ($O(\log_2(n))$) relabeling technique [SLFW05] that need not relabel the entire *Edge* table. One possible direction for future work would be to investigate if this relabeling technique could be extended to native XML documents.

Bibliography

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web*. Morgan Kaufmann Publishers Inc., 2000.

[ANS86] ANSI. American National Standards Institute: The database language SQL. Document ANSI X3.135, 1986.

[BCD$^+$02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*, volume 2461, pages 152–164, 2002.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of data*, pages 310–321, Madison, Wisconsin, USA, June 3-6 2002.

[CBB$^+$97] R. G. G. Cattell, Douglas K. Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CVZT02] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, and Vassilis J. Tsotras. Efficient structural joins on indexed XML documents. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 263–274, Hong Kong, China, August 20-23 2002.

[dbl09] *The DBLP Computer Science Bibliography*. http://dblp.uni-trier.de/, October 1 2009.

[Die82] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 122–127, San Francisco, CA, United States, May 5-7 1982.

[EN00] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 3rd edition, 2000.

[GC07] Gang Gou and Rada Chirkova. Efficiently querying large XML data repositories: A survey. *Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, October 2007.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 436–445, San Francisco, CA, USA, August 25-29 1997.

[HBG+03] Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy, Ajith Nagaraja Rao, Feng Tian, Stratis D. Viglas, Yuan Wang, Jeffrey F. Naughton, and David J. DeWitt. Mixed mode XML query processing. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 225–236, Berlin, Germany, September 9-12 2003.

[HR05] Su Cheng Haw and G. S. V. Radha Krishna Rao. Query optimization techniques for XML databases. *International Journal of Information Technology*, 2(1):97–104, 2005.

[JLWO03] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, pages 253–263, Bangalore, India, March 5-8 2003.

[JWLY03] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 273–284, Berlin, Germany, September 9-12 2003.

[LLHC04] Hanyu Li, Mong-Li Lee, Wynne Hsu, and Chao Chen. An evaluation of XML indexes for structural join. *SIGMOD Record*, 33(3):28–33, September 2004.

[LM01] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 361–370, San Francisco, CA, United States, September 11-14 2001.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proceedings of the 11th International Conference on Data Engineering (ICDE'95)*, pages 251–260, Taipei, Taiwan, March 6-10 1995. IEEE Computer Society.

[RM01] Flavio Rizzolo and Alberto O. Mendelzon. Indexing XML data with ToXin. In *Proceedings of the 4th International Workshop on the Web and Databases (WebDB'2001)*, pages 49–54, Santa Barbara, CA, USA, May 24-25 2001.

[RM04] Praveen Rao and Bongki Moon. PRIX: Indexing and querying XML using prfer sequences. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, pages 288–300, Washington, DC, United States, 2004 2004.

[SLFW05] William M. Shui, Franky Lam, Damien K. Fisher, and Raymond K. Wong. Querying and maintaining ordered XML data using relational databases. In *Proceedings of the 16th Australasian Database Conference (ACD'05)*, volume 39, pages 85–94, Newcastle, Australia, 2005 2005. Australian Computer Society, Inc.

[tre09] *The Penn Treebank Project (University of Pennsylbania, USA).* http://www.cis.upenn.edu/ treebank/, October 1 2009.

[TVB$^+$02] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conferende on Management of Data (SIGMOD'02)*, pages 204–215, Madison, WI, United States, June 4-6 2002. ACM.

[WM05] Haixun Wang and Xiaofeng Meng. On the sequencing of tree structures for XML indexing. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 372–383, Tokyo, Japan, April 5-8 2005.

[WPFY03] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *Proceedings of the 2003 ACM SIGMOD International Conferende on Management of Data (SIGMOD'03)*, pages 110–121, New York, NY, United States, 2003 2003.

[xml] *Extensible Markup Language (XML).* http://www.w3.org/XML/.

[xml09] *XML Data Repository (University of Washington, USA).* http://www.cs.washington.edu/research/xmldatasets/, October 1 2009.

[xpa09] *XPath 2.0.* http://www.brics.dk/ amoeller/XML/linking/xpath20.html, July 30 2009.

[xqu09] *XQuery 1.0: An XML Query Language.* http://www.w3.org/TR/2005/CR-xquery-20051103/, July 30 2009.

Appendices

Appendix A

TwigStack Graphs

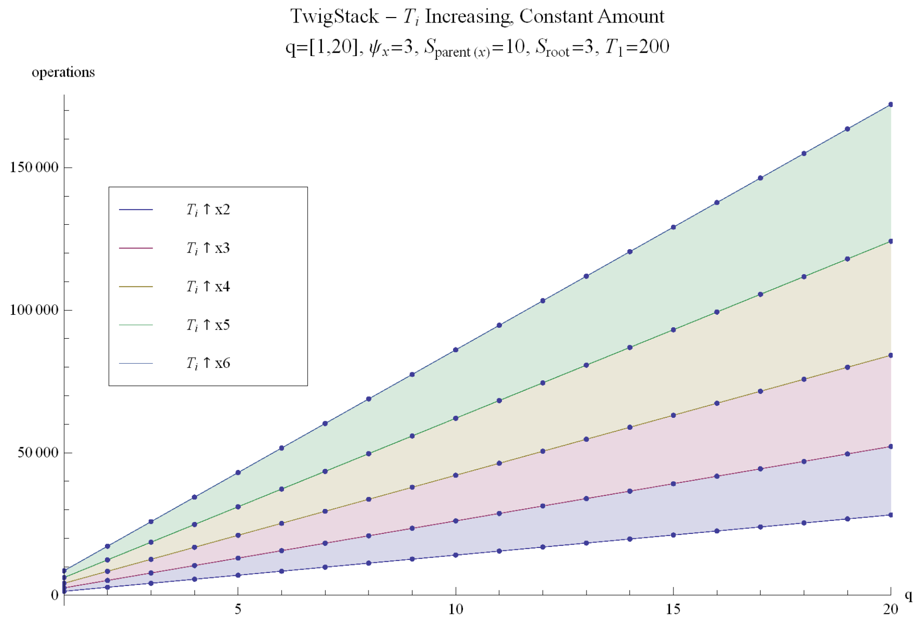


Figure A.1: TwigStack, stream size increasing by constant factors, low base case

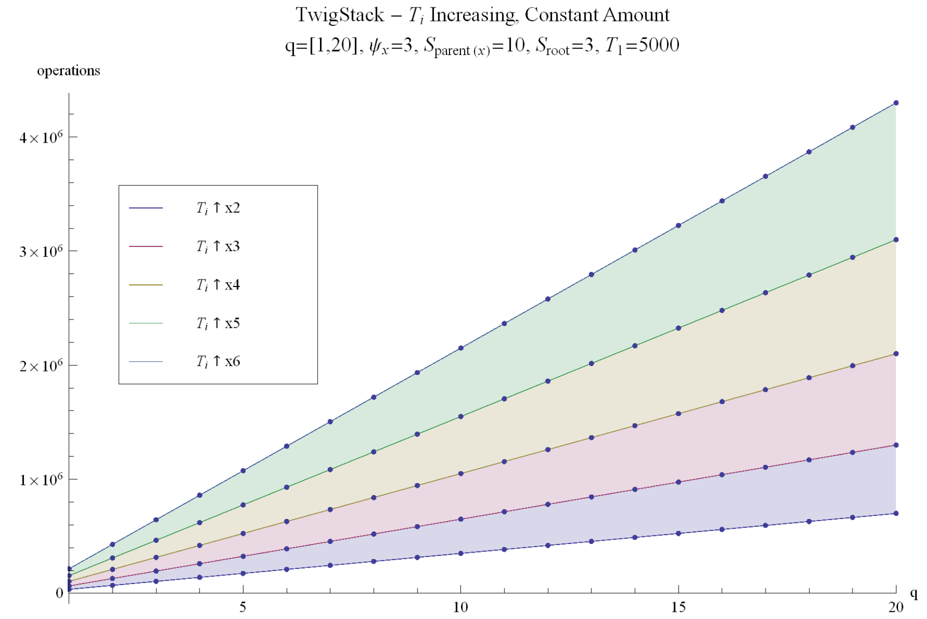


Figure A.2: TwigStack, stream size increasing by constant factors, high base case

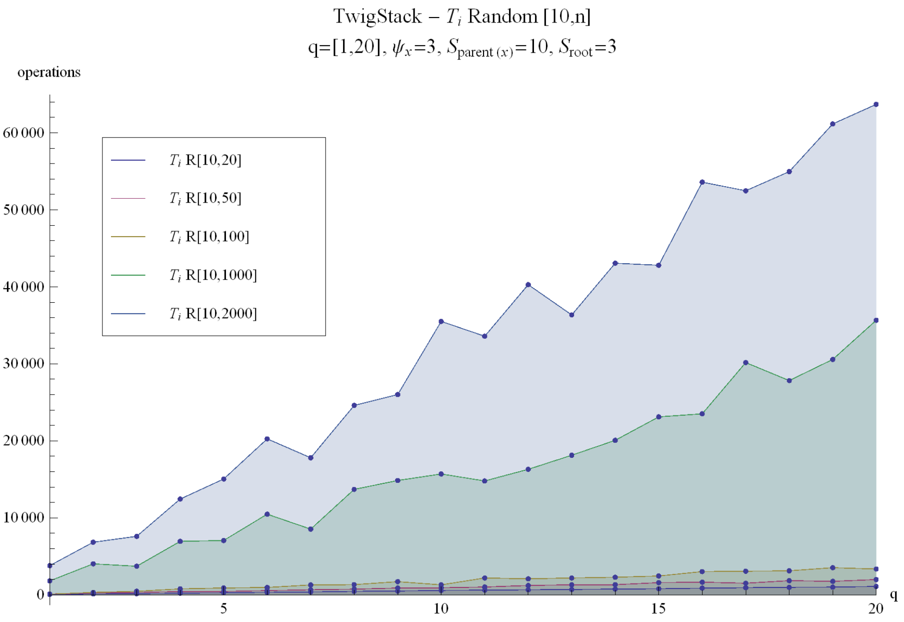


Figure A.3: TwigStack, random stream sizes - single run

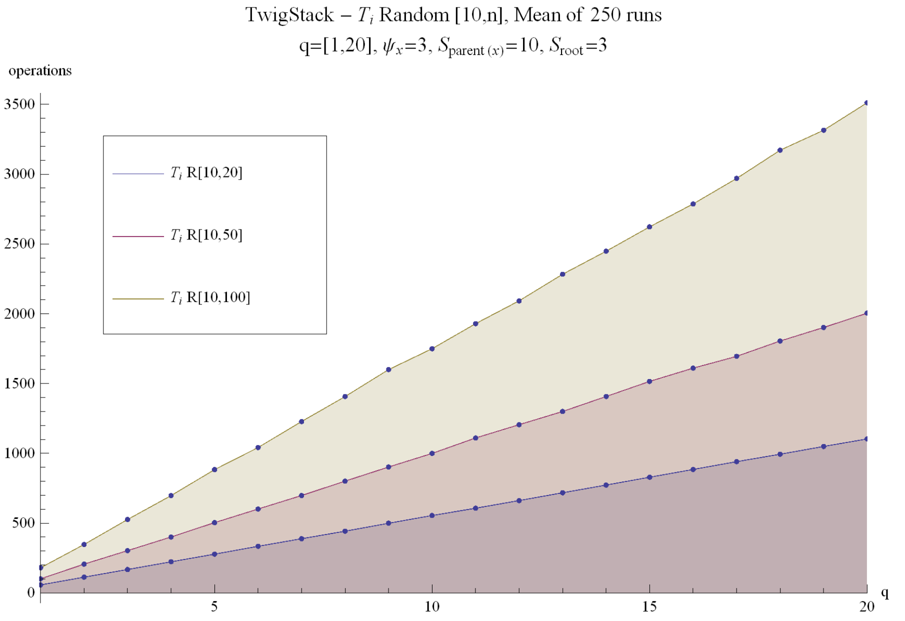


Figure A.4: TwigStack, small random stream sizes - 250 runs

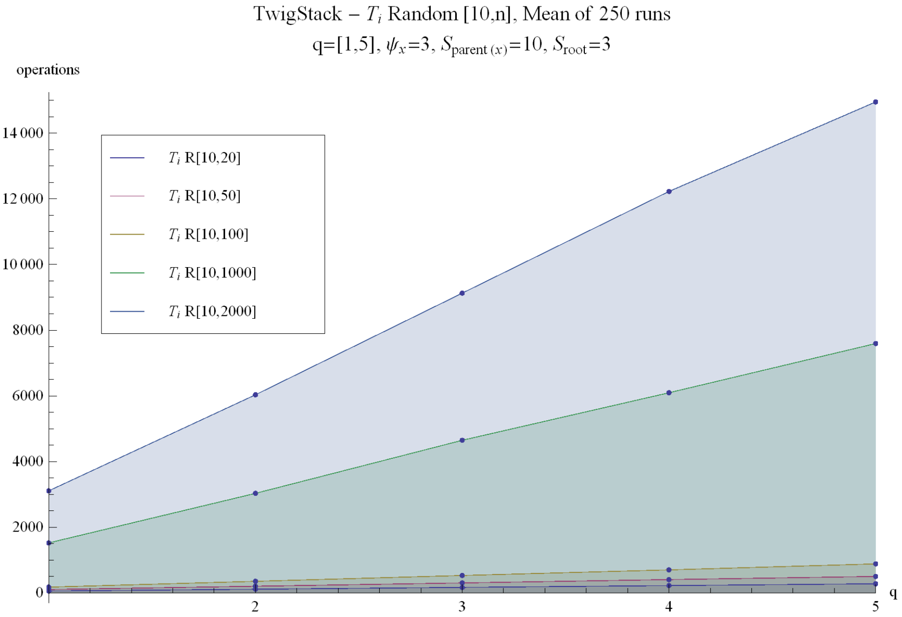


Figure A.5: TwigStack, random stream sizes (smaller query) - 250 runs

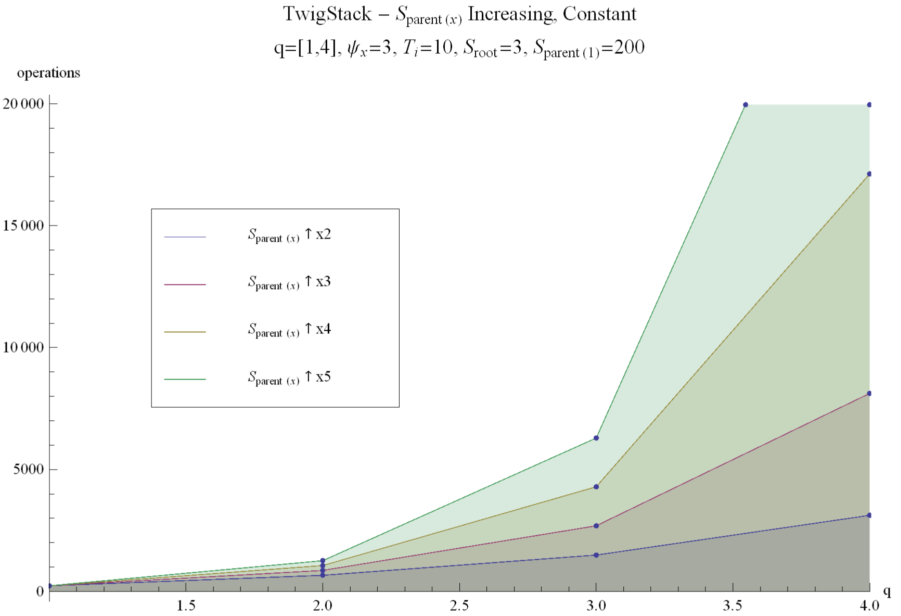


Figure A.6: TwigStack, stack size increasing by constant factors ($S_{parent(1)} = 200$)

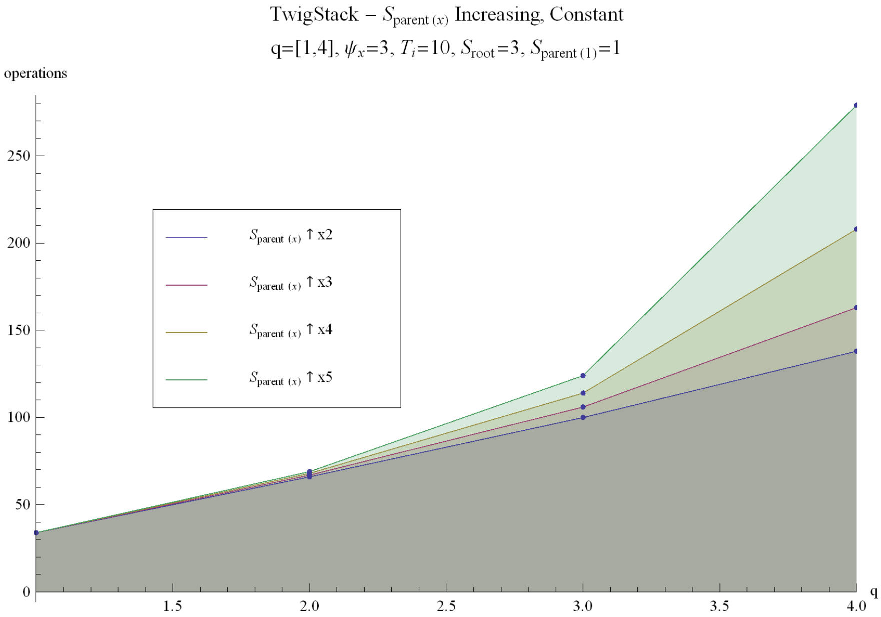


Figure A.7: TwigStack, stack size increasing by constant factors ($S_{parent(1)} = 1$)

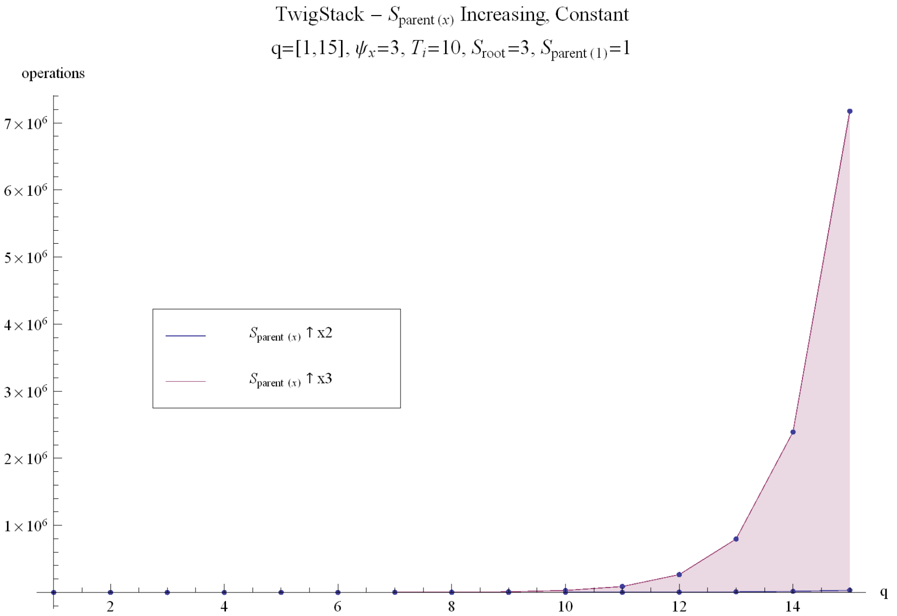


Figure A.8: TwigStack, stack size increasing by constant factors ($S_{parent(1)} = 1$, medium query)

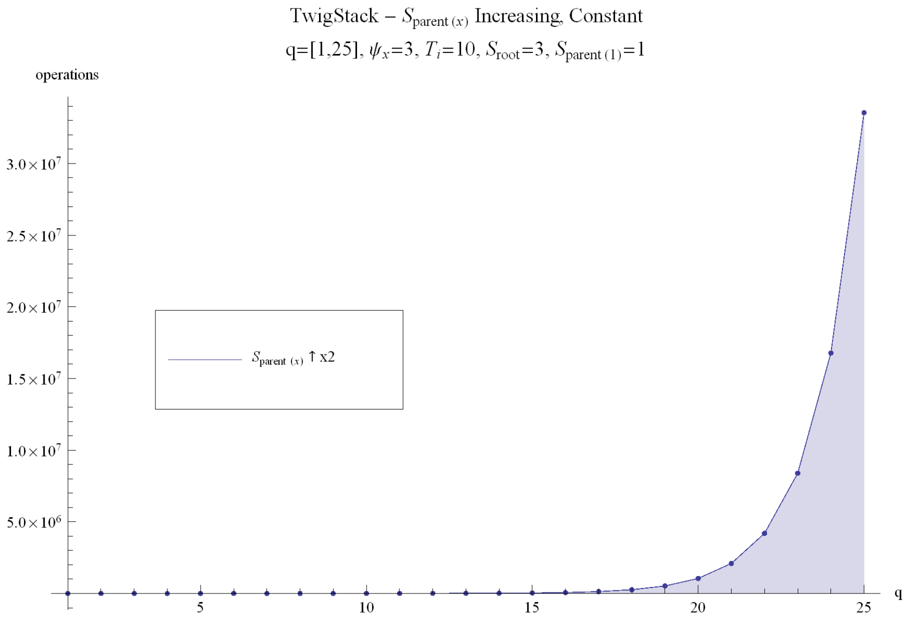


Figure A.9: TwigStack, stack size increasing by constant factors ($S_{parent(1)} = 1$, larger query)

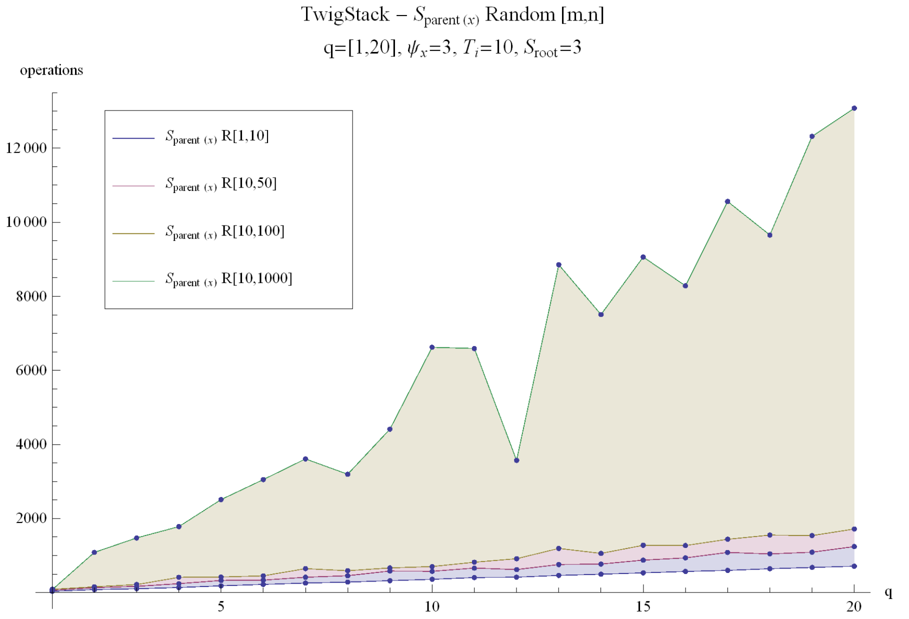


Figure A.10: TwigStack, random stack size up to 1000 - single run

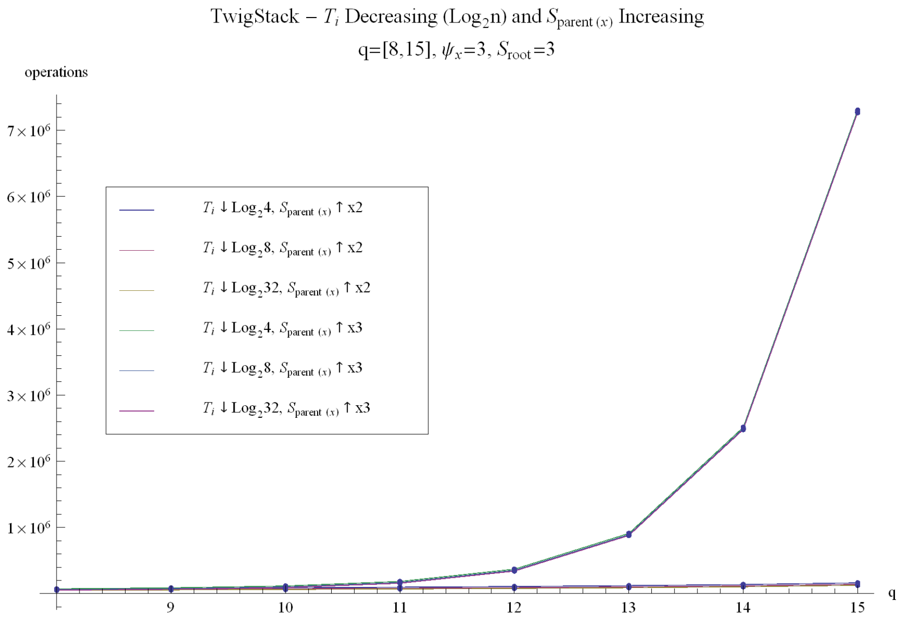


Figure A.11: TwigStack, stream size decreasing and stack size increasing (larger query)

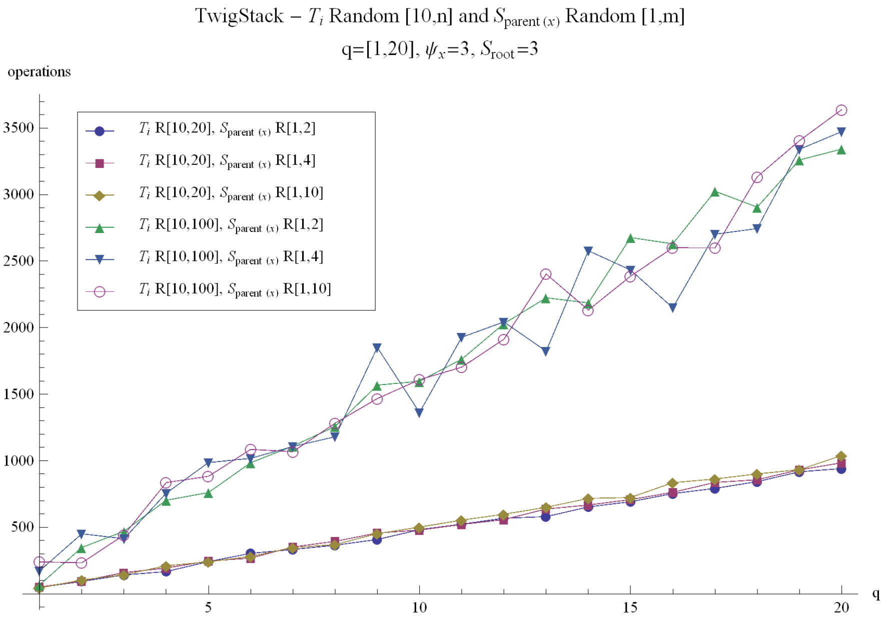


Figure A.12: TwigStack, random stream and stack sizes - single run

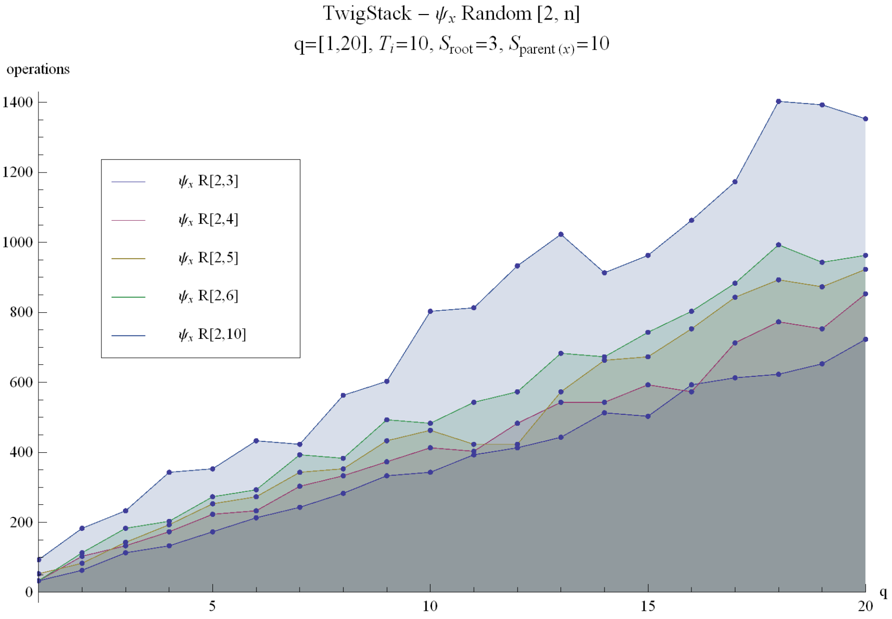


Figure A.13: TwigStack, random query fan-out - single run

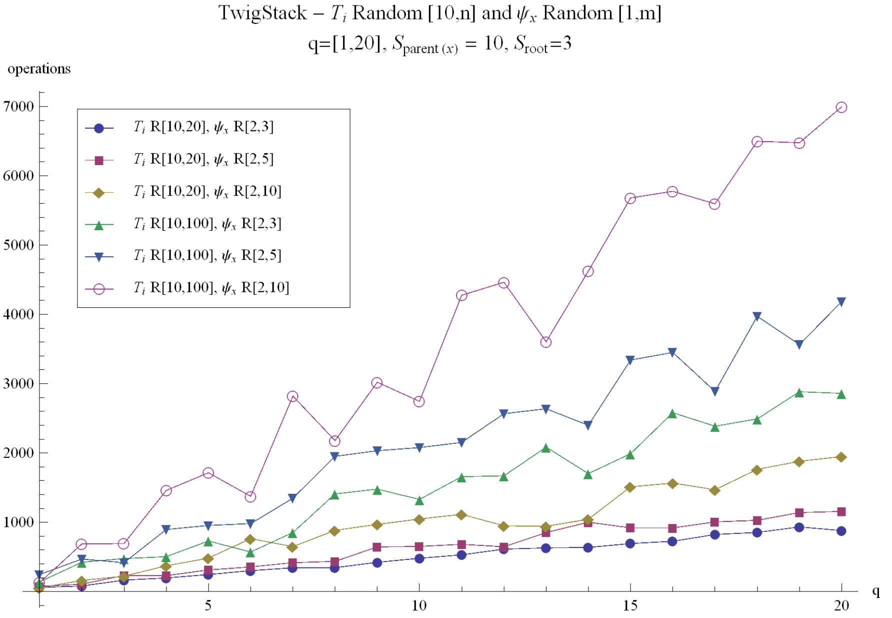


Figure A.14: TwigStack, random stream sizes and query fan-out - single run

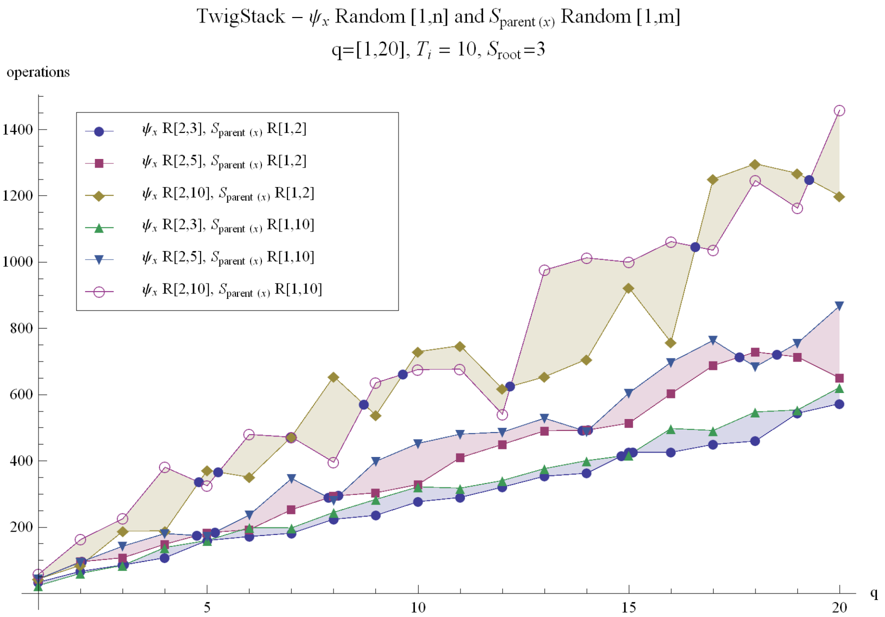


Figure A.15: TwigStack, random query fan-out and stack sizes - single run

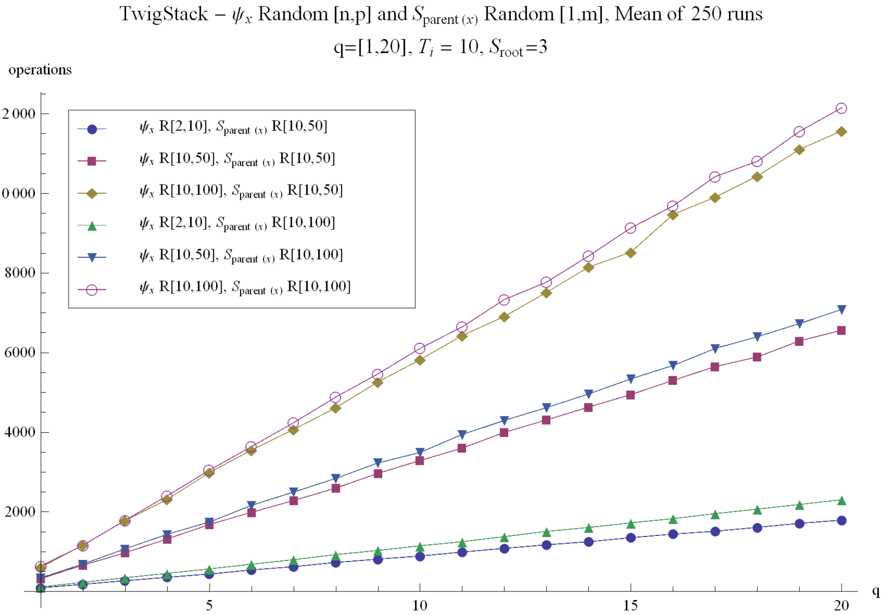


Figure A.16: TwigStack, random query fan-out and stack sizes (larger stacks) - 250 runs

Appendix B

Constraint Sequencing Graphs

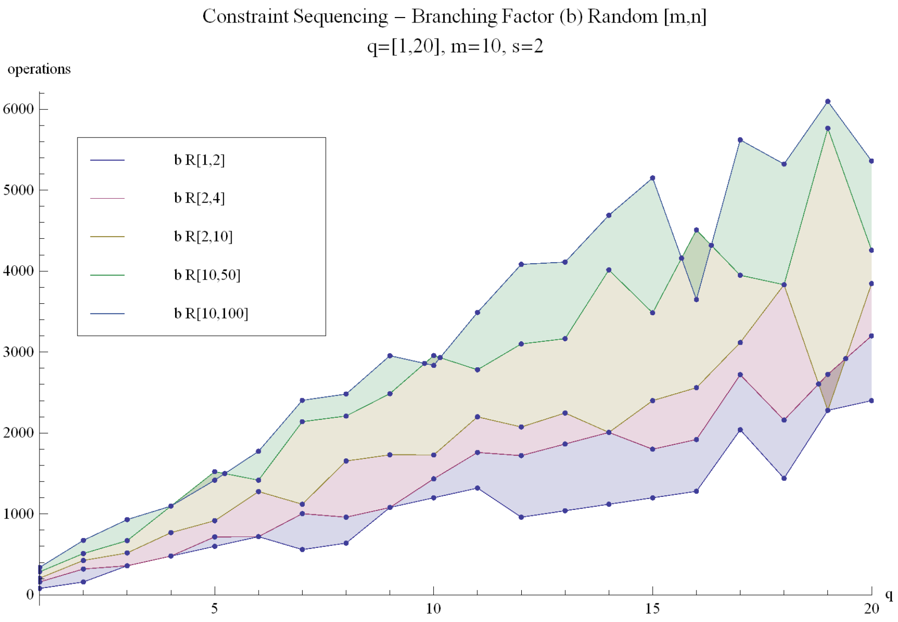


Figure B.1: Constraint Sequencing, random branching factor - 250 runs

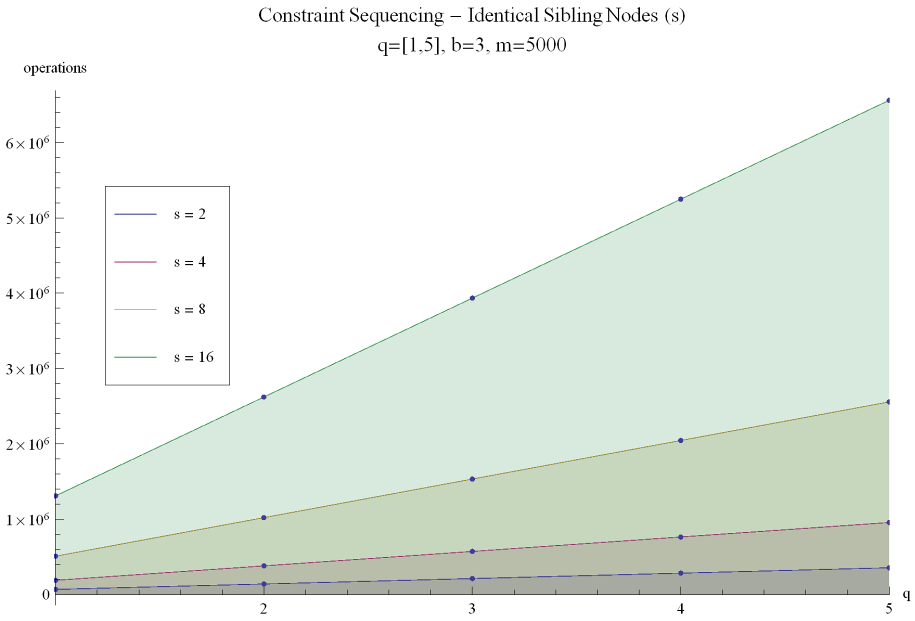


Figure B.2: Constraint Sequencing, identical sibling nodes increasing

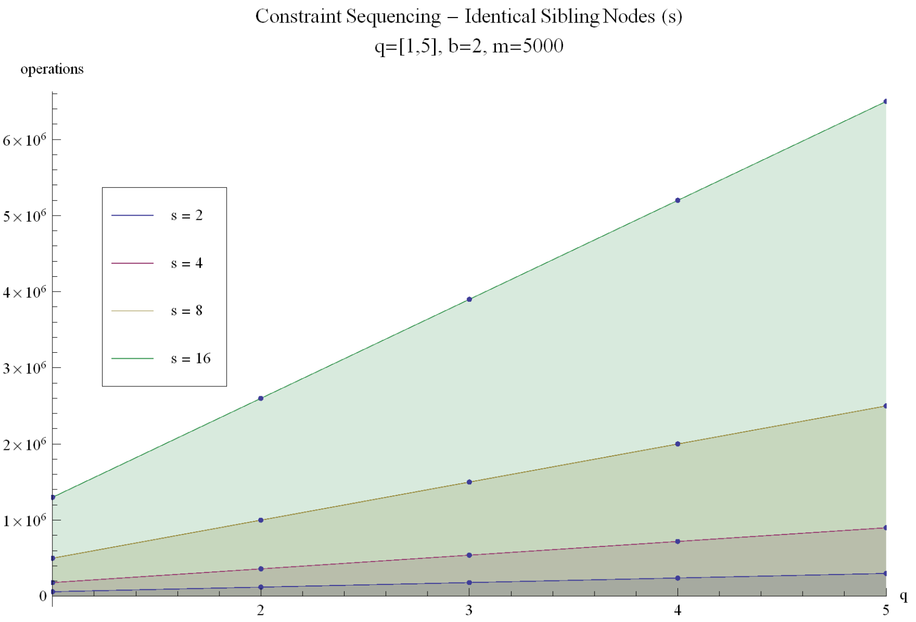


Figure B.3: Constraint Sequencing, identical sibling nodes increasing (smaller branching factor)

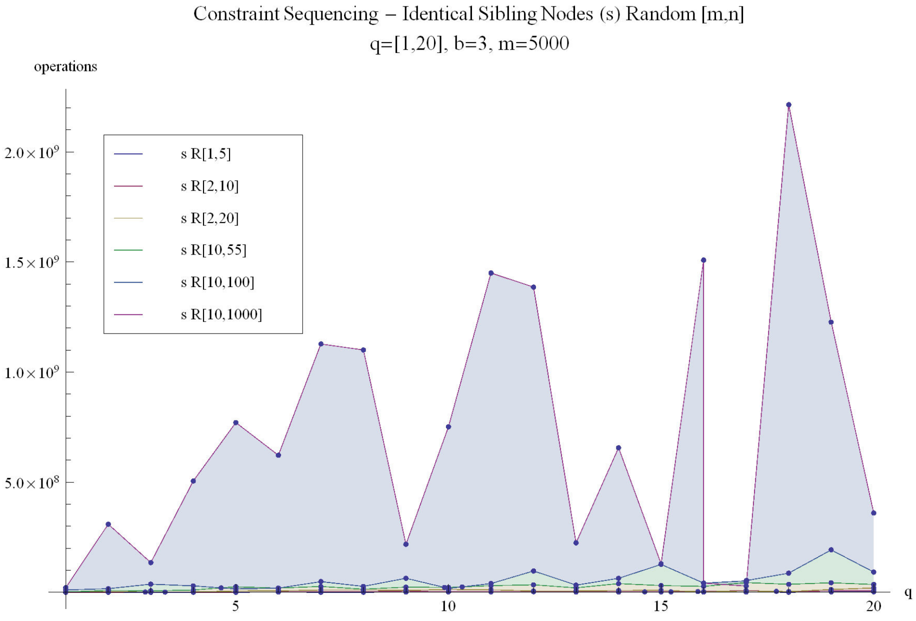


Figure B.4: Constraint Sequencing, identical sibling nodes increasing (1000 max) - single run

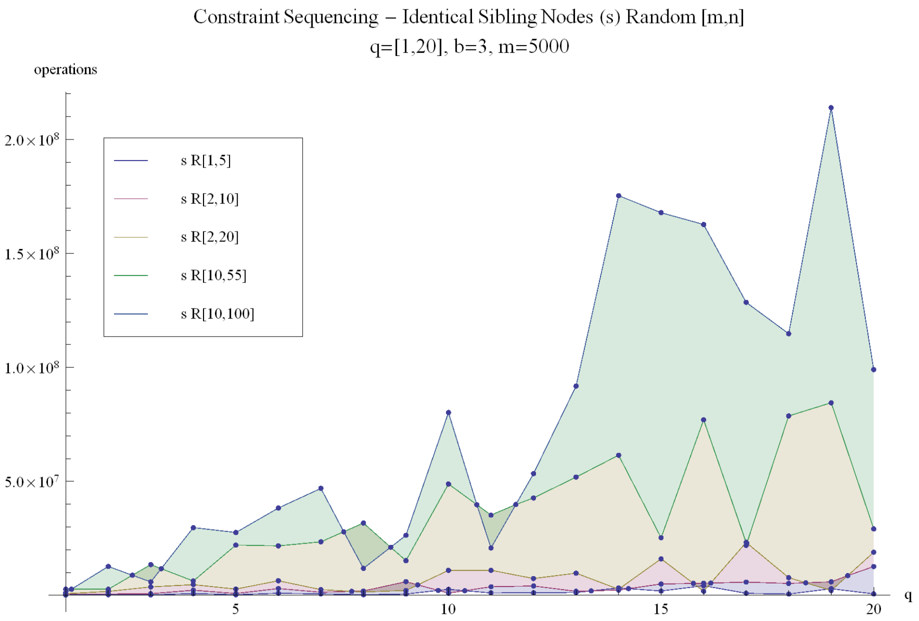


Figure B.5: Constraint Sequencing, identical sibling nodes increasing (100 max) - single run

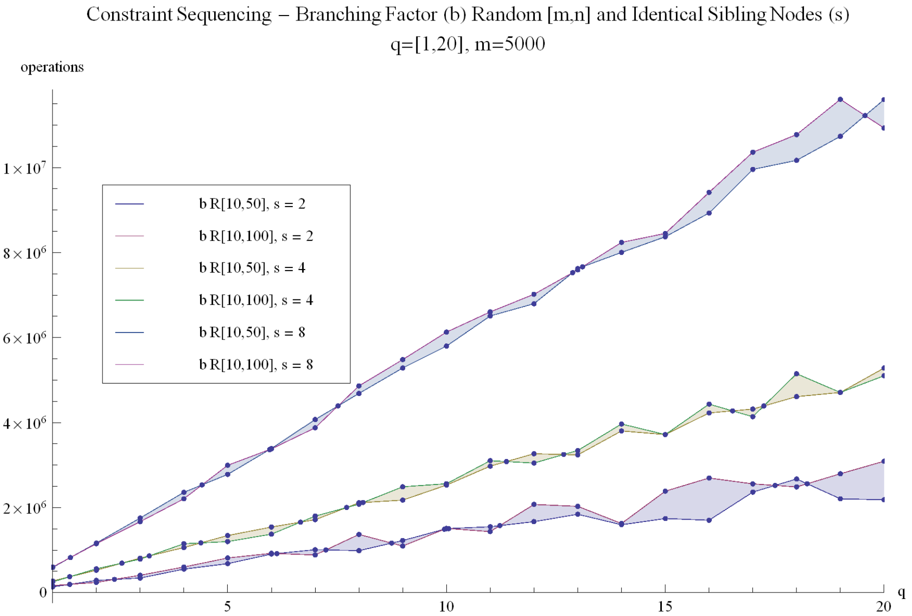


Figure B.6: Constraint Sequencing, random branching factor and constant identical sibling nodes - single run

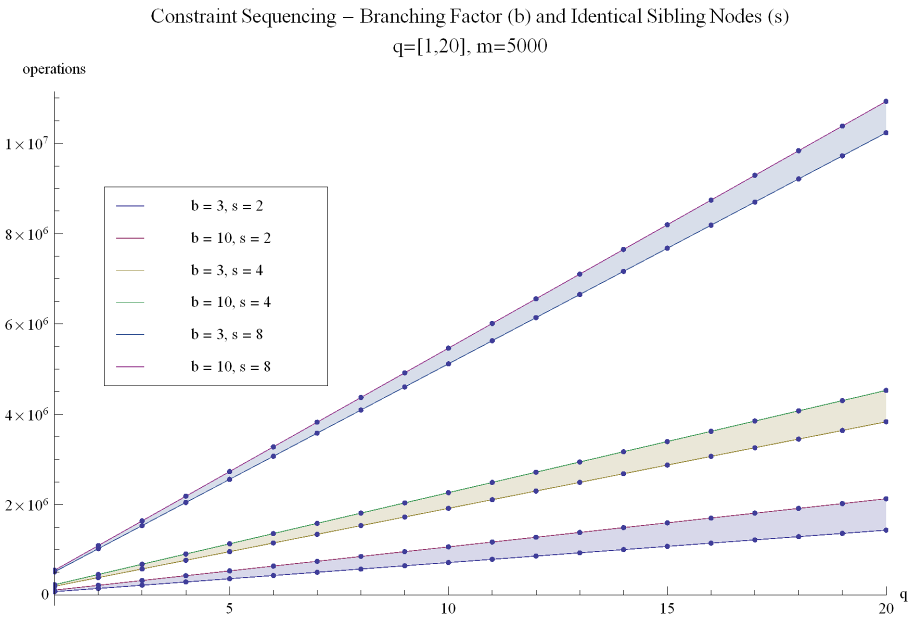


Figure B.7: Constraint Sequencing, various branching factors and identical sibling nodes

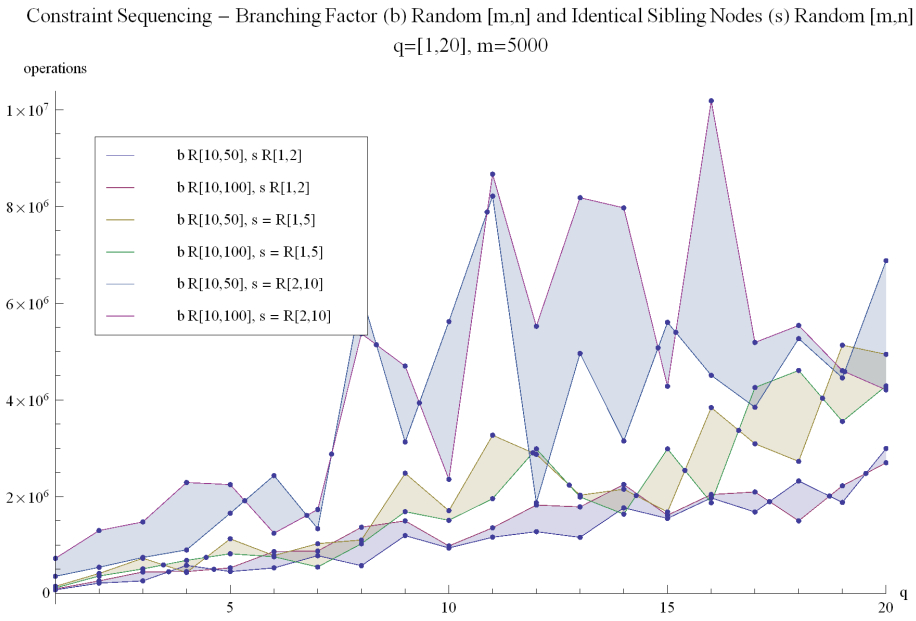


Figure B.8: Constraint Sequencing, random branching factor and identical sibling nodes (larger range) - single run

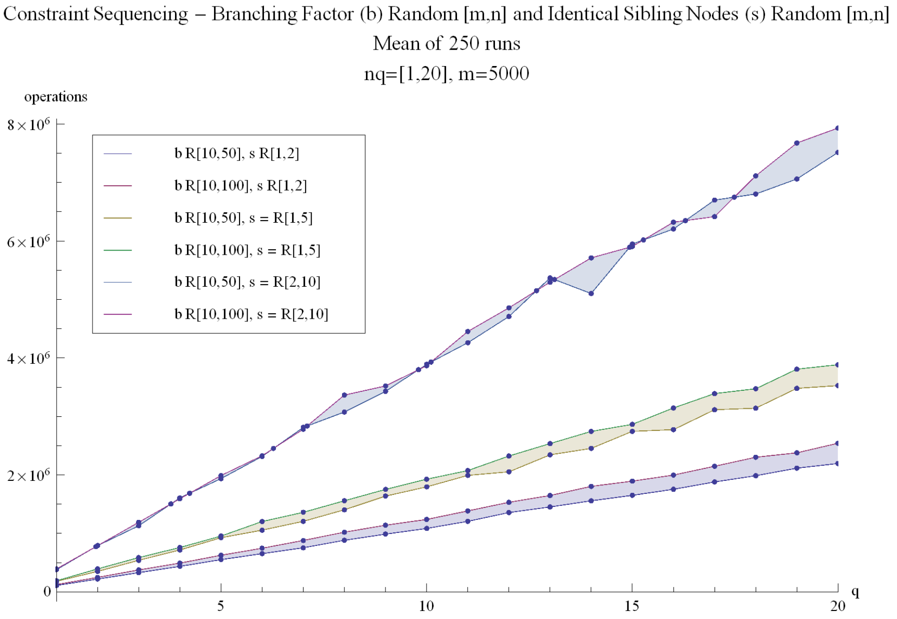


Figure B.9: Constraint Sequencing, random branching factor and identical sibling nodes (larger range) - 250 runs

Appendix C

SS-Join Graphs

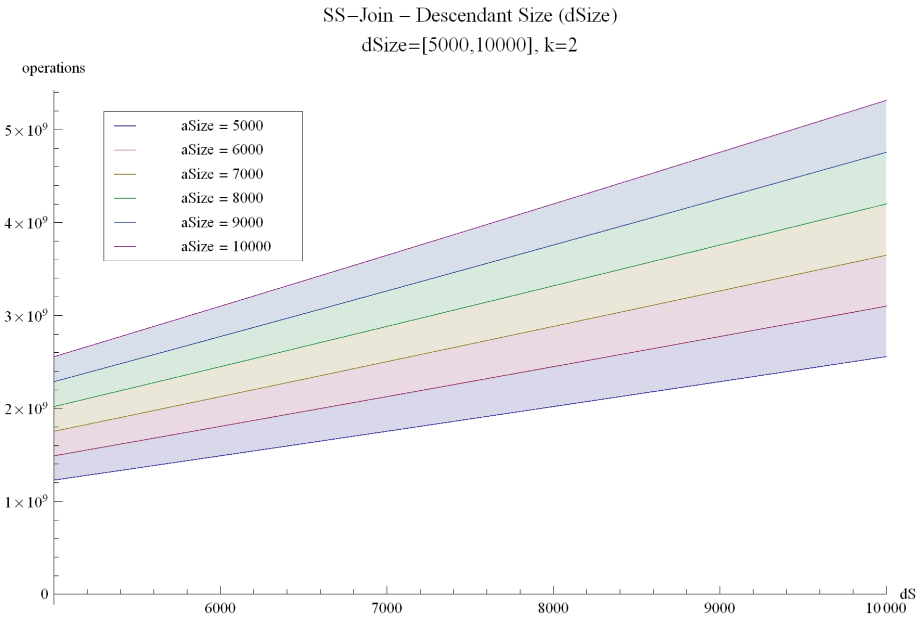


Figure C.1: SS-Join, various descendant list sizes (larger range)

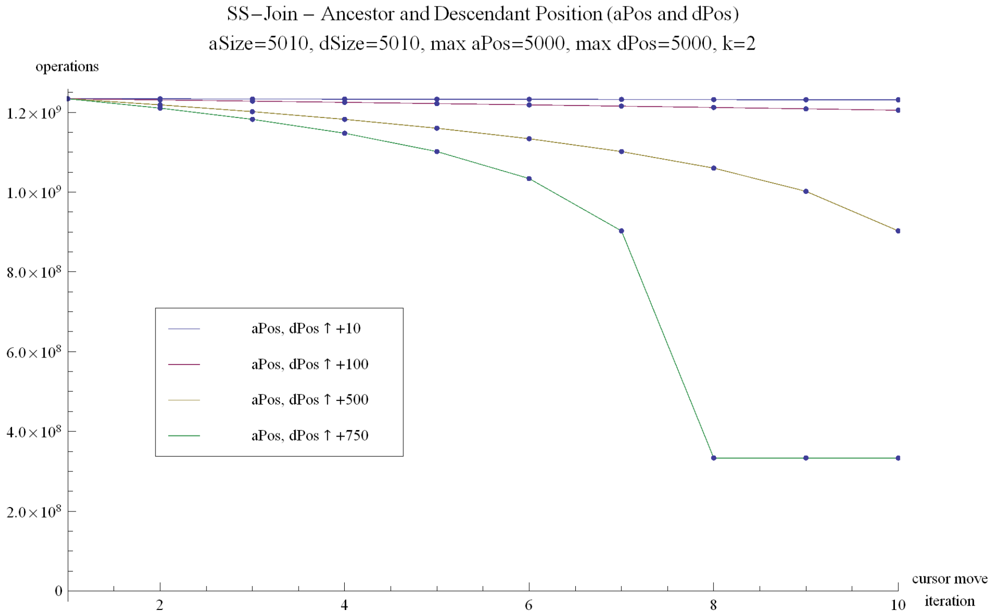


Figure C.2: SS-Join, $aPos/dPos$ increasing (larger lists)

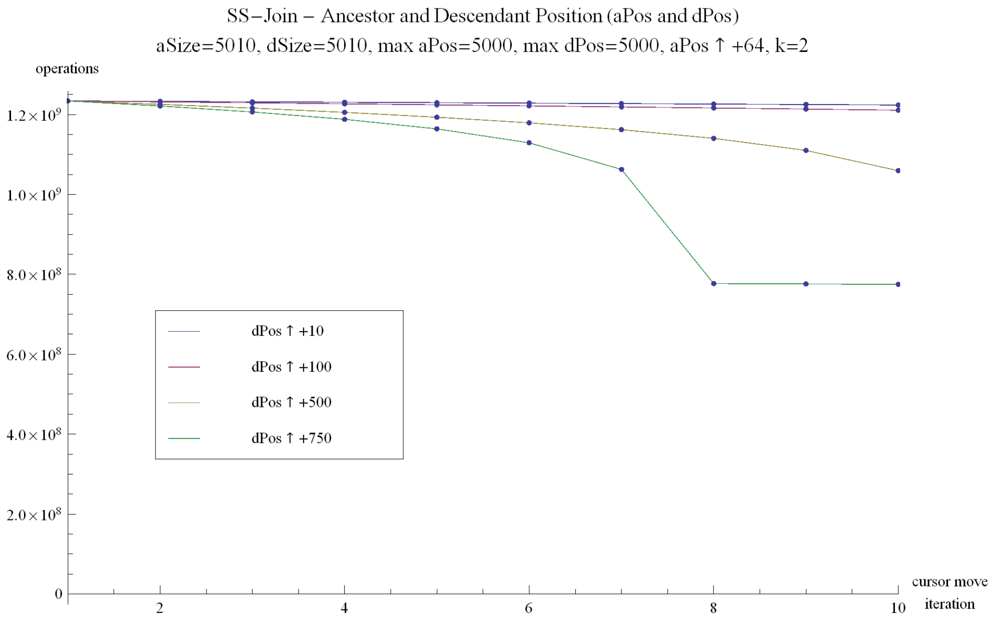


Figure C.3: SS-Join, *dPos* increasing by various amounts, *aPos* increasing by fixed amount

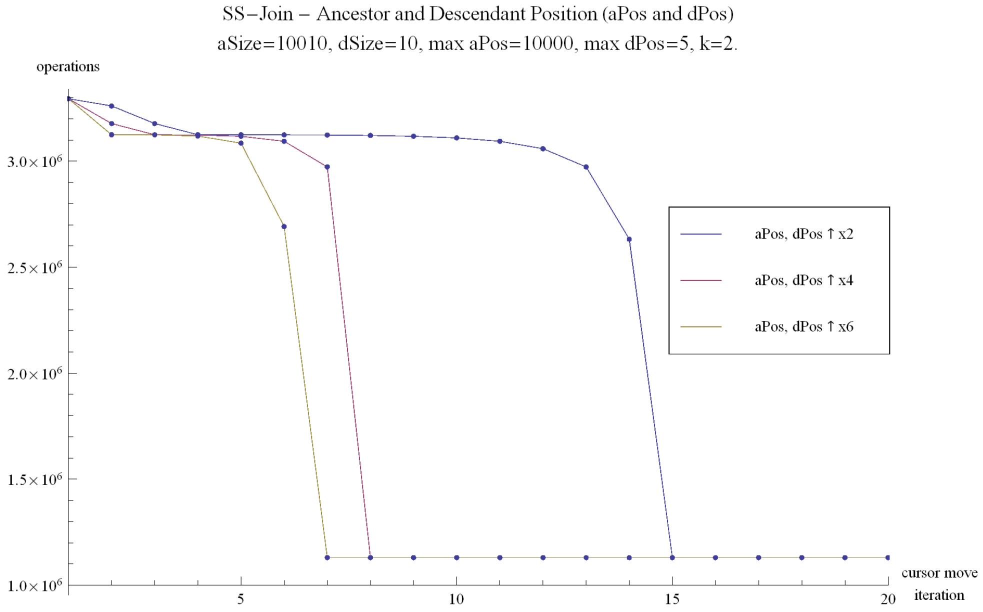


Figure C.4: SS-Join, *aPos*/*dPos* increasing (different size lists)

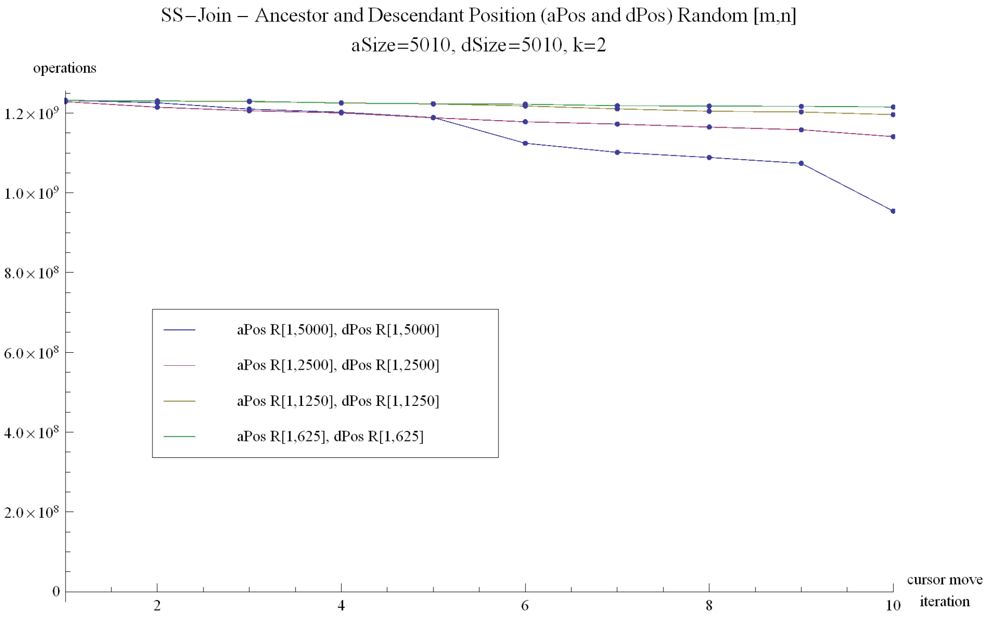


Figure C.5: SS-Join, random increases to $aPos/dPos$ (large, identical ranges) - single run

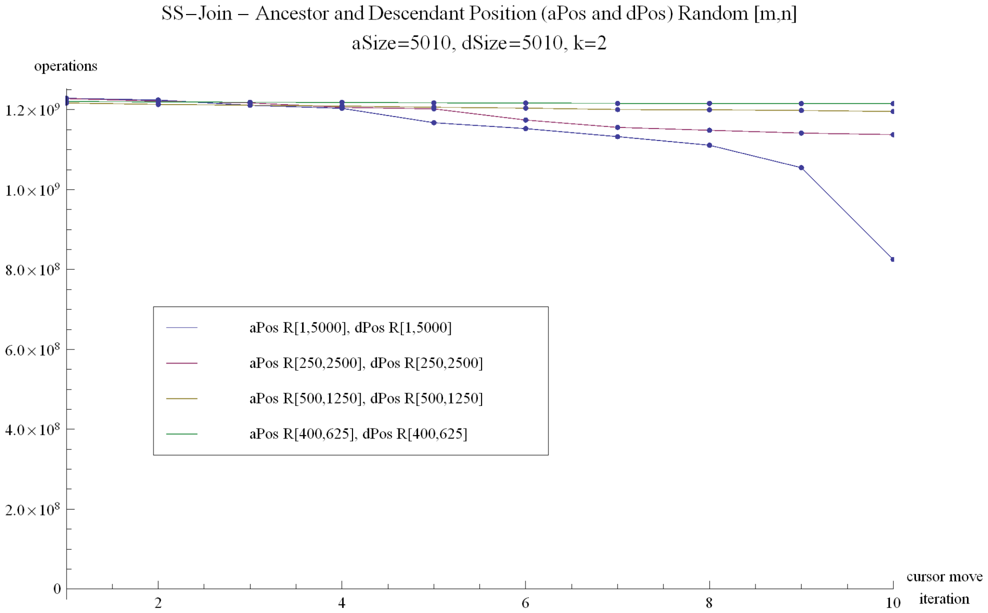


Figure C.6: SS-Join, random increases to $aPos/dPos$ (narrowing, identical ranges) - single run

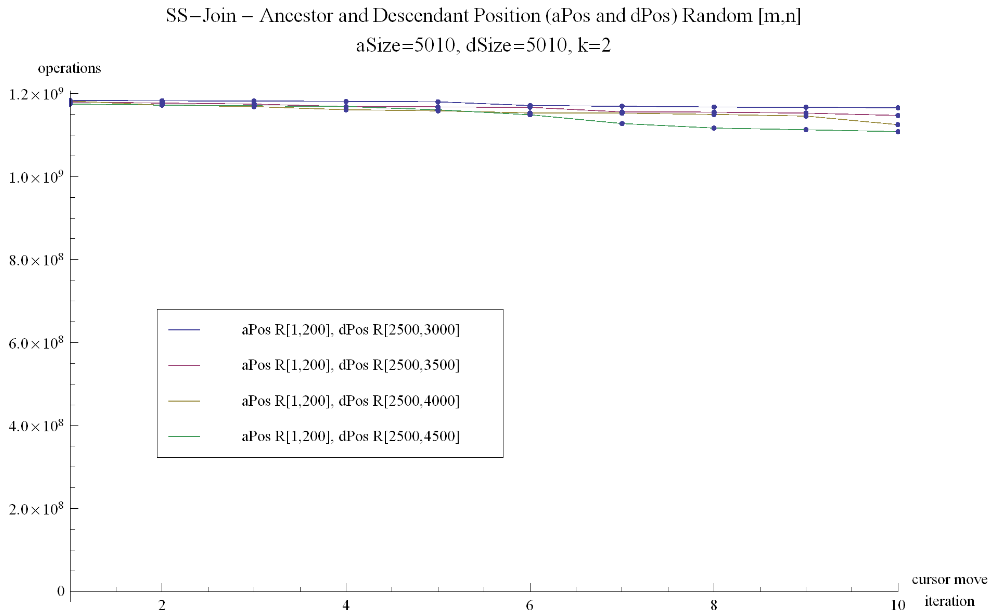


Figure C.7: SS-Join, random increases to *aPos/dPos* (one range fixed) - single run

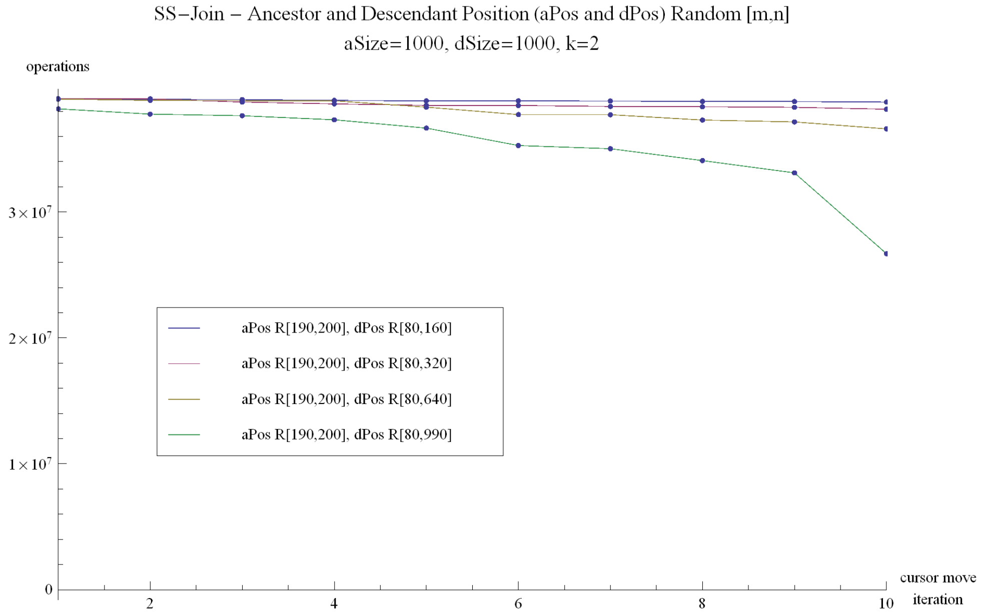


Figure C.8: SS-Join, random increases to *aPos/dPos* (one range fixed small) - single run

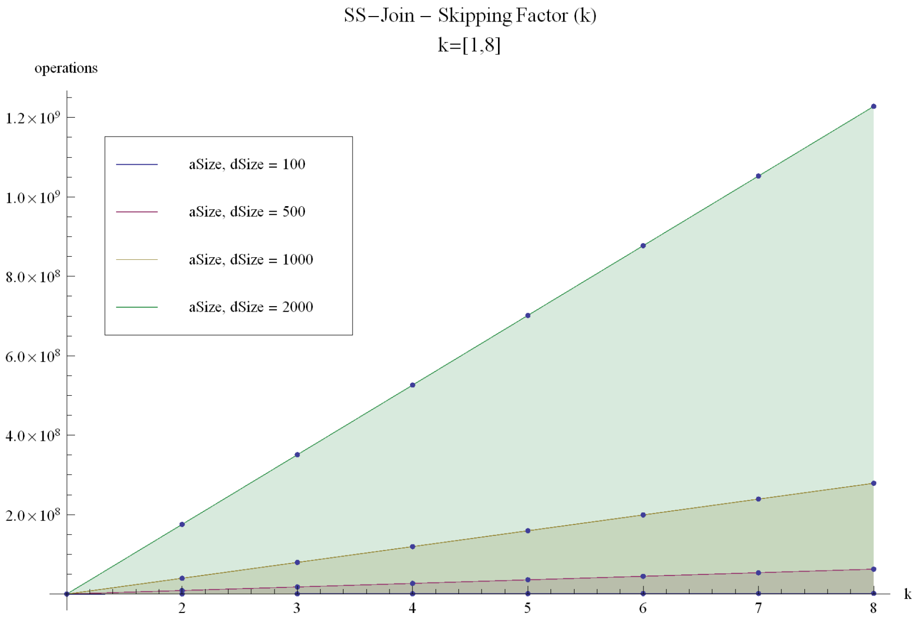


Figure C.9: SS-Join, skipping factor increasing (large lists)

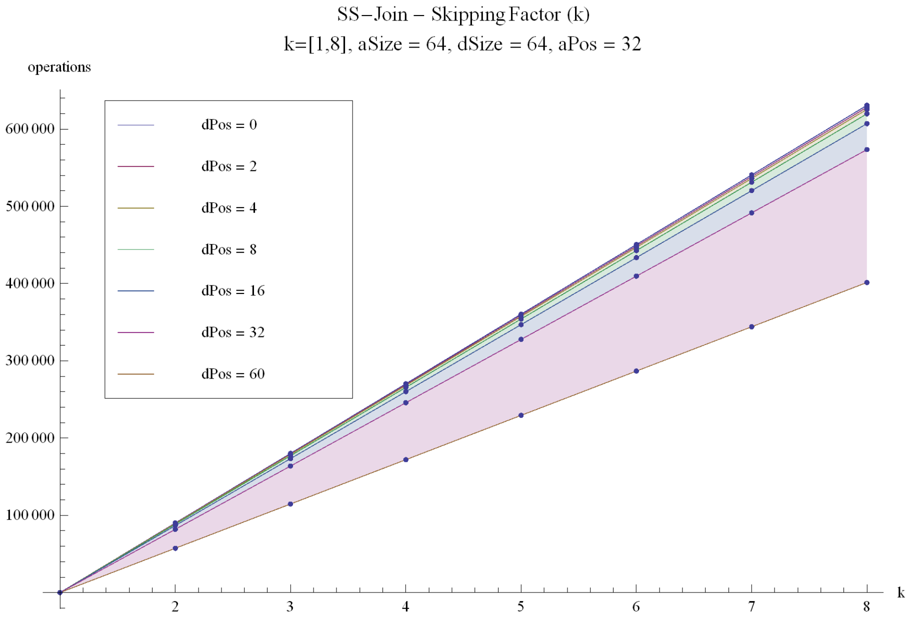


Figure C.10: SS-Join, skipping factor increasing (*aPos* fixed at 32, small lists)

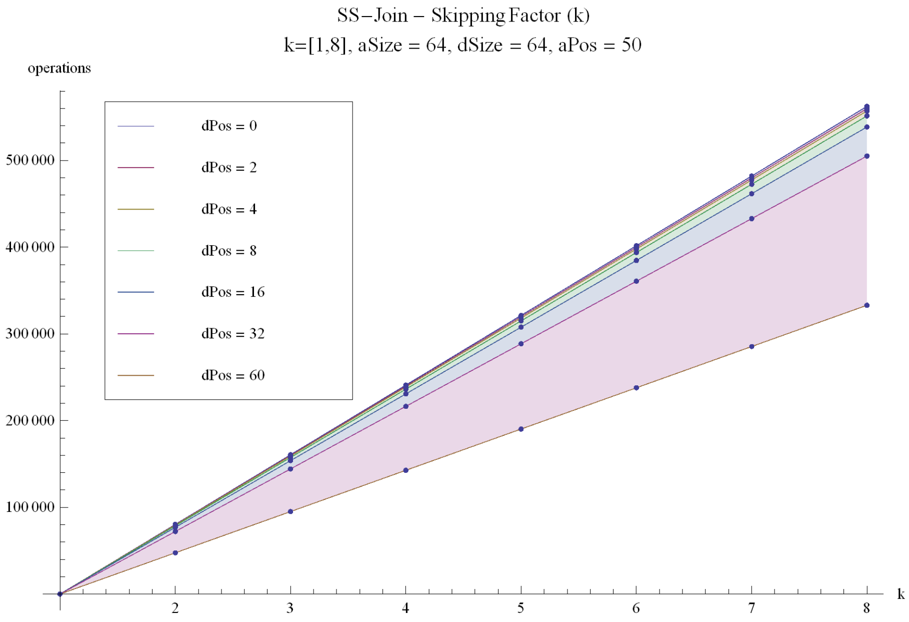


Figure C.11: SS-Join, skipping factor increasing (*aPos* fixed at 50, small lists)

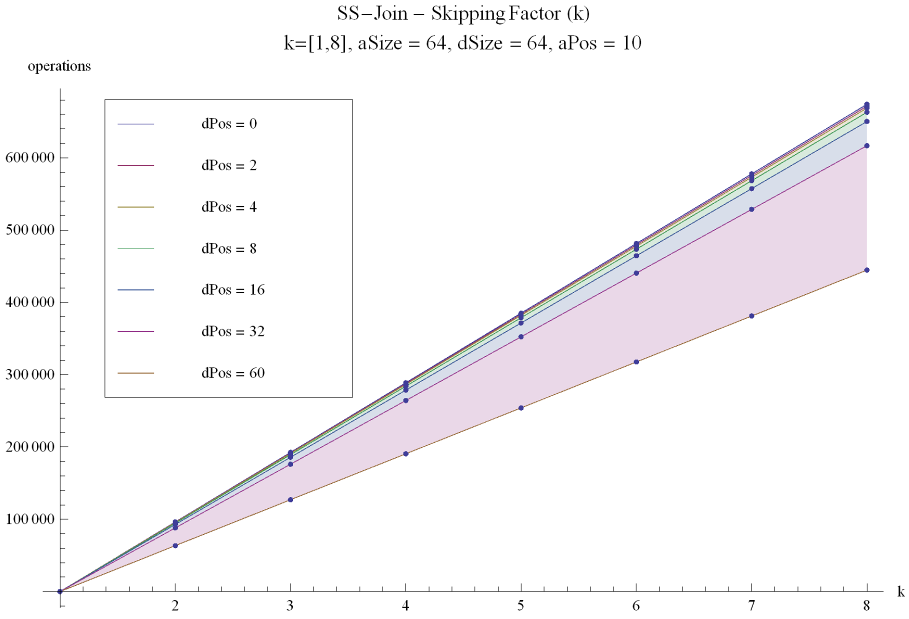


Figure C.12: SS-Join, skipping factor increasing (*aPos* fixed at 10, small lists)

Appendix D

RDBQuery Graphs

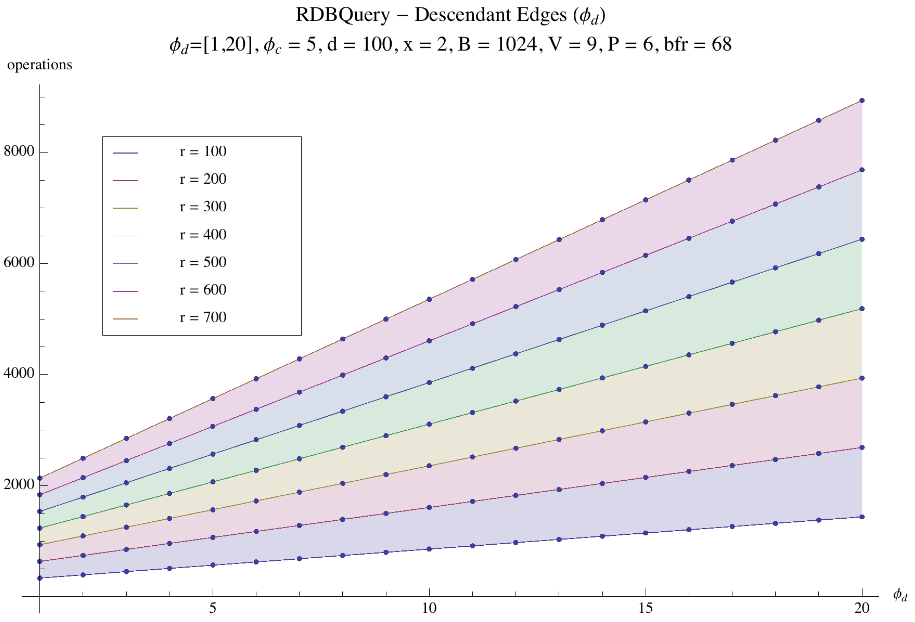


Figure D.1: RDBQuery, record size increasing (small record range)

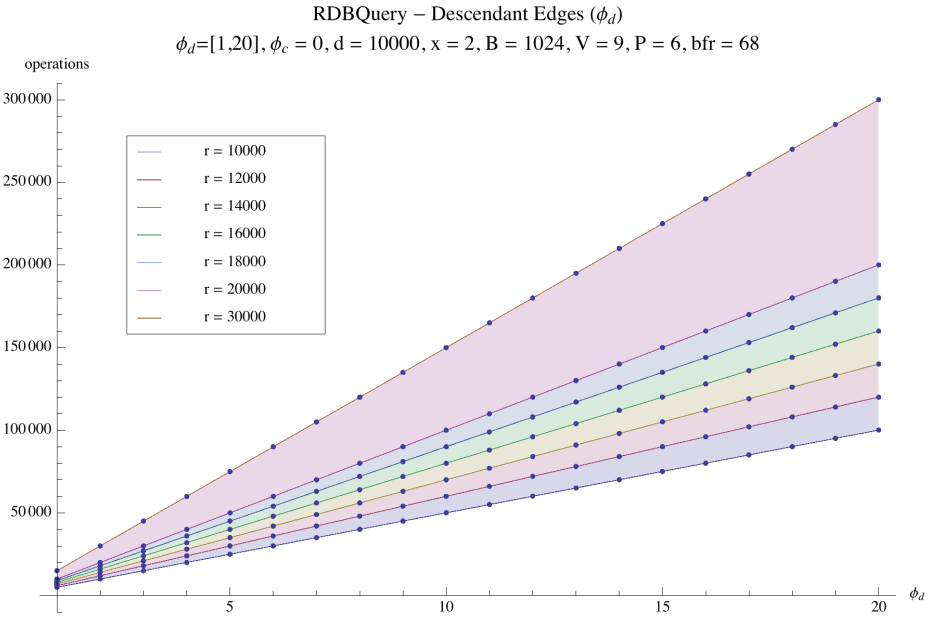


Figure D.2: RDBQuery, record size increasing(no child edges)

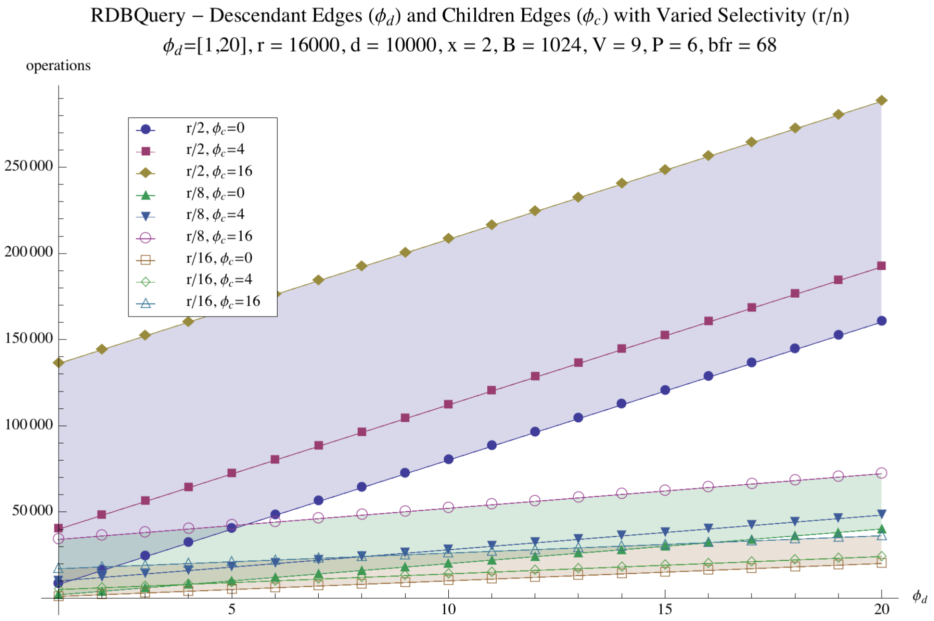


Figure D.3: RDBQuery, selectivity and descendant/child edges increasing (all values shown)

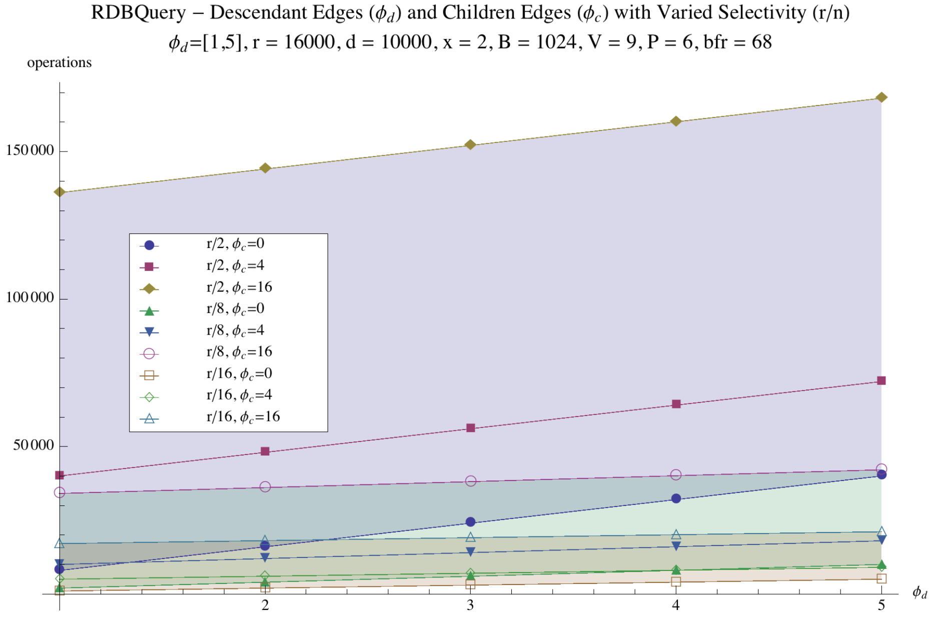


Figure D.4: RDBQuery, selectivity and descendant/child edges increasing(smaller query, all values shown)

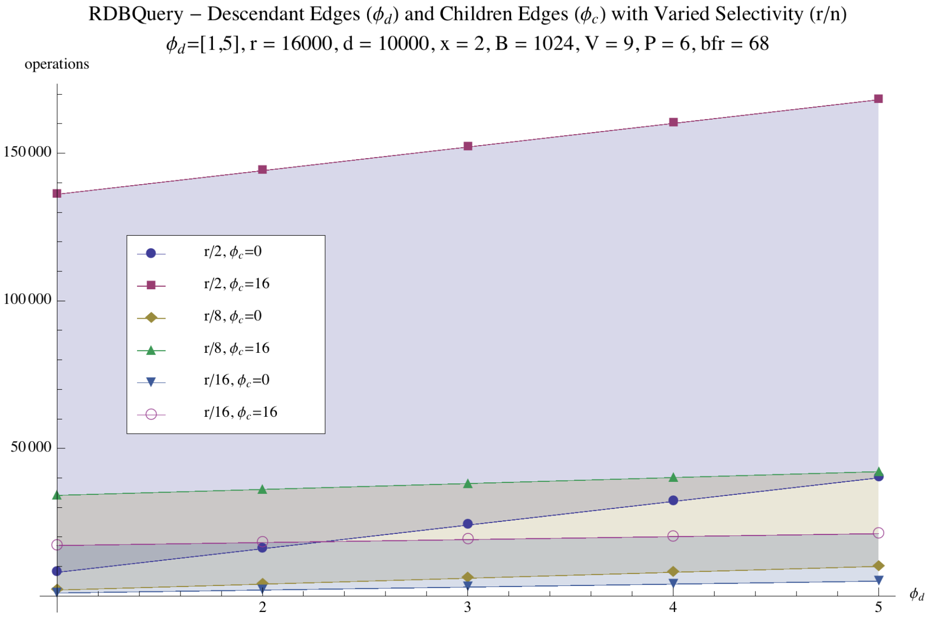


Figure D.5: RDBQuery, selectivity and descendant/child edges increasing(smaller query, partial values shown)

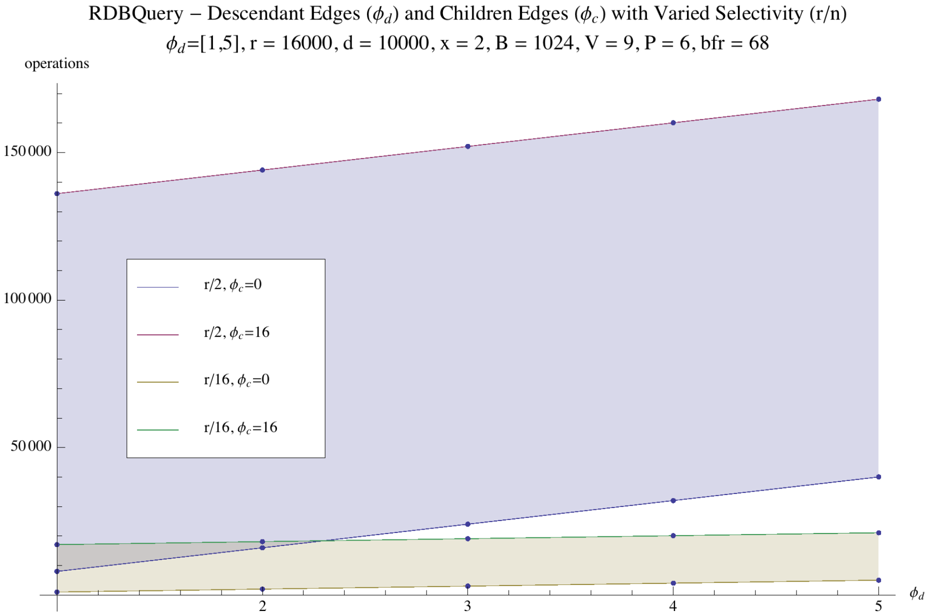


Figure D.6: RDBQuery, selectivity and descendant/child edges increasing (smaller query, min/max values shown)

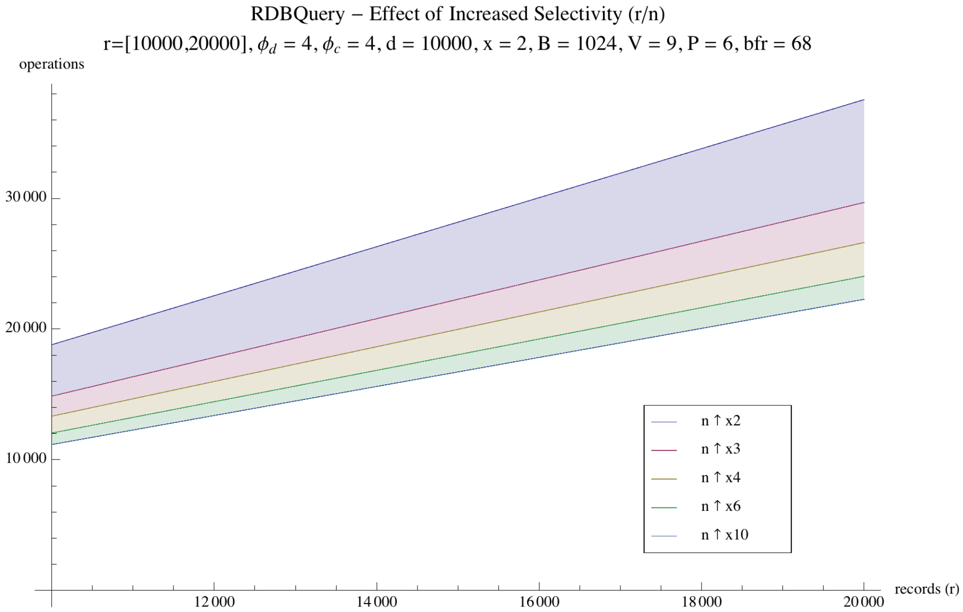


Figure D.7: RDBQuery, selectivity increasing by constant factors (smaller query)

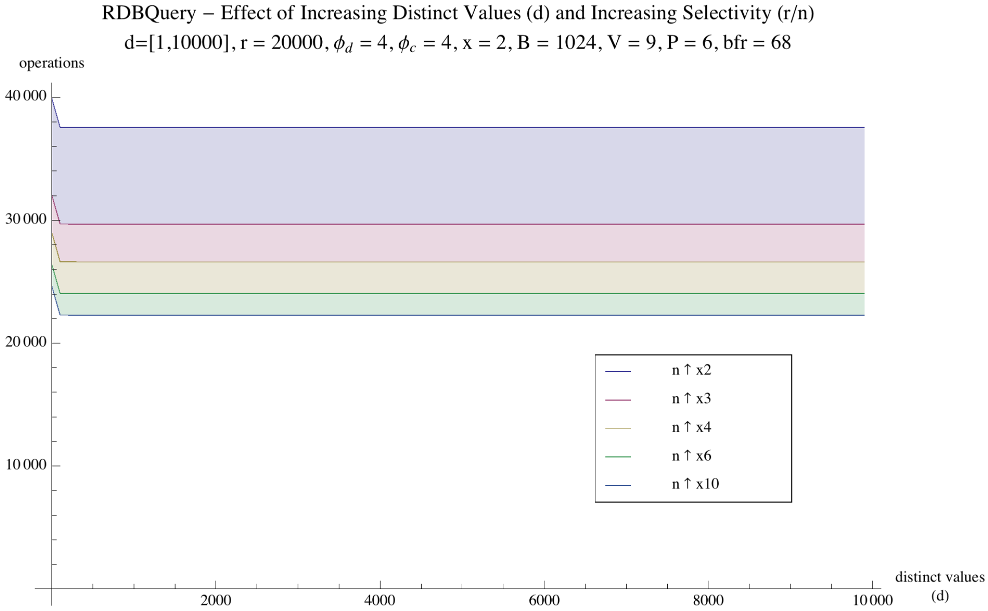


Figure D.8: RDBQuery, distinct values and selectivity increasing

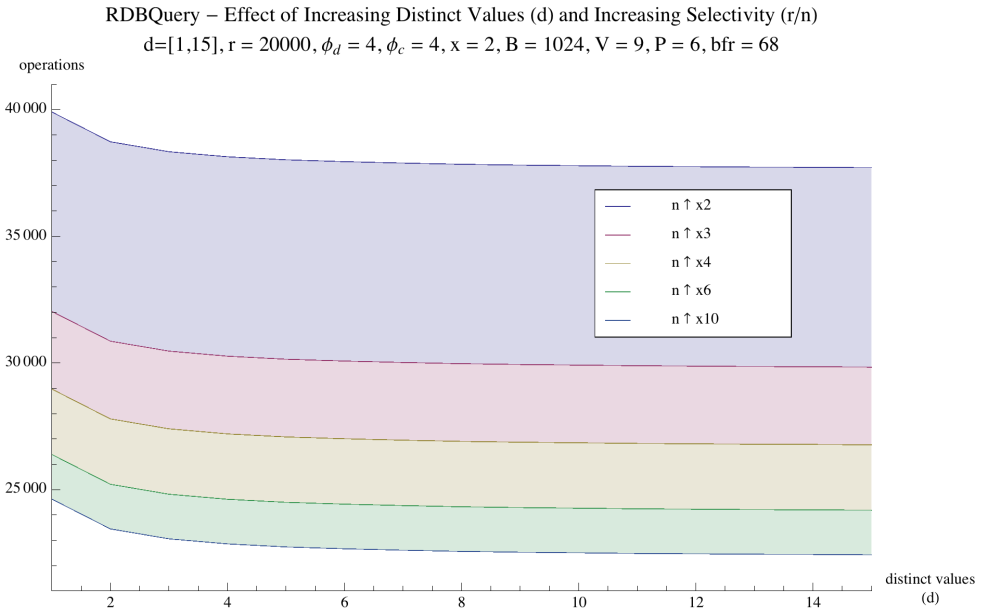


Figure D.9: RDBQuery, small range of distinct values and selectivity increasing

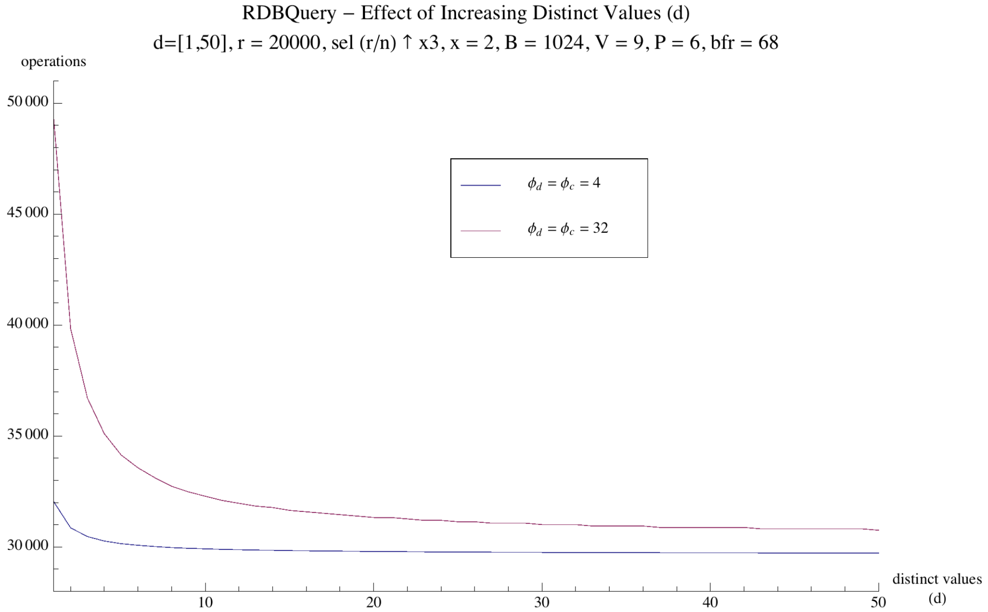


Figure D.10: RDBQuery, medium range of distinct values and selectivity increasing

Appendix E

Native Comparison Graphs

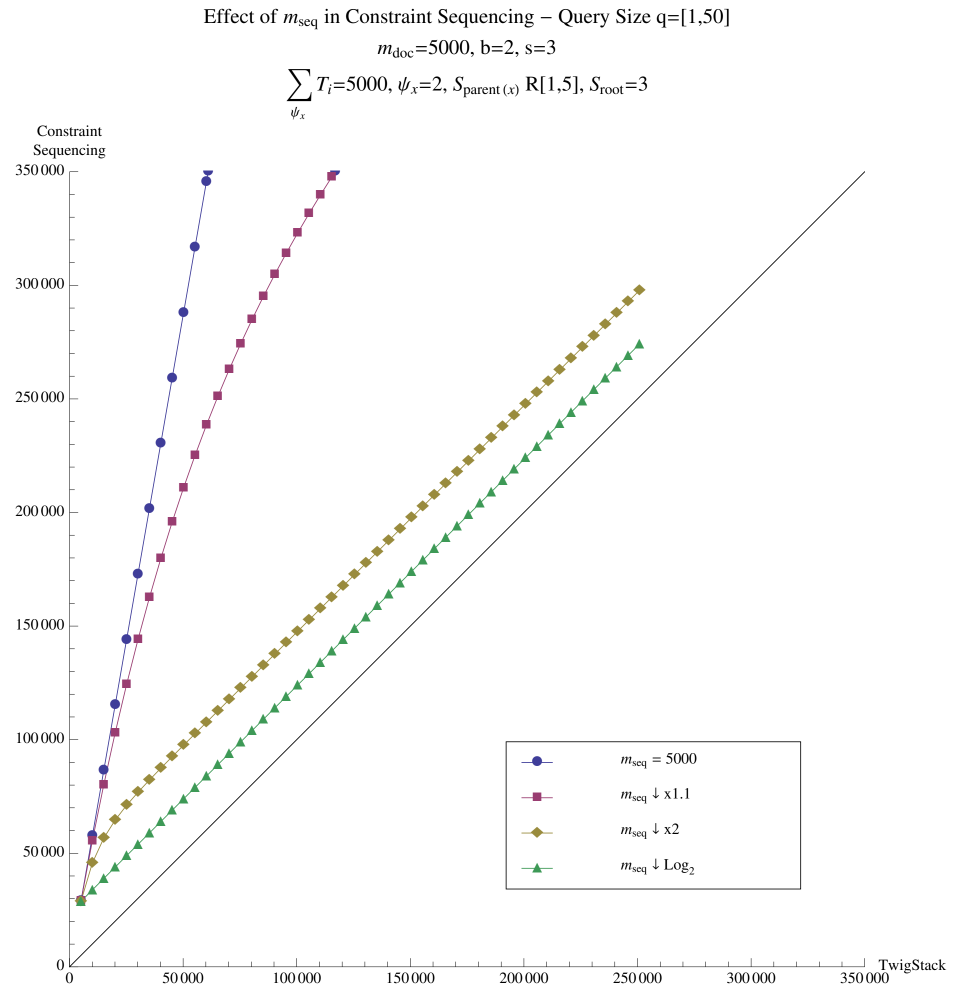


Figure E.1: CS, vary sequence size (low random $S_{parent(x)}$, increased s) - Deep

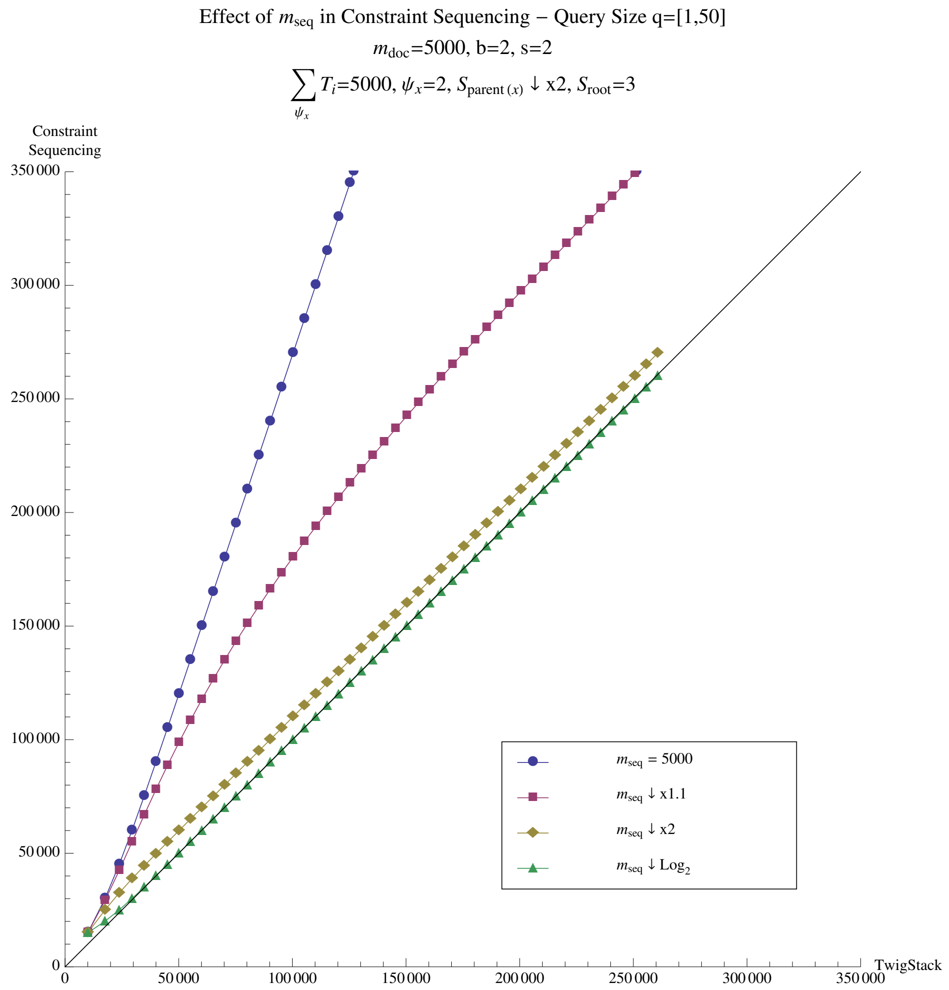


Figure E.2: CS, vary sequence size (decreasing $S_{parent(x)}$) - Deep

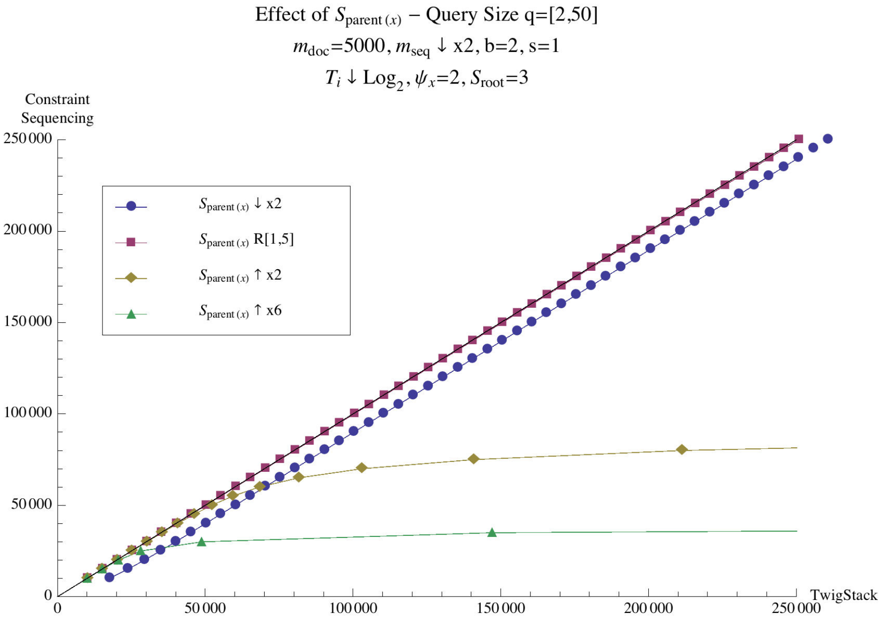


Figure E.3: TS, vary stack size in small random range (low s) - Deep

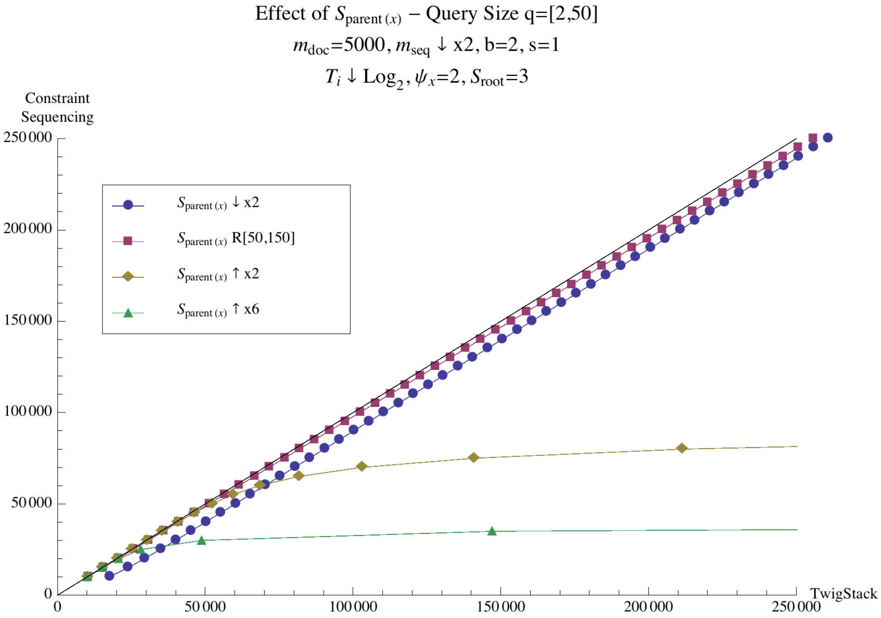


Figure E.4: TS, vary stack size in medium random range (low s) - Deep

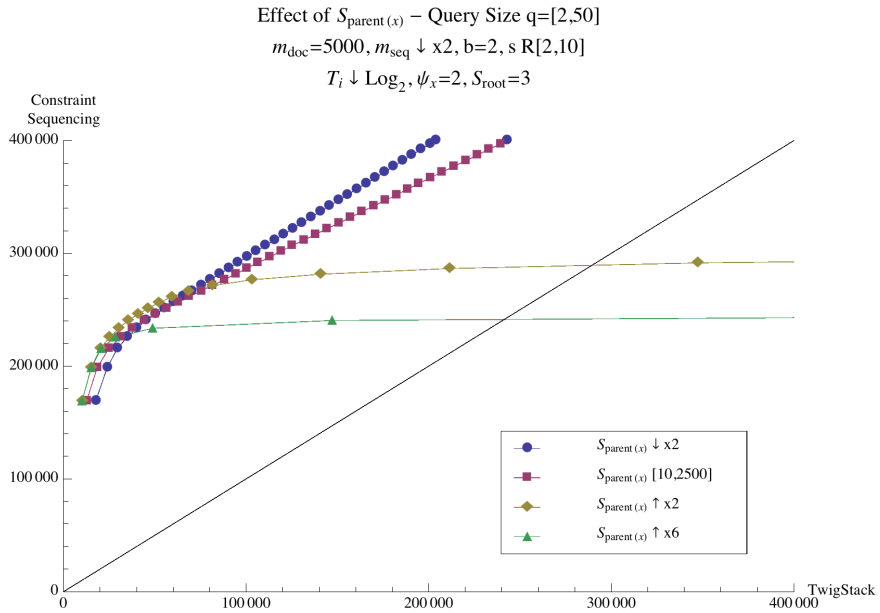


Figure E.5: TS, vary stack size in medium random range (random s) - Deep

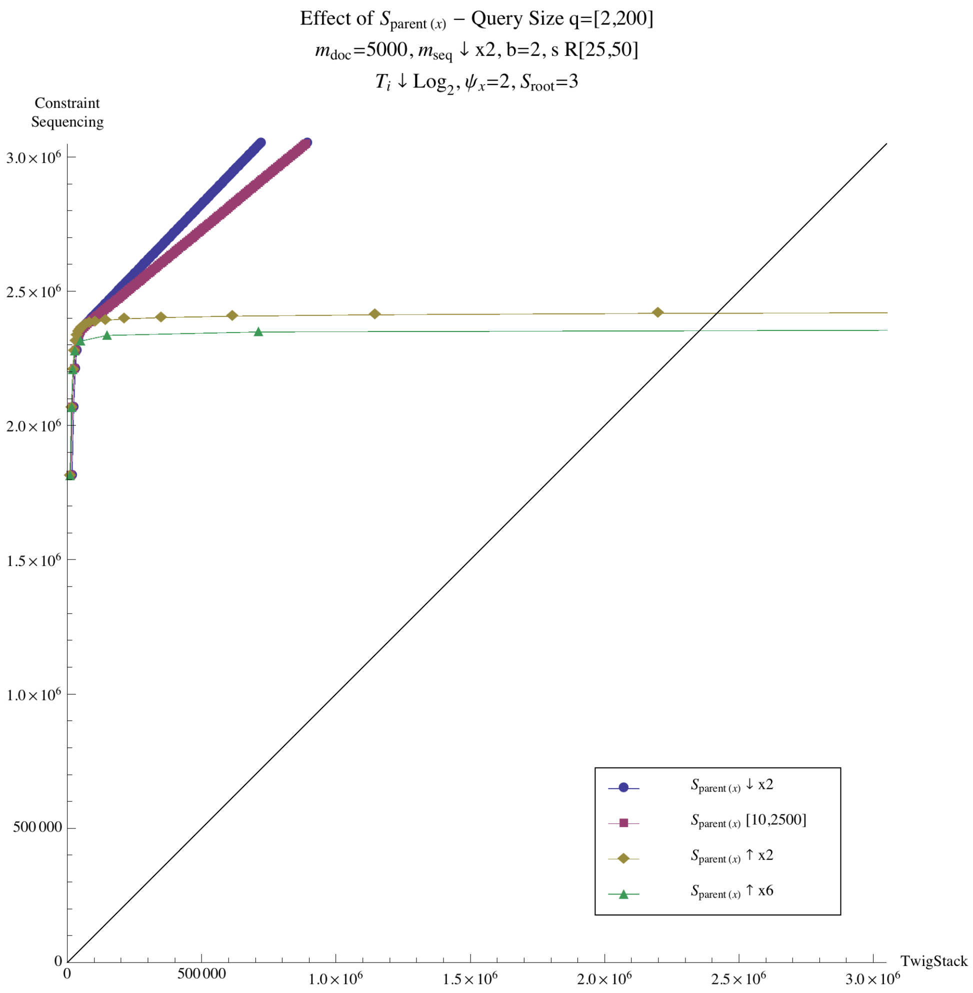


Figure E.6: TS, vary stack size in medium random range (large random s) - Deep

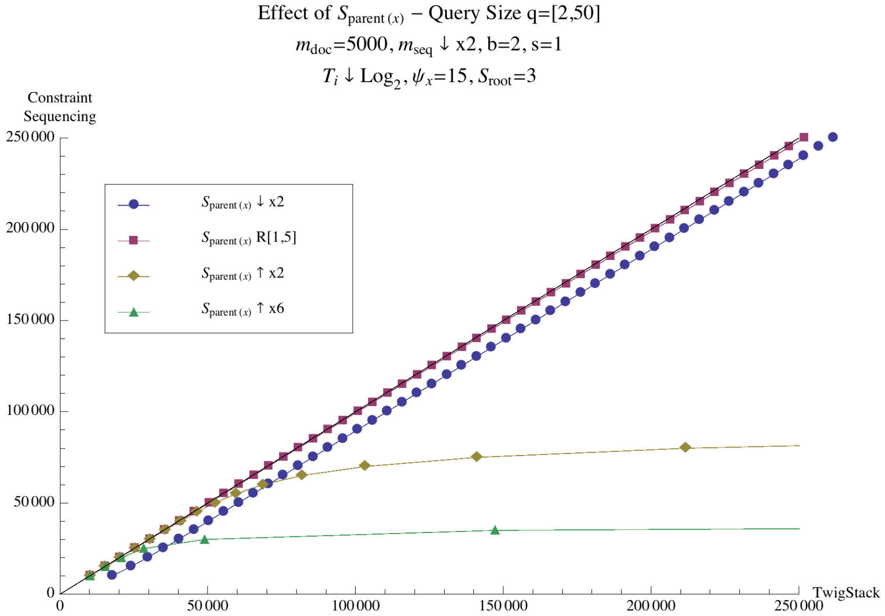


Figure E.7: TS, vary stack size in small random range (low s, high ψ_x) - Deep

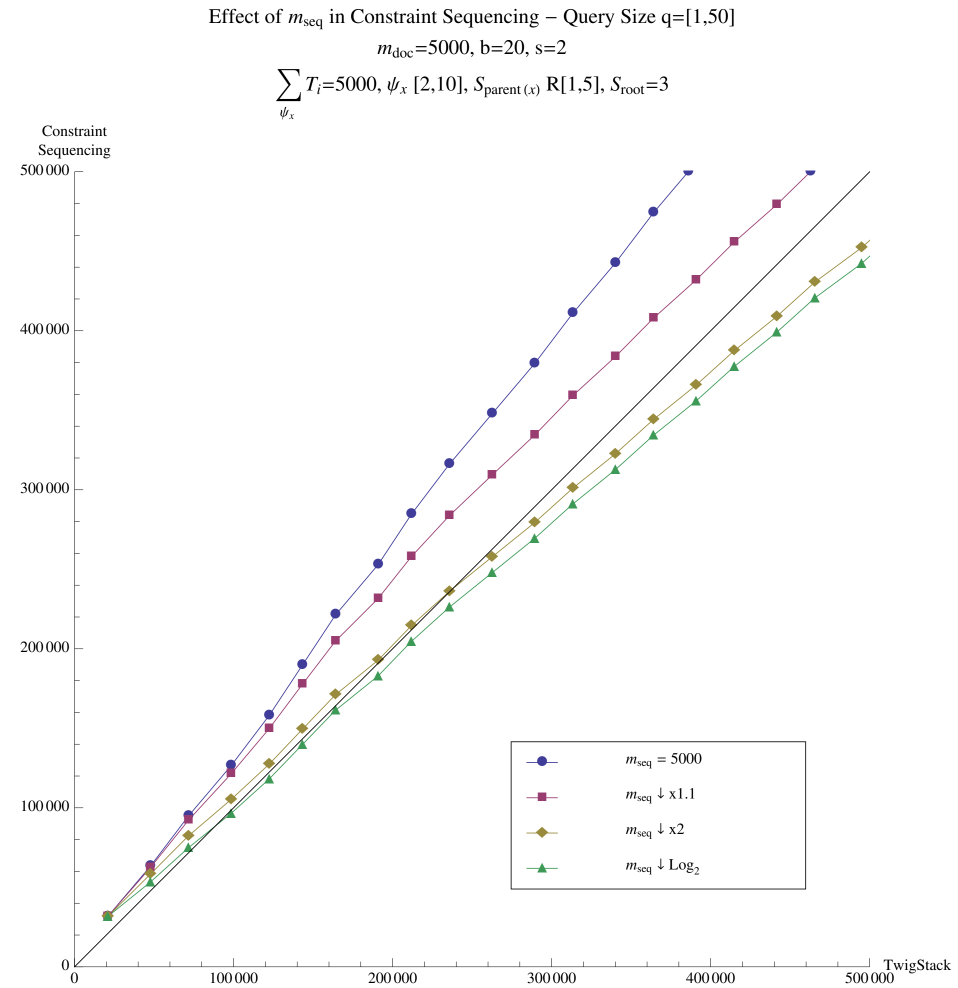


Figure E.8: CS, vary sequence size (low random $S_{parent(x)}$, random ψ_x) - Wide

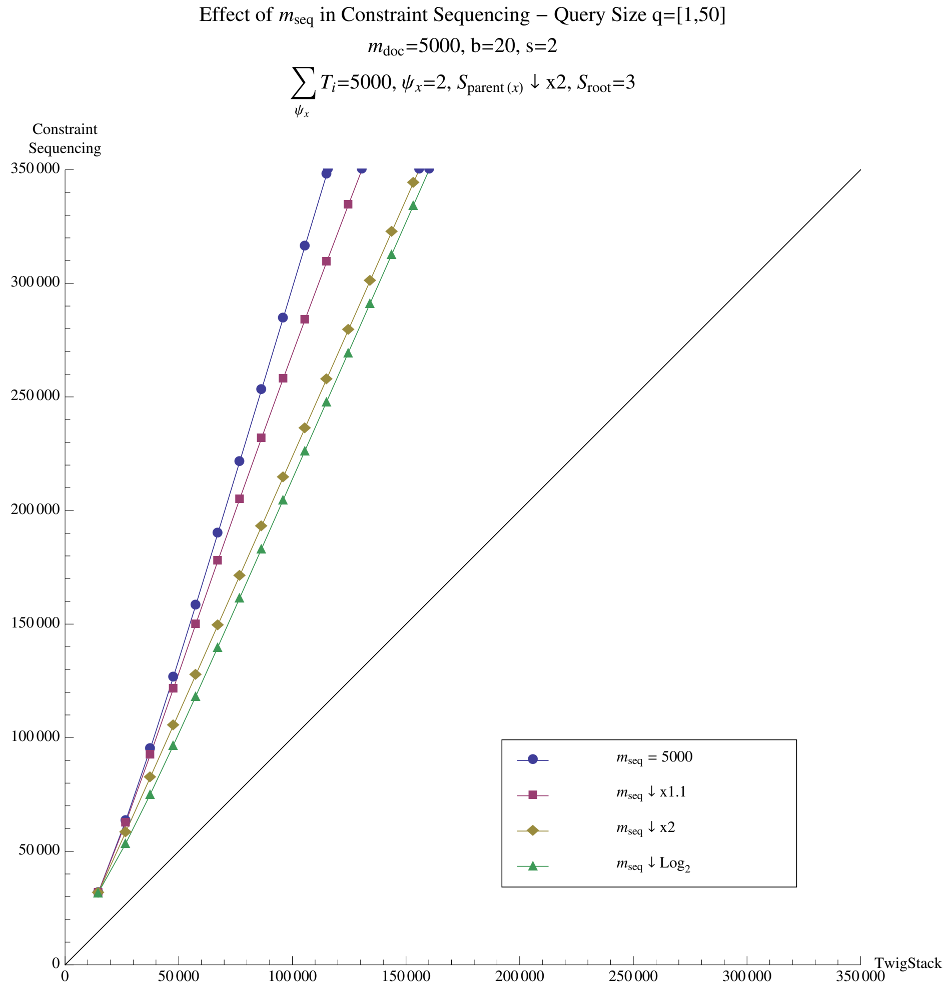


Figure E.9: CS, vary sequence size (decreasing $S_{parent(x)}$) - Wide

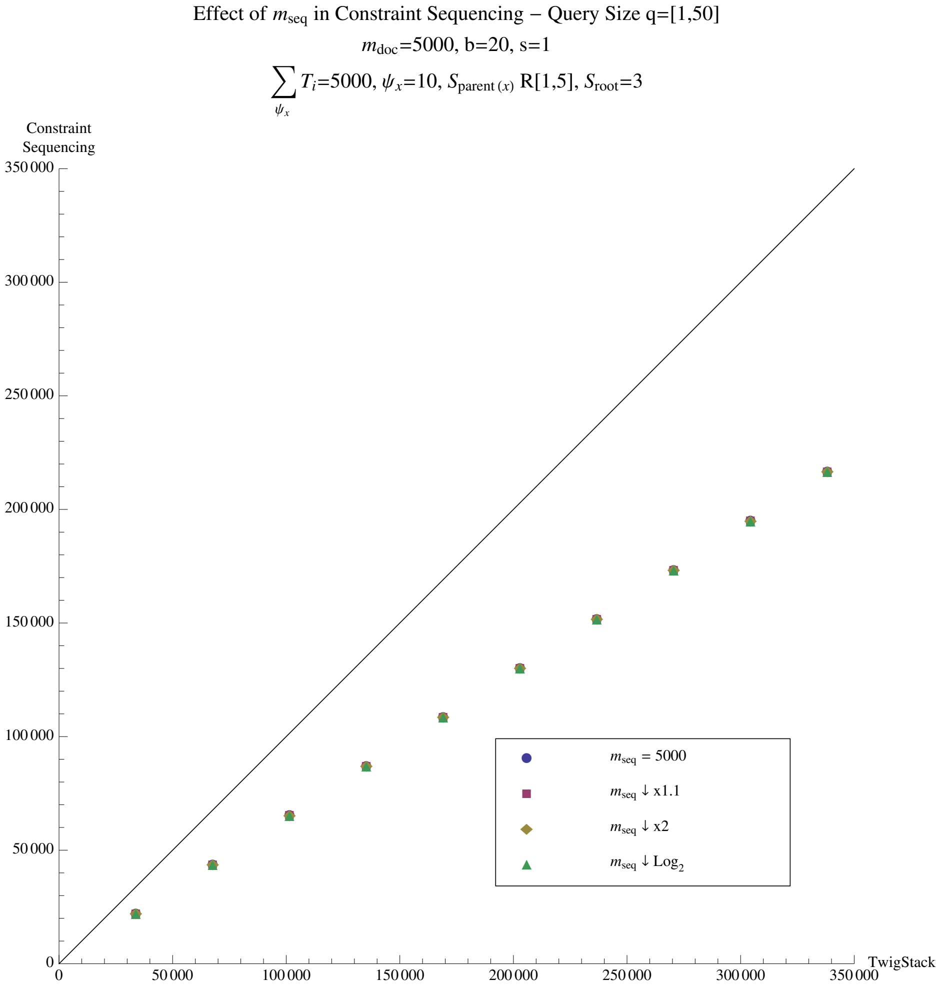


Figure E.10: CS, vary sequence size (low s, high ψ_x) - Wide

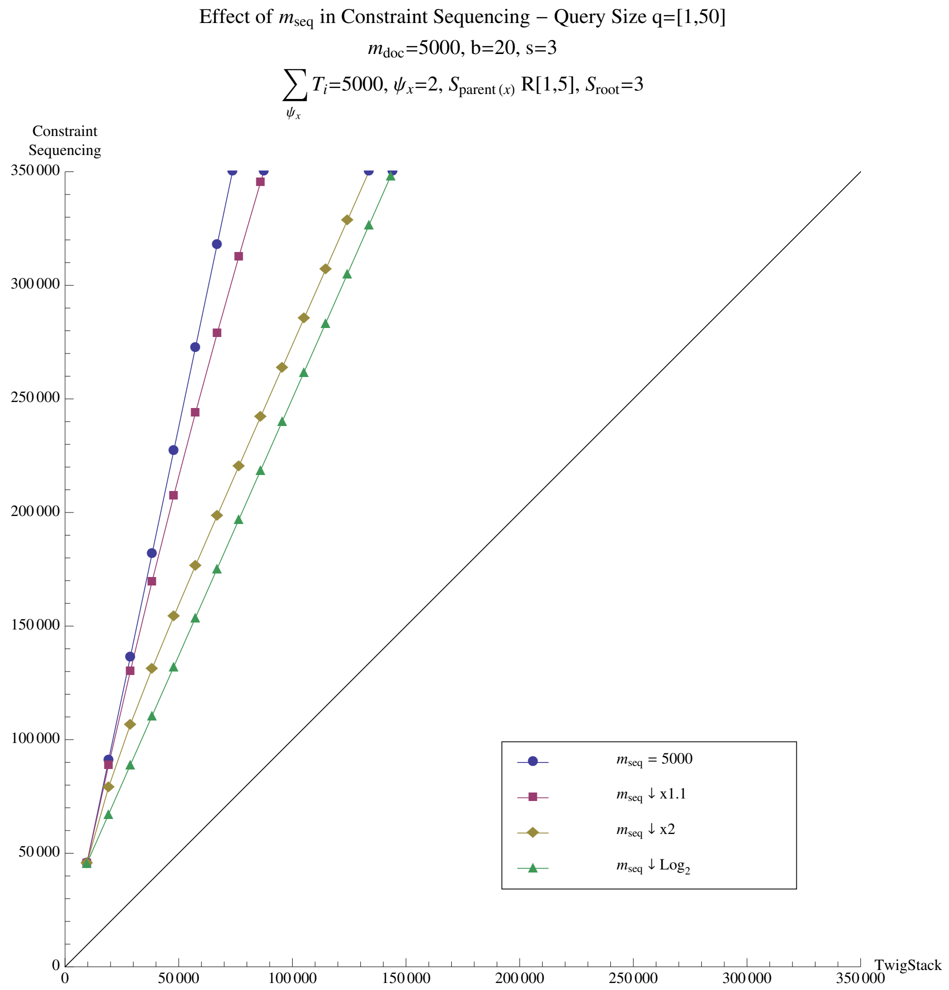


Figure E.11: CS, vary sequence size (low random $S_{parent(x)}$, increased s) - Wide

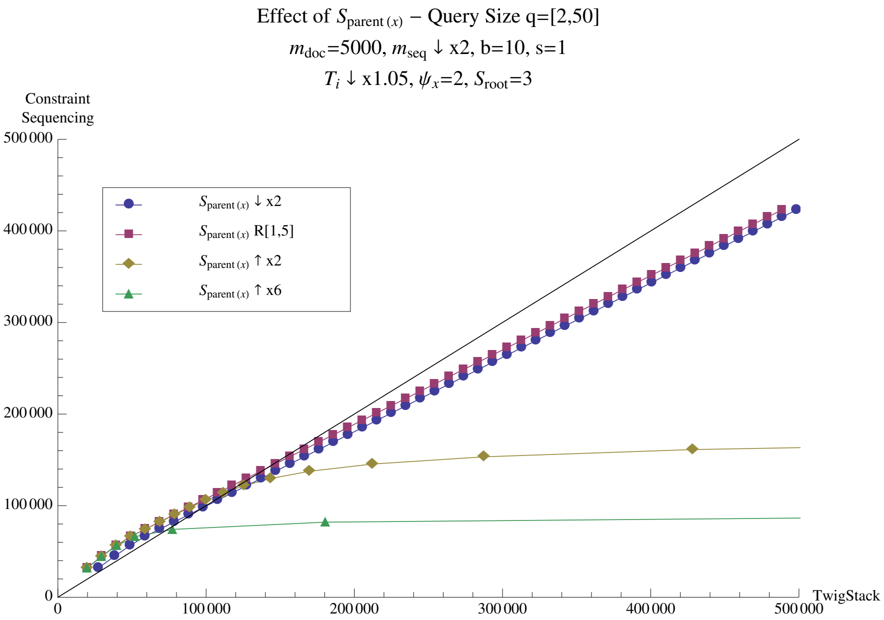


Figure E.12: TS, vary stack size in small random range (low s) - Wide

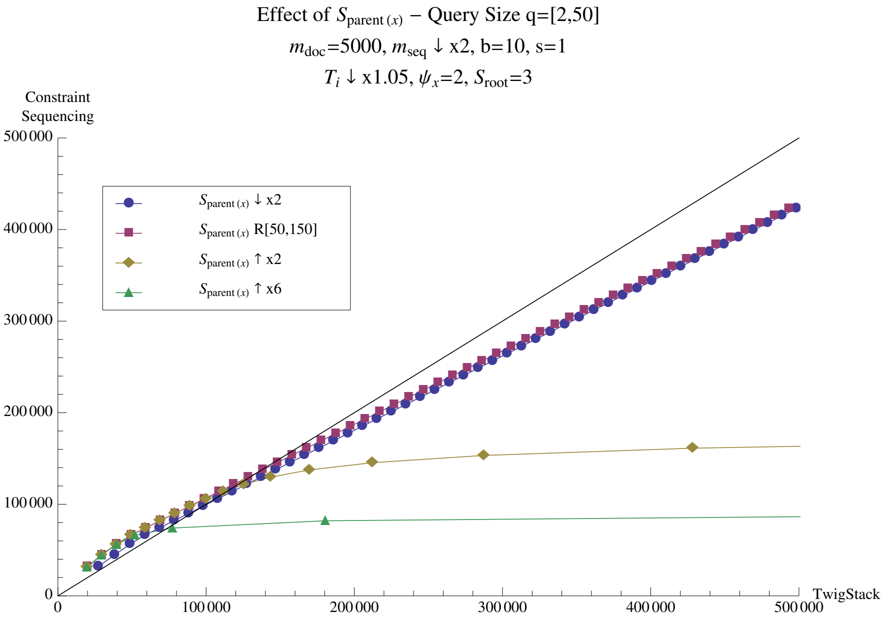


Figure E.13: TS, vary stack size in medium random range (low s) - Wide

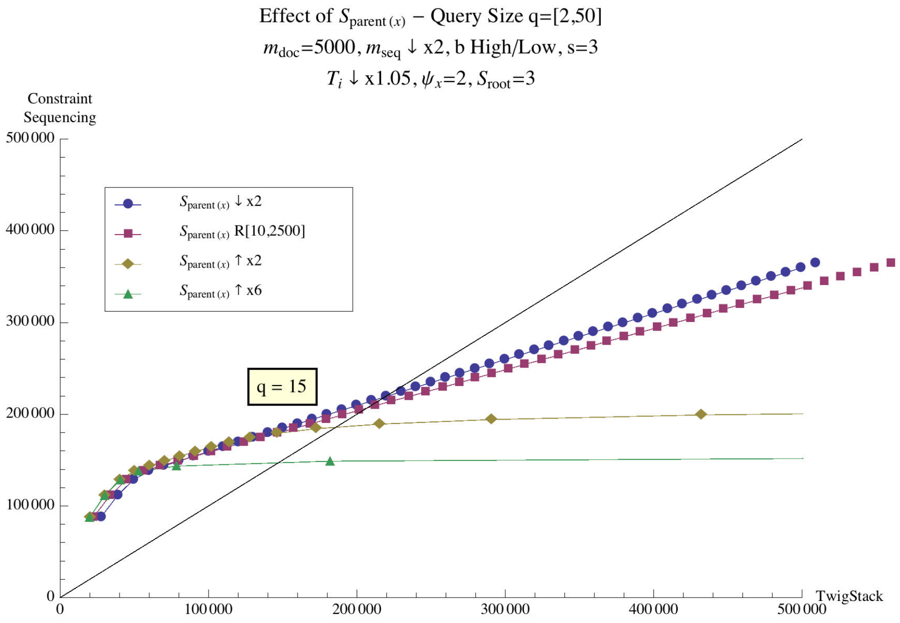


Figure E.14: TS, vary stack size (b high/low, increased s) - Wide

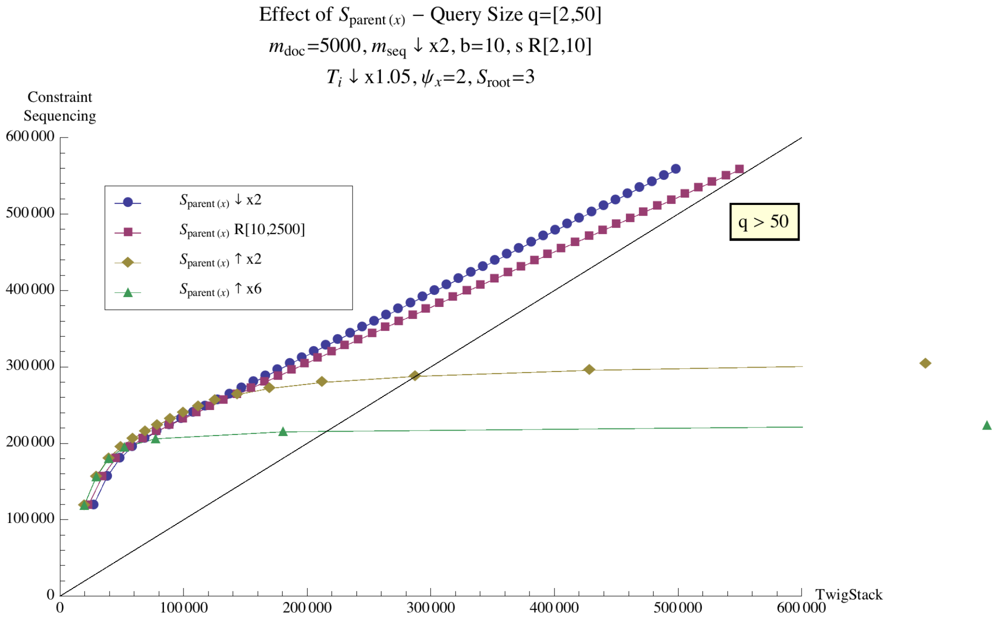


Figure E.15: TS, vary stack size in medium random range (random s) - Wide

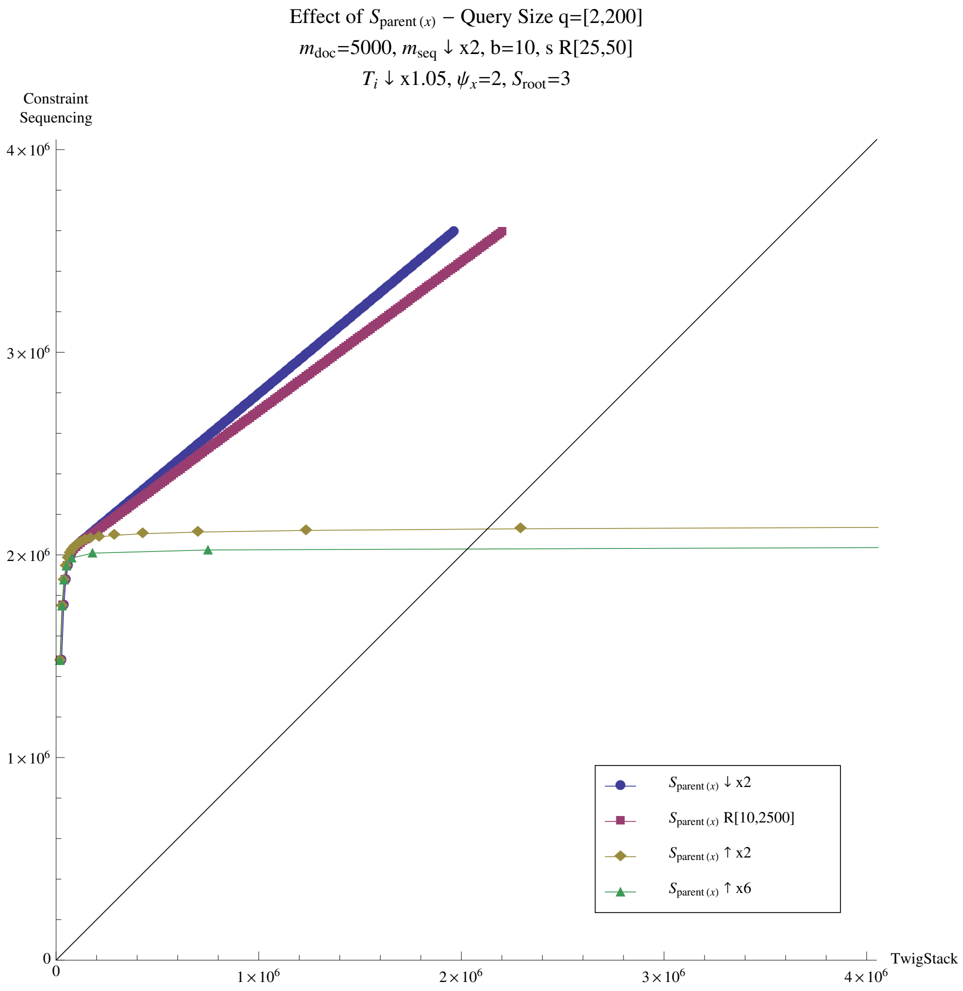


Figure E.16: TS, vary stack size in medium random range (high random s) - Wide

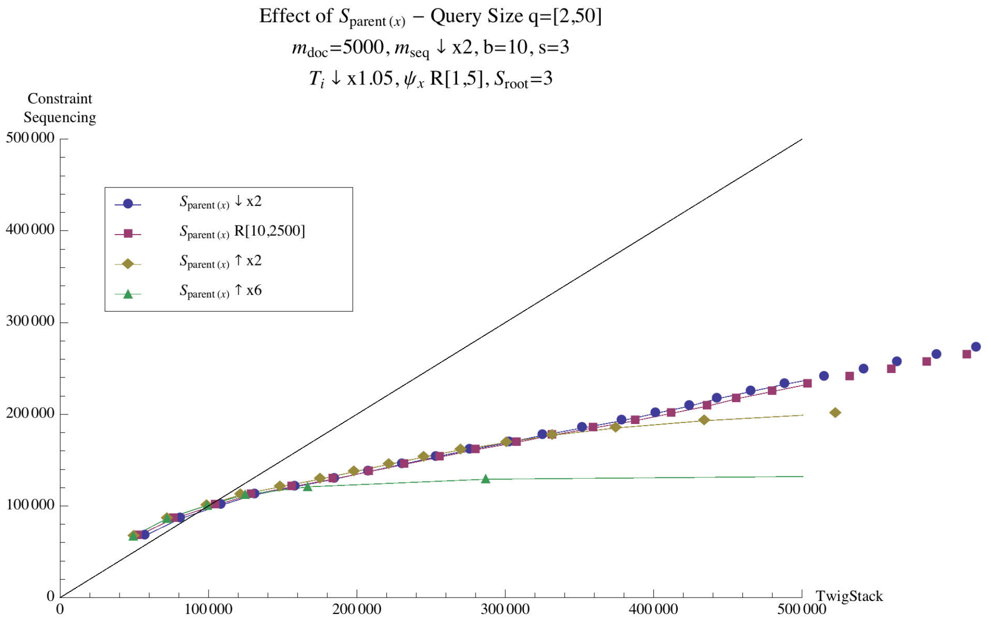


Figure E.17: TS, vary stack size (increased s, high ψ_x) - Wide

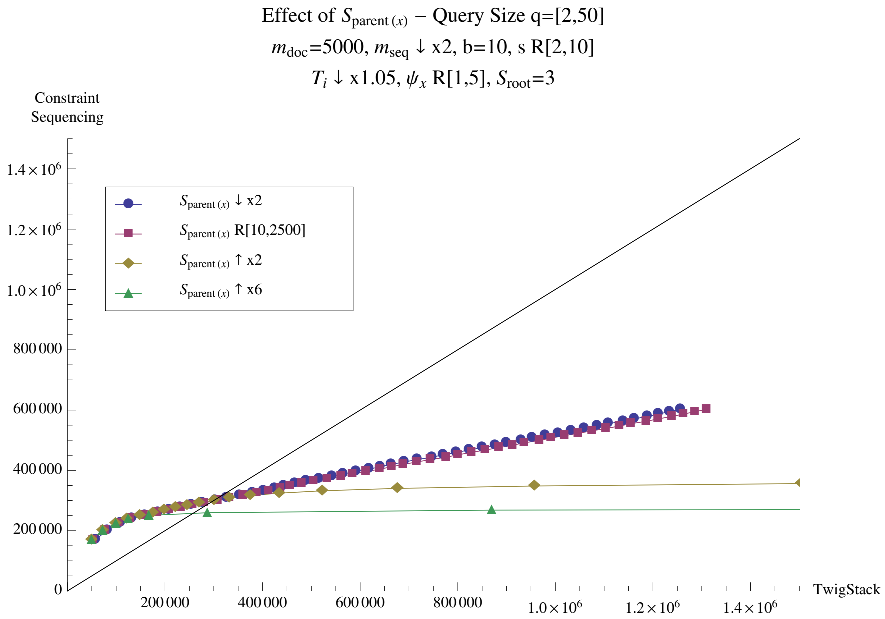


Figure E.18: TS, vary stack size (low random s, high ψ_x) - Wide

Appendix F

Native Comparison Graphs (Similar Depth and Breadth)

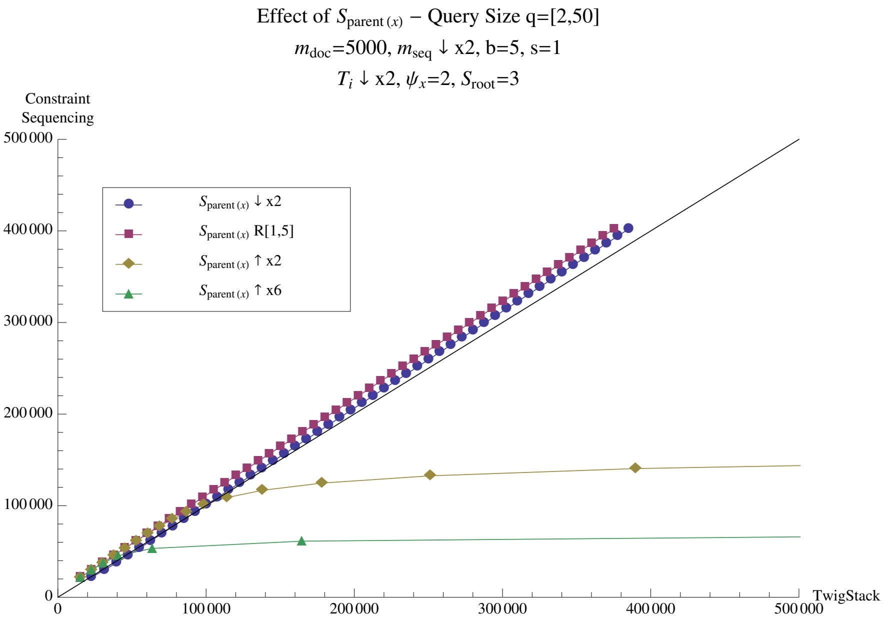


Figure F.1: TS, vary stack size in small random range (low s) - Similar Depth/Breadth

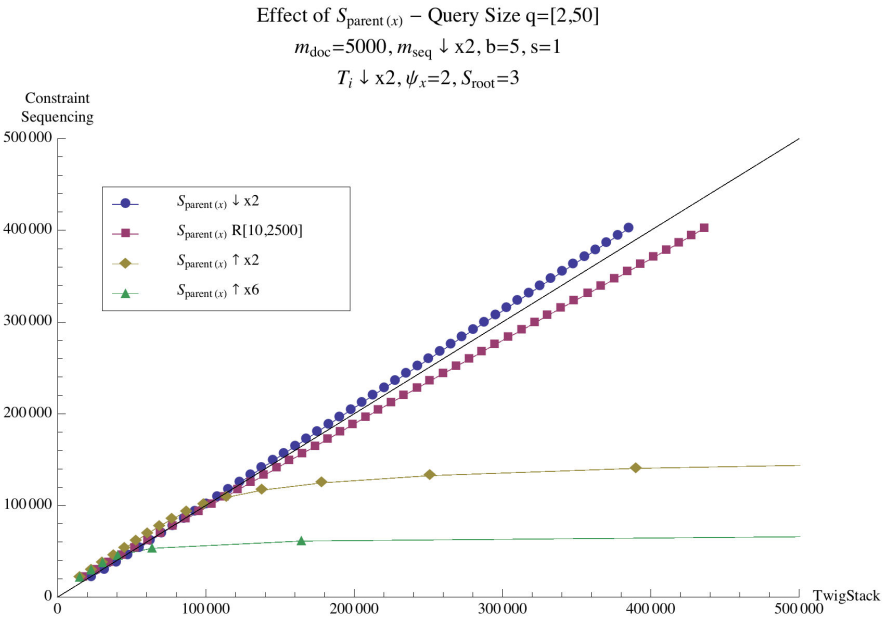


Figure F.2: TS, vary stack size (low s) - Similar Depth/Breadth

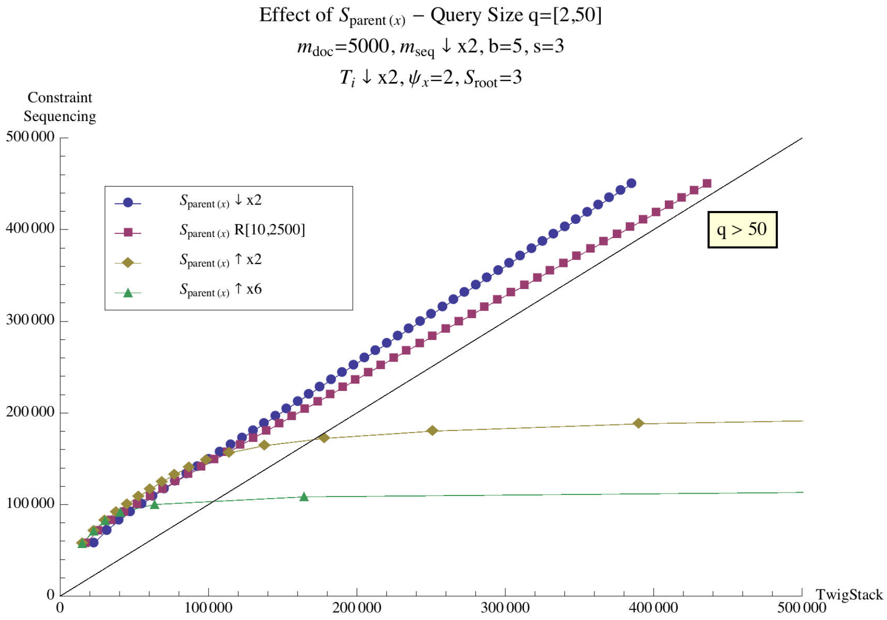


Figure F.3: TS, vary stack size (increased s) - Similar Depth/Breadth

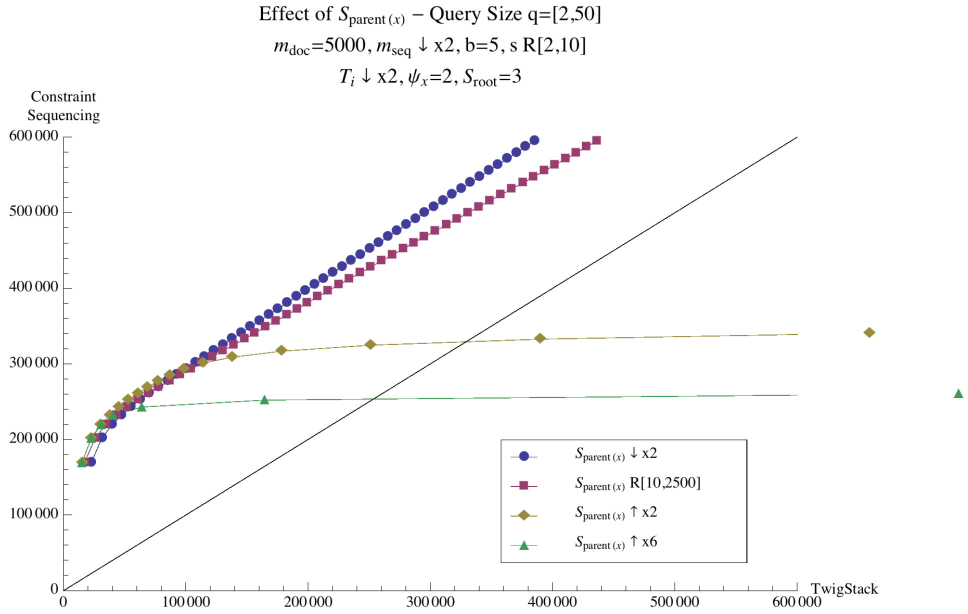


Figure F.4: TS, vary stack size in medium random range (random s) - Similar Depth/Breadth

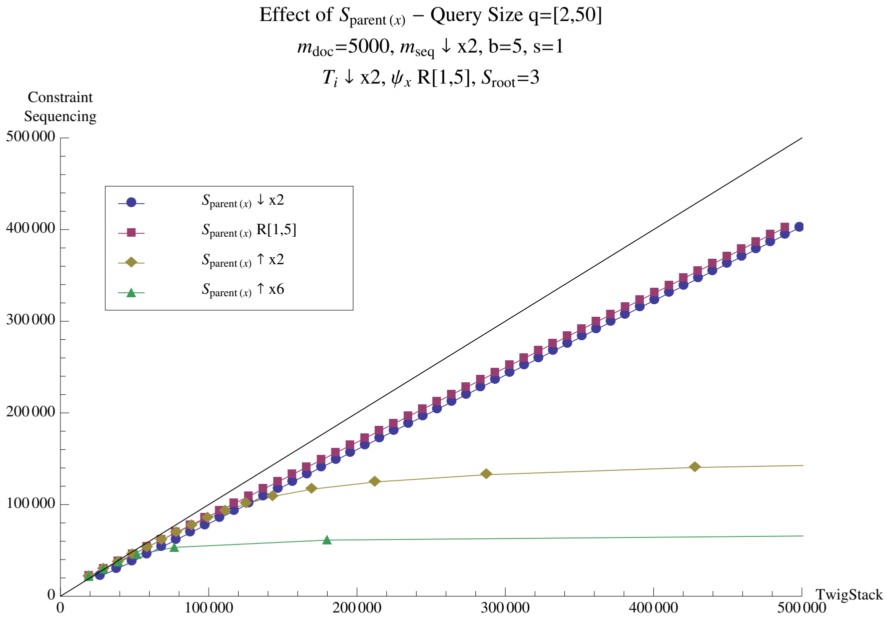


Figure F.5: TS, vary stack size in small random range (low s, high ψ_x) - Similar Depth/Breadth

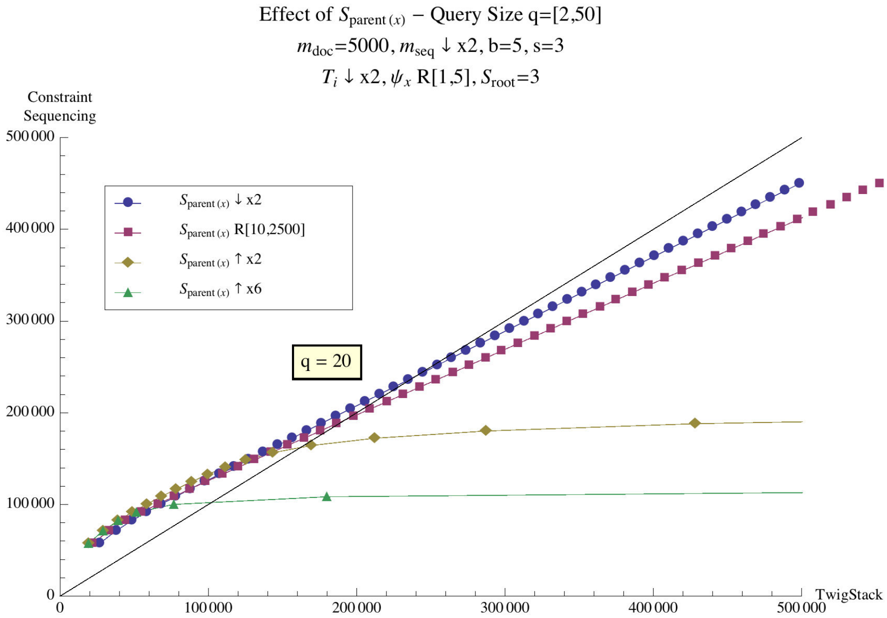


Figure F.6: TS, vary stack size (increased s, high ψ_x) - Similar Depth/Breadth

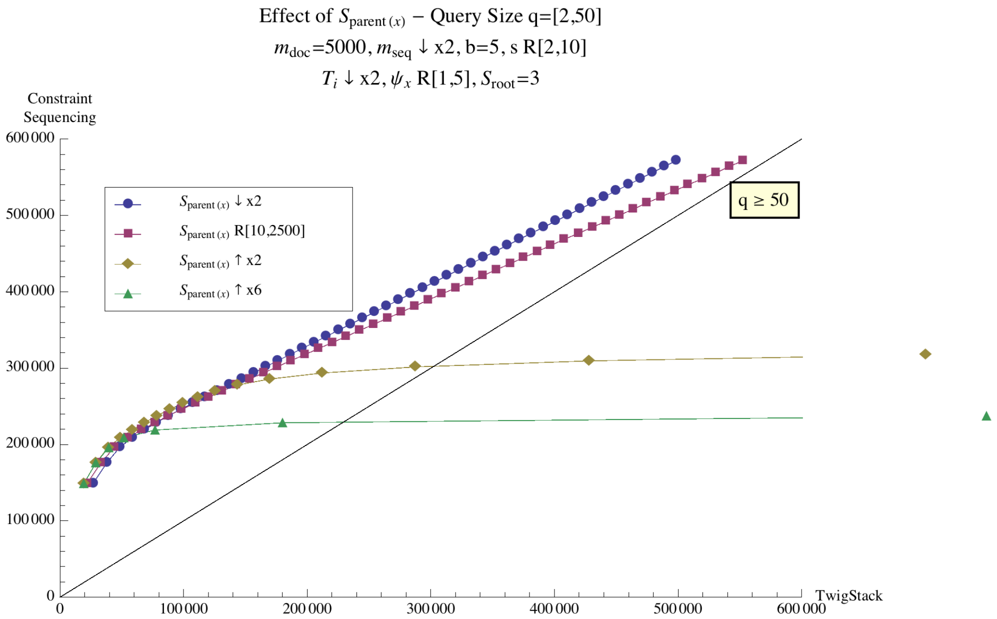


Figure F.7: TS, vary stack size (low random s, high ψ_x) - Similar Depth/Breadth

Appendix G

Native Comparison Graphs (DBLP XML Dataset)

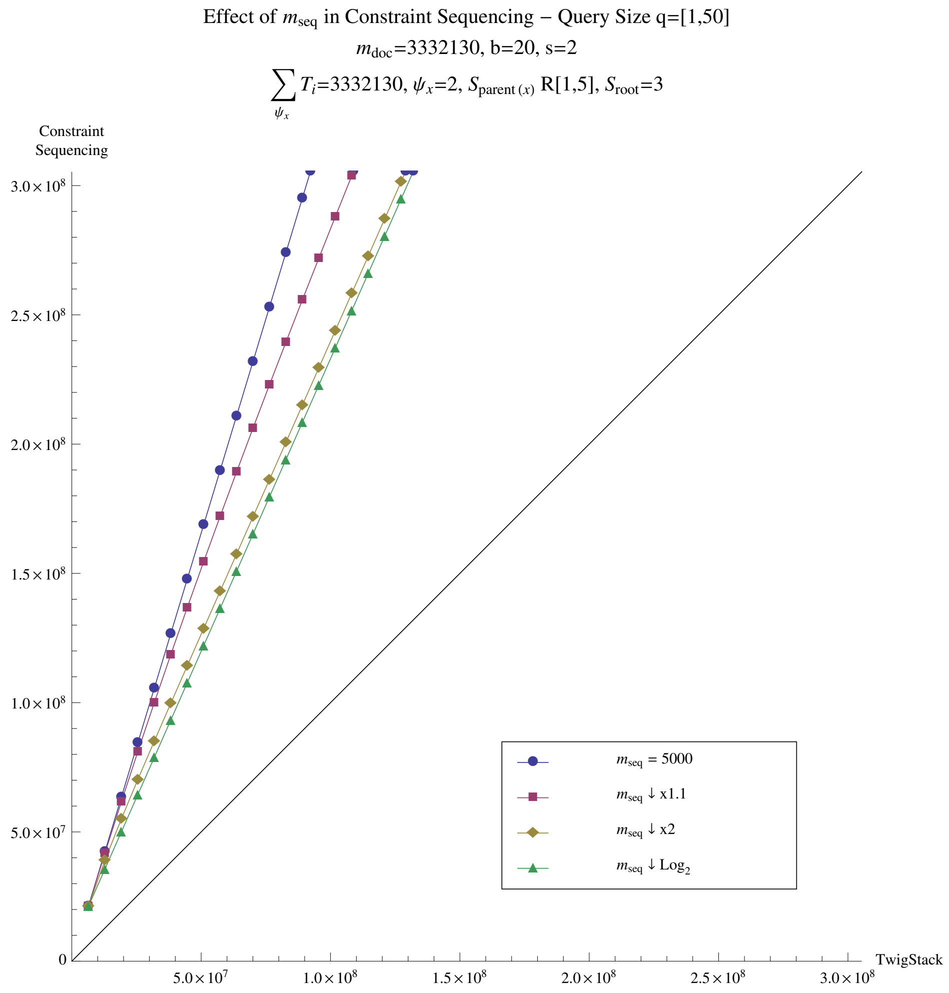


Figure G.1: CS, vary sequence size (low random $S_{parent(x)}$, low ψ_x) - DBLP

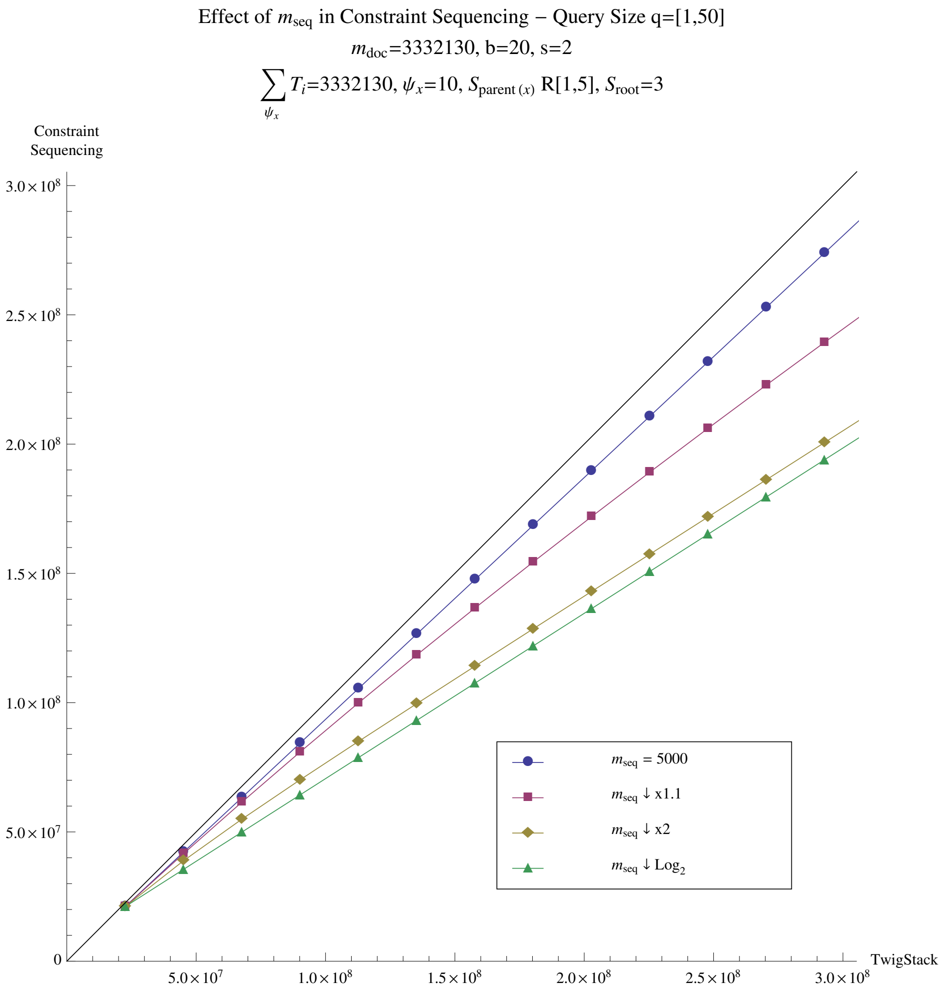


Figure G.2: CS, vary sequence size (low random $S_{parent(x)}$, high ψ_x) - DBLP

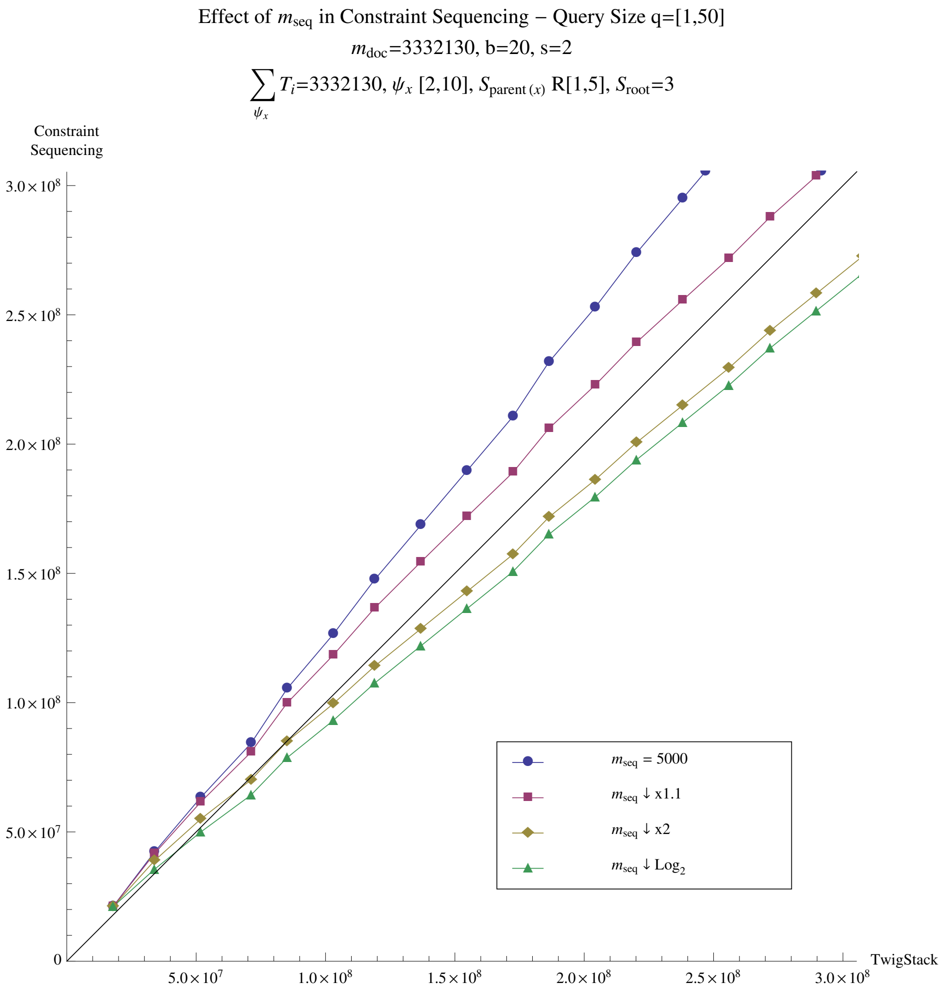


Figure G.3: CS, vary sequence size (low random $S_{parent(x)}$, random ψ_x) - DBLP

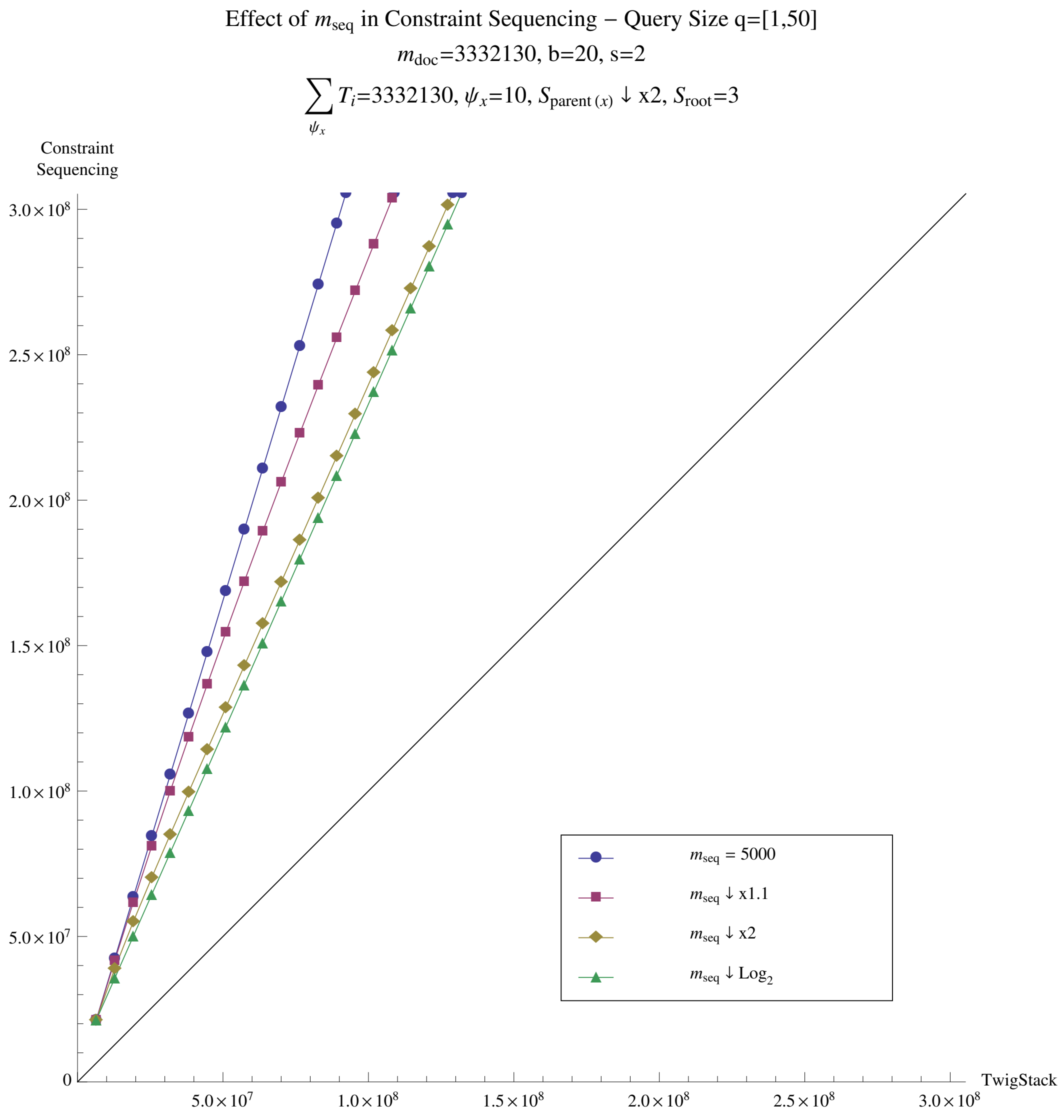


Figure G.4: CS, vary sequence size (decreasing $S_{parent(x)}$) - DBLP

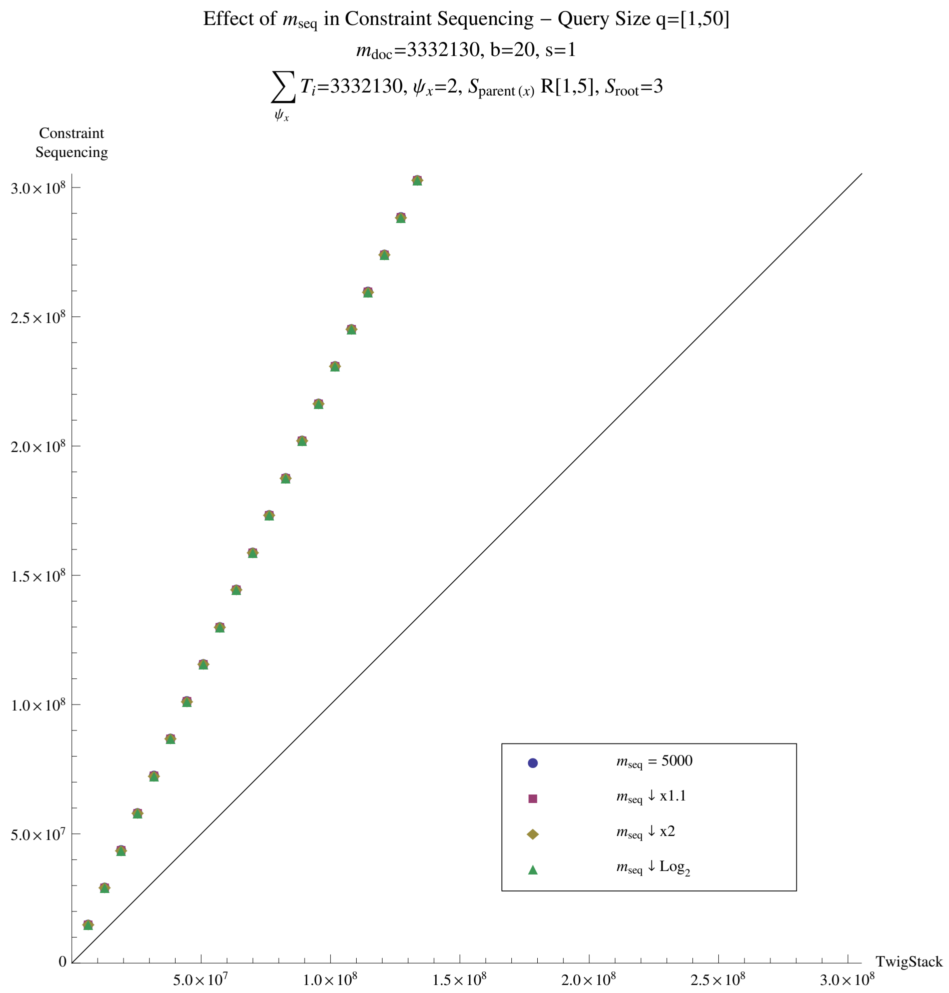


Figure G.5: CS, vary sequence size (low s, low ψ_x) - DBLP

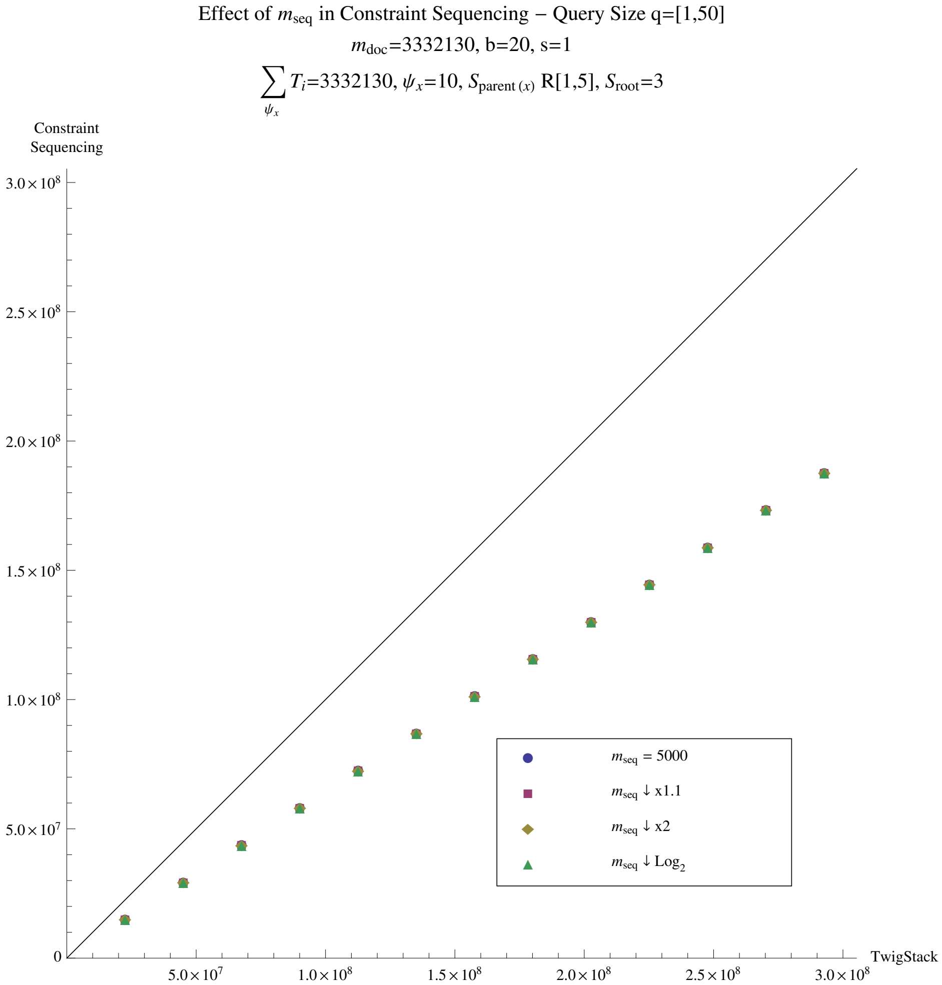


Figure G.6: CS, vary sequence size (low s, high ψ_x) - DBLP

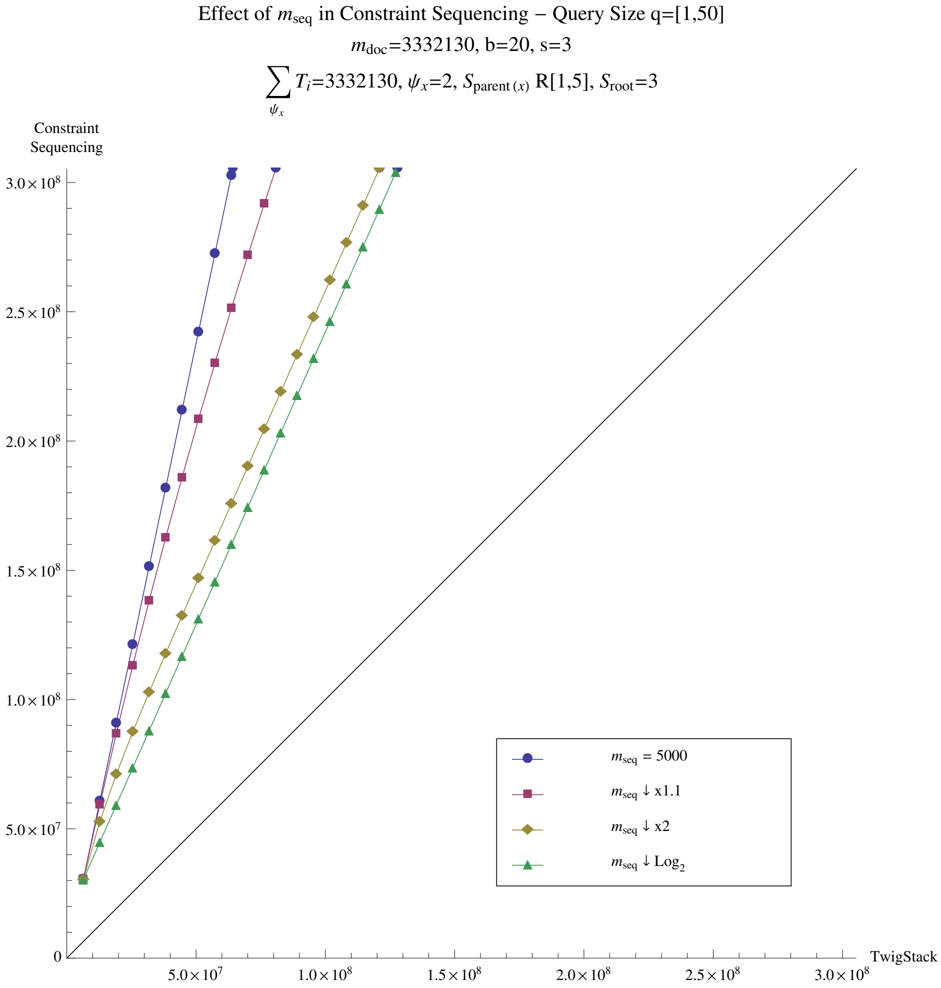


Figure G.7: CS, vary sequence size (increased s) - DBLP

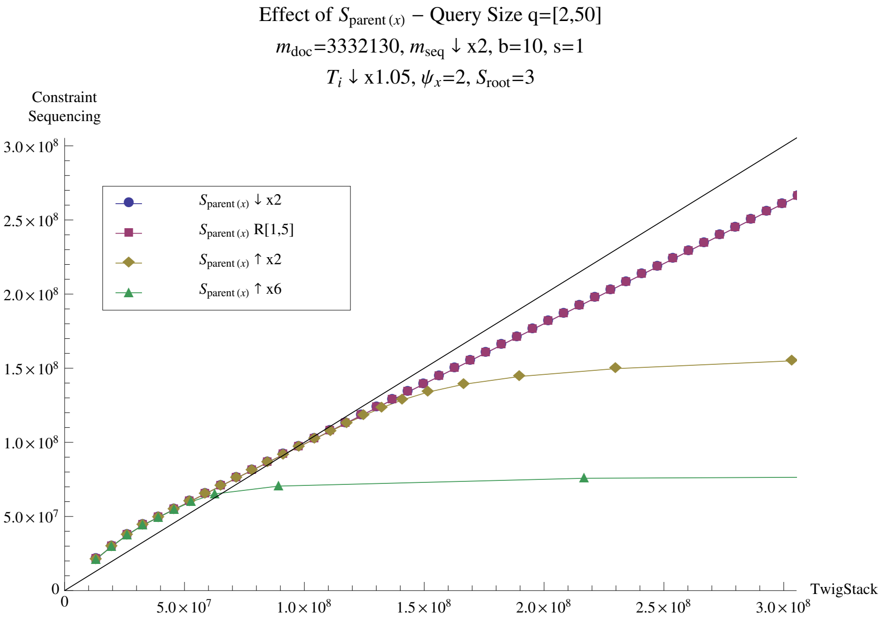


Figure G.8: TS, vary stack size in small random range (low s) - DBLP

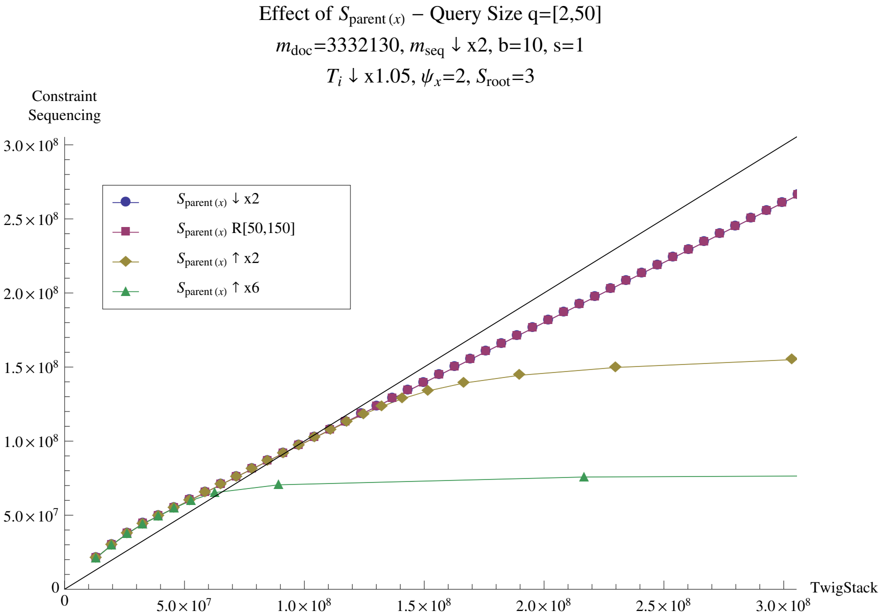


Figure G.9: TS, vary stack size in medium random range (low s) - DBLP

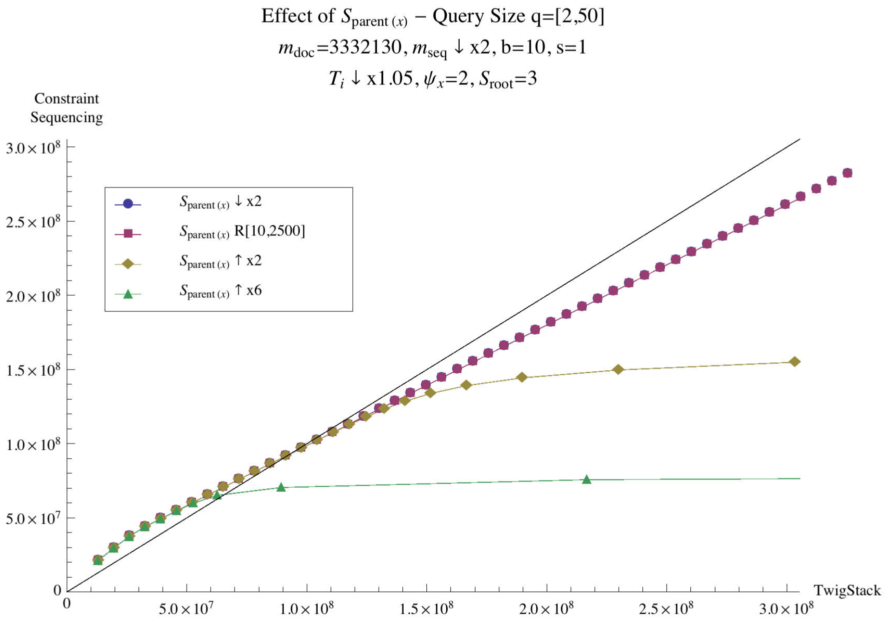


Figure G.10: TS, vary stack size (low s) - DBLP

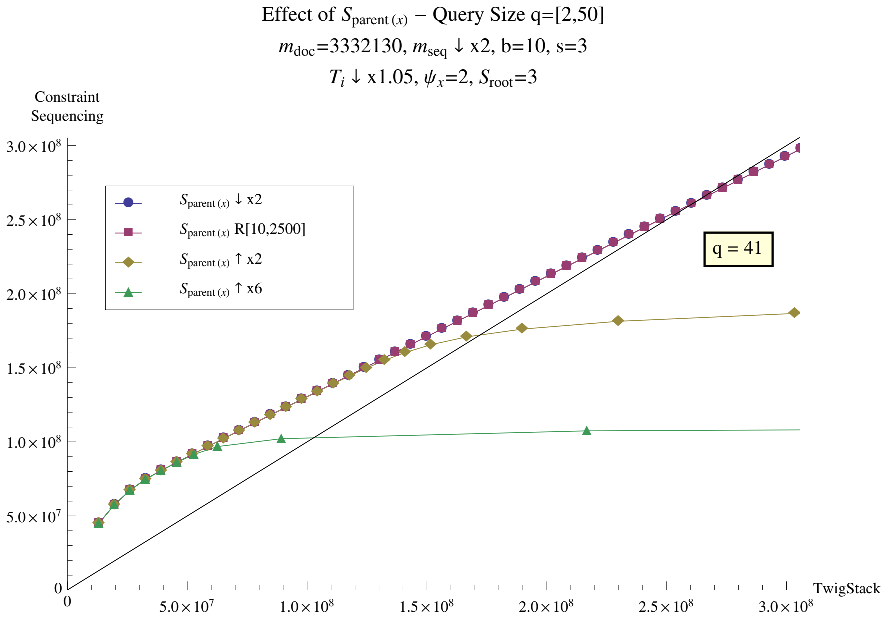


Figure G.11: TS, vary stack size (increased s) - DBLP

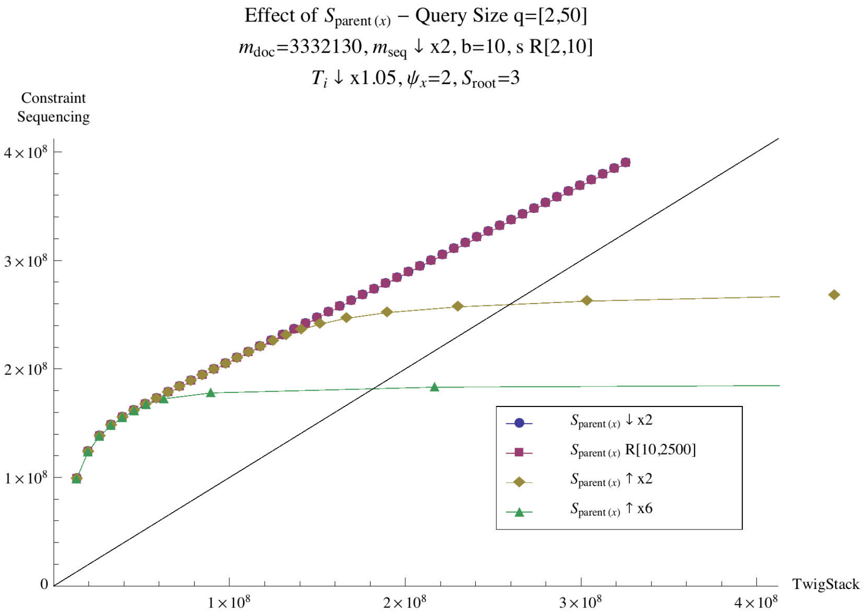


Figure G.12: TS, vary stack size in medium random range (random s) - DBLP

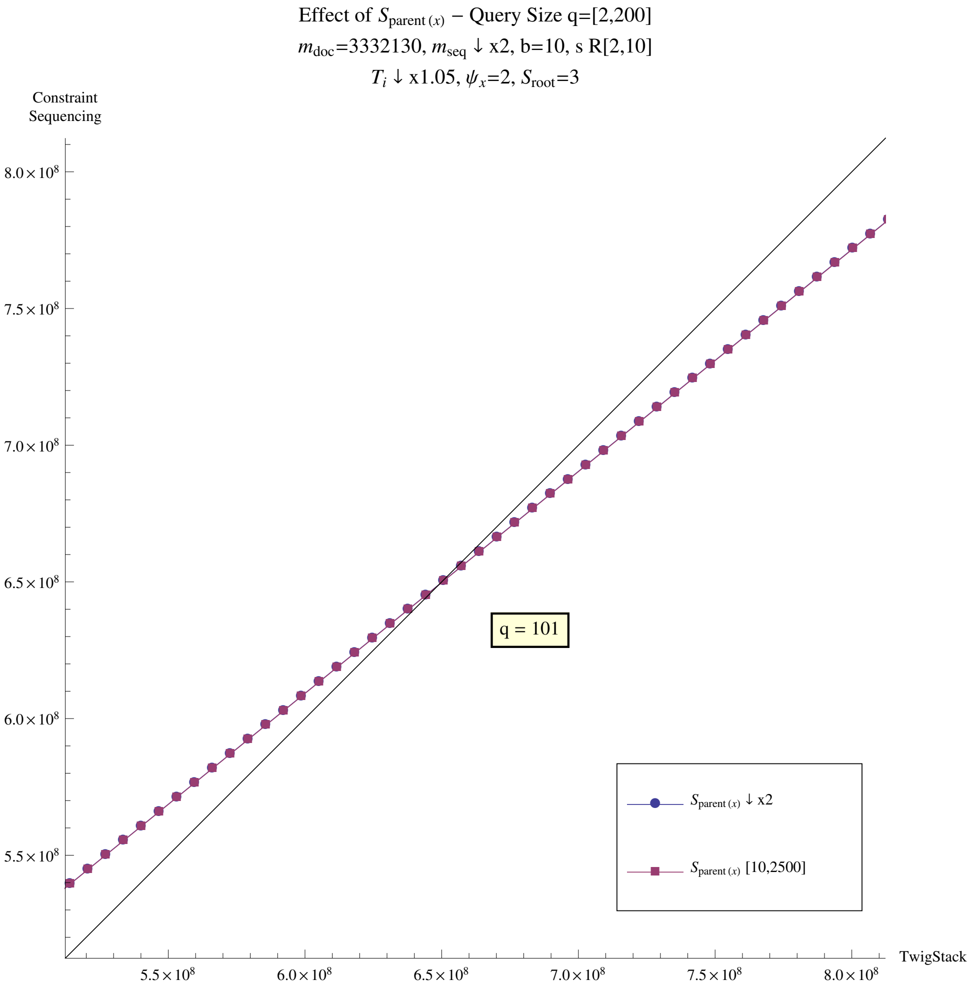


Figure G.13: TS, vary stack size in medium random range (random s, larger query sizes) - DBLP (Zoom)

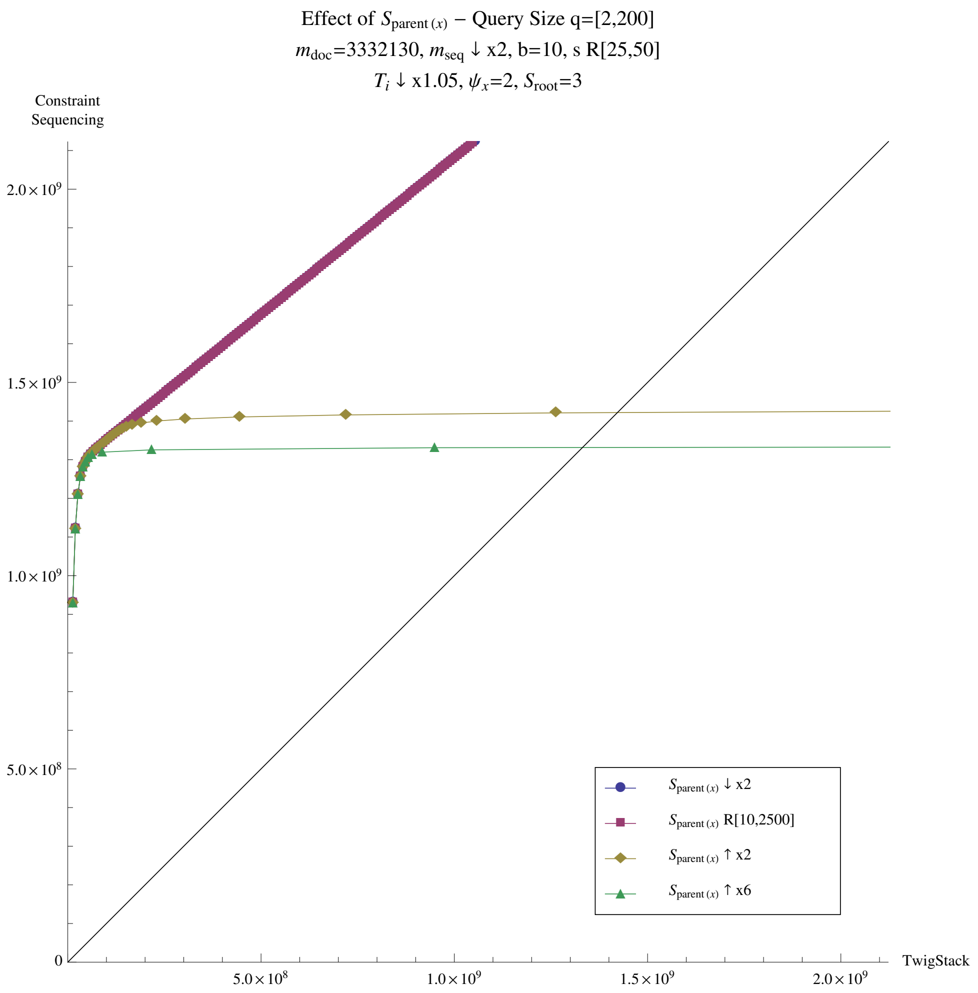


Figure G.14: TS, vary stack size in medium random range (large random s) - DBLP

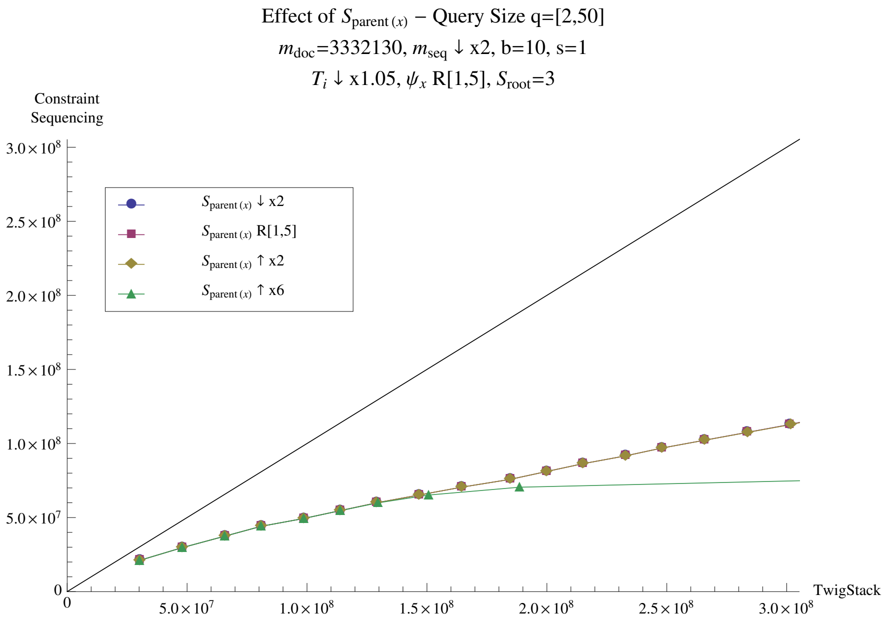


Figure G.15: TS, vary stack size in small random range (low s, high ψ_x) - DBLP

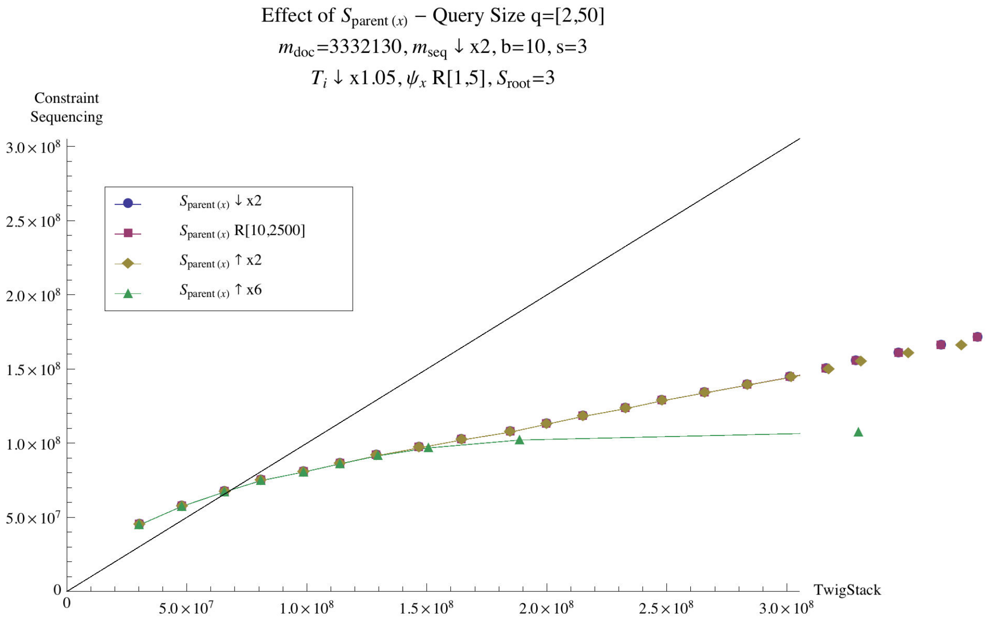


Figure G.16: TS, vary stack size (increased s, high ψ_x) - DBLP

Appendix H

CS/RDBQuery Comparison Graphs

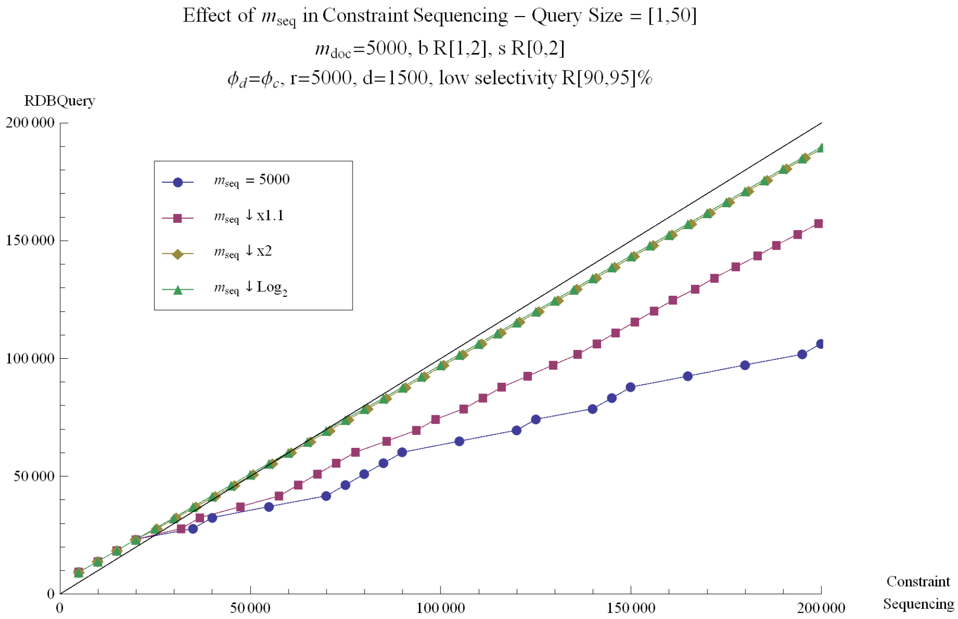


Figure H.1: CS, vary sequence size (low selectivity range) - Deep

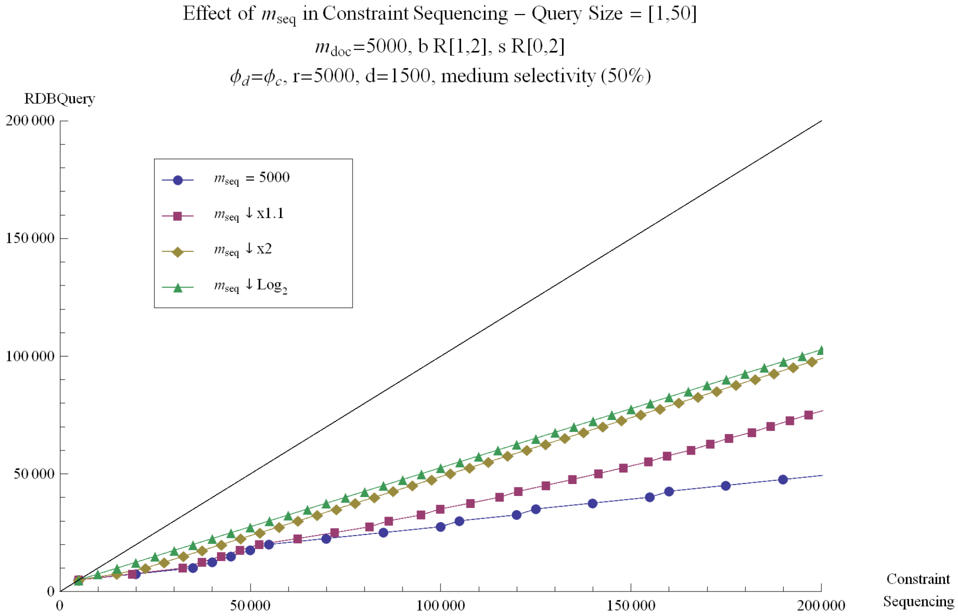


Figure H.2: CS, vary sequence size (medium selectivity) - Deep

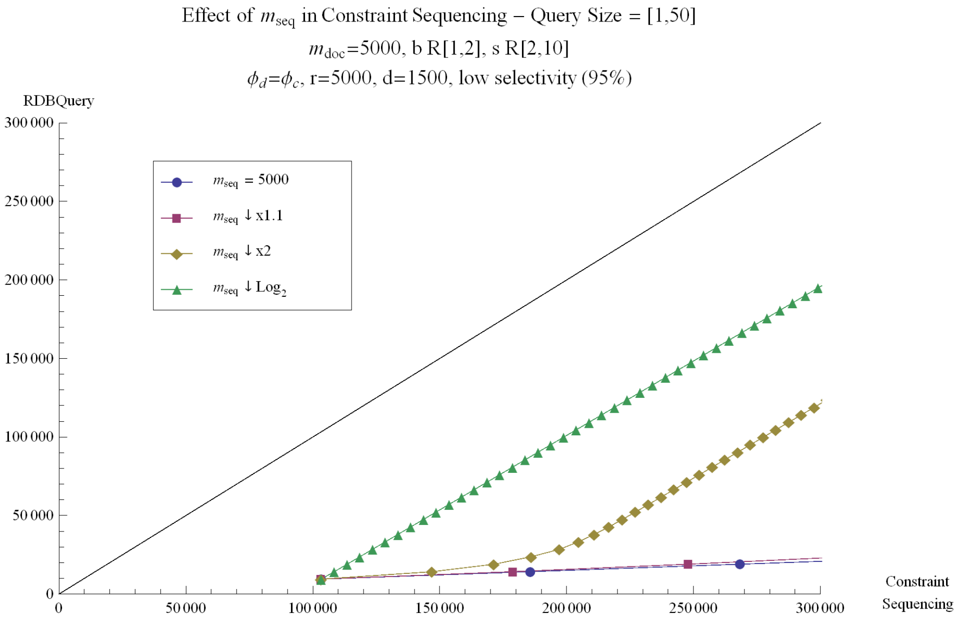


Figure H.3: CS, vary sequence size (low selectivity, increased s) - Deep

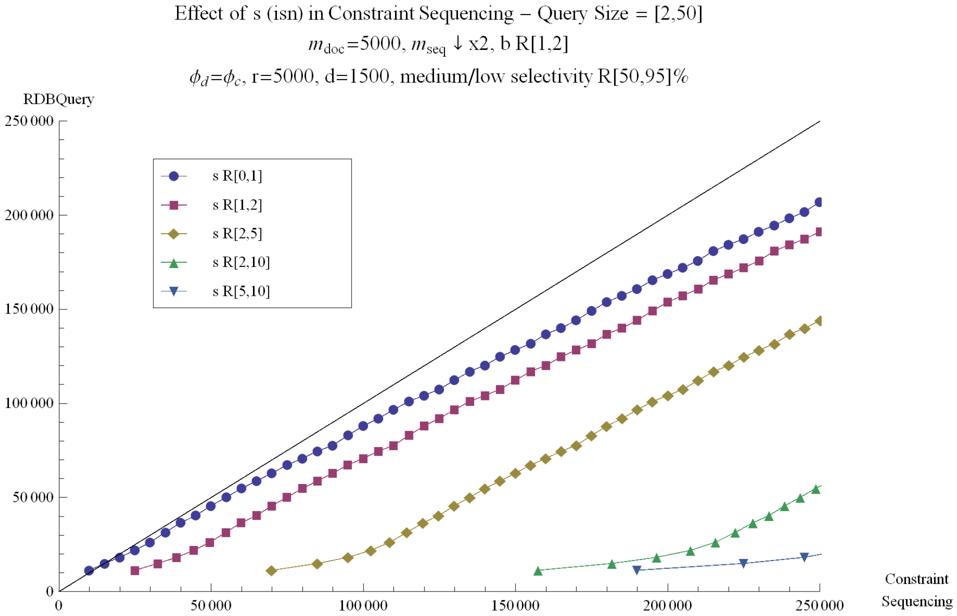


Figure H.4: CS, random identical sibling nodes (medium/low selectivity) - Deep

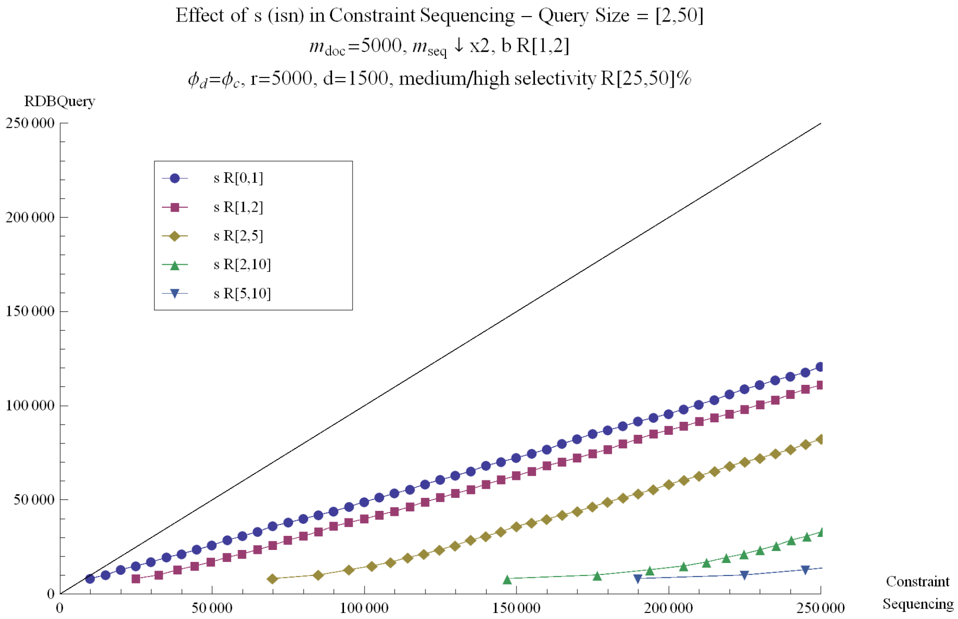


Figure H.5: CS, random identical sibling nodes (medium/high selectivity) - Deep

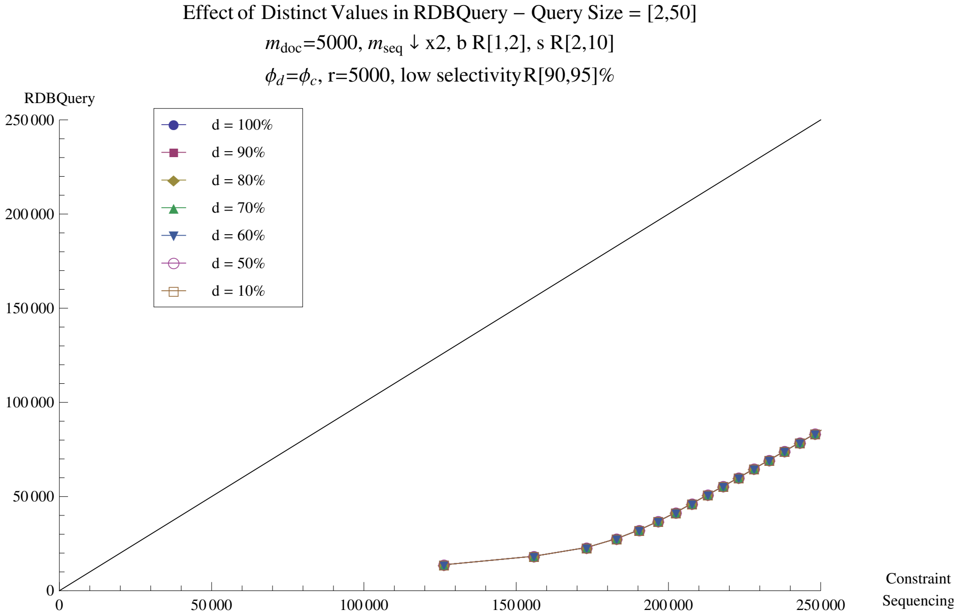


Figure H.6: RDBQuery, vary distinct values (low selectivity, high s) - Deep

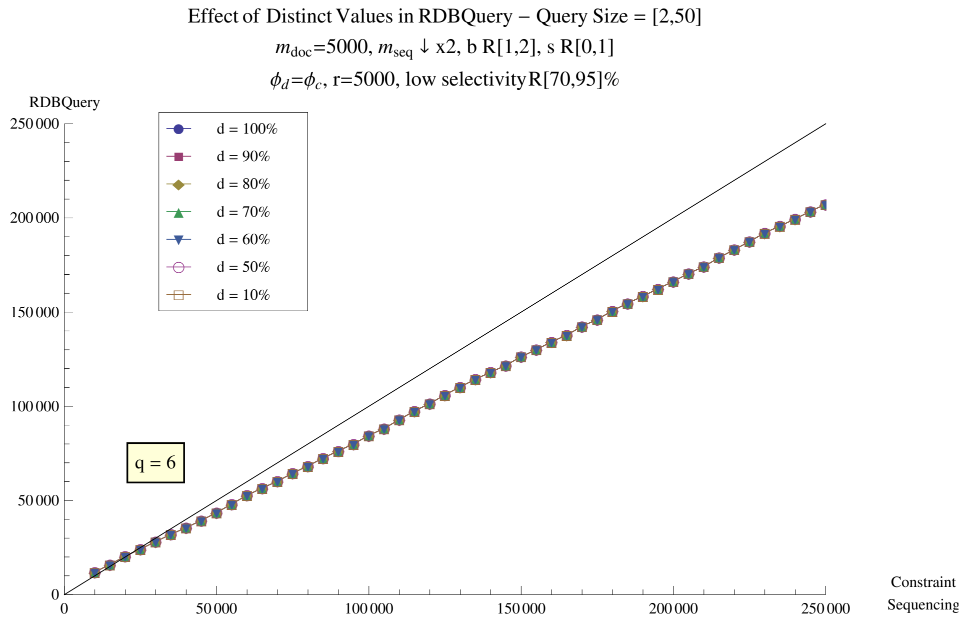


Figure H.7: RDBQuery, vary distinct values (increased selectivity, low s) - Deep

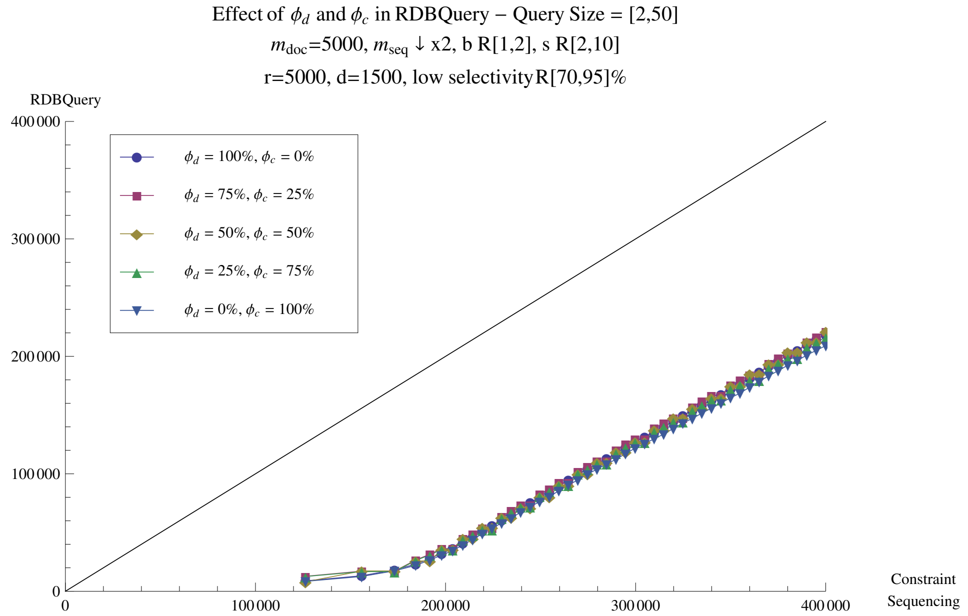


Figure H.8: RDBQuery, query edge distribution (increased selectivity, high s) - Deep

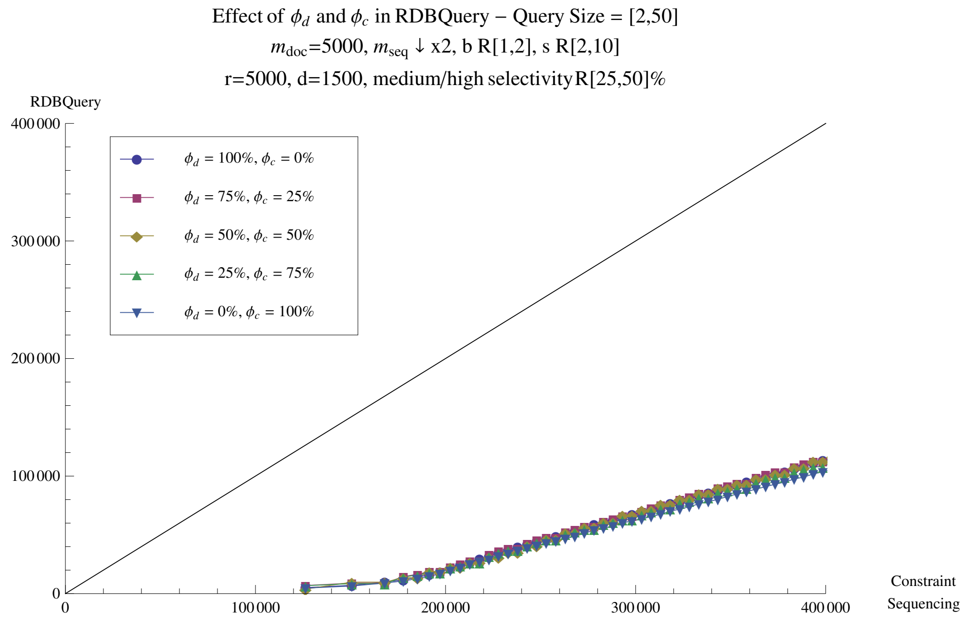


Figure H.9: RDBQuery, query edge distribution (high selectivity, high s) - Deep

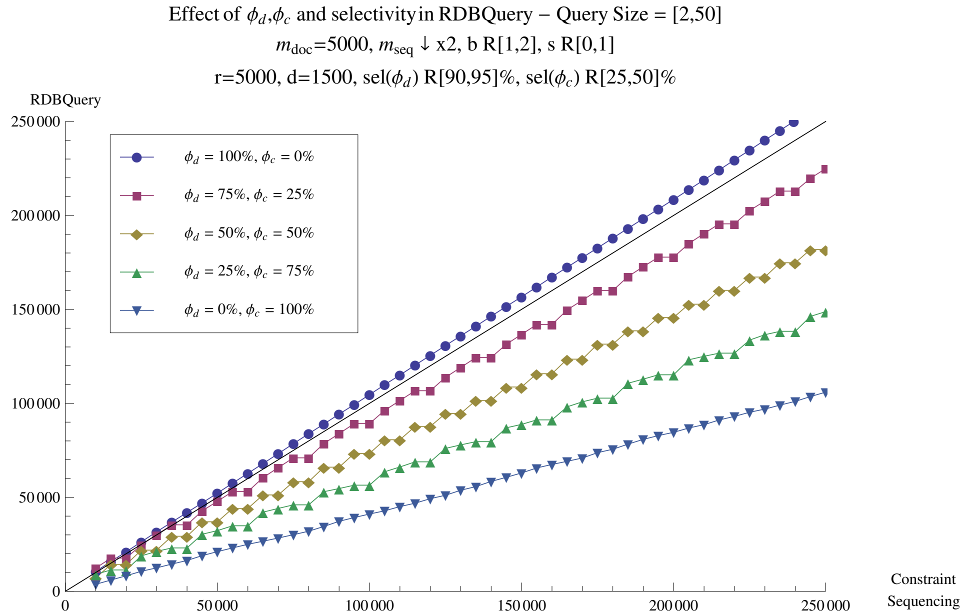


Figure H.10: RDBQuery, low sel(ϕ_d) < medium/high sel(ϕ_c) - Deep

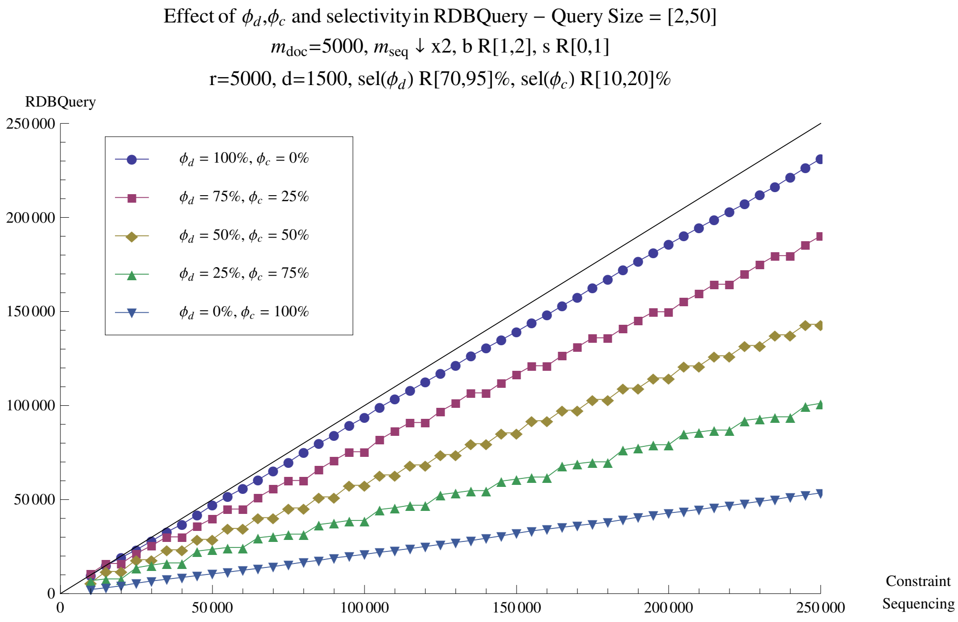


Figure H.11: RDBQuery, low sel(ϕ_d) < high sel(ϕ_c) - Deep

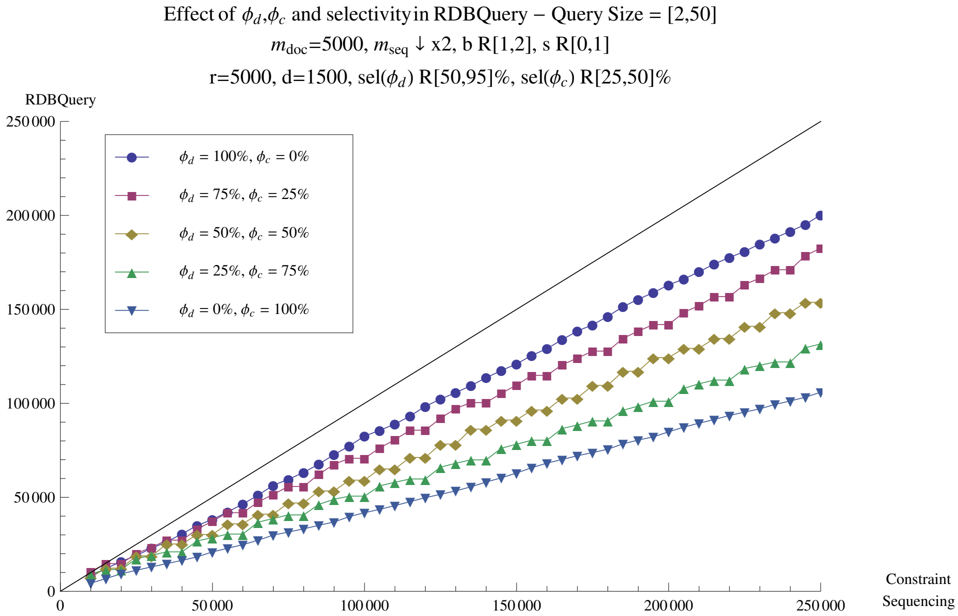


Figure H.12: RDBQuery, medium/low sel(ϕ_d) < medium/high sel(ϕ_c) - Deep

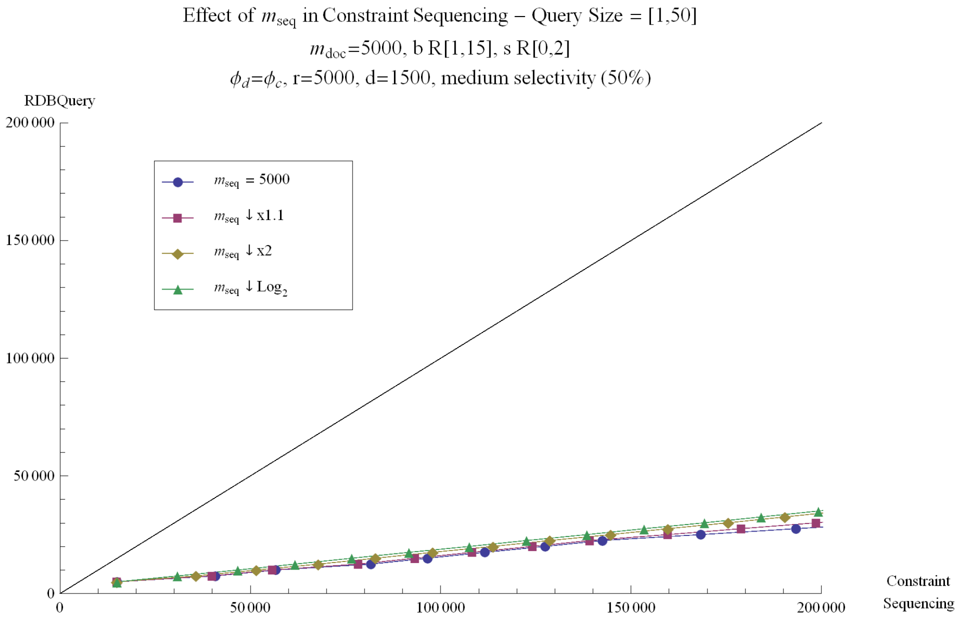


Figure H.13: CS, vary sequence size (medium selectivity) - Wide

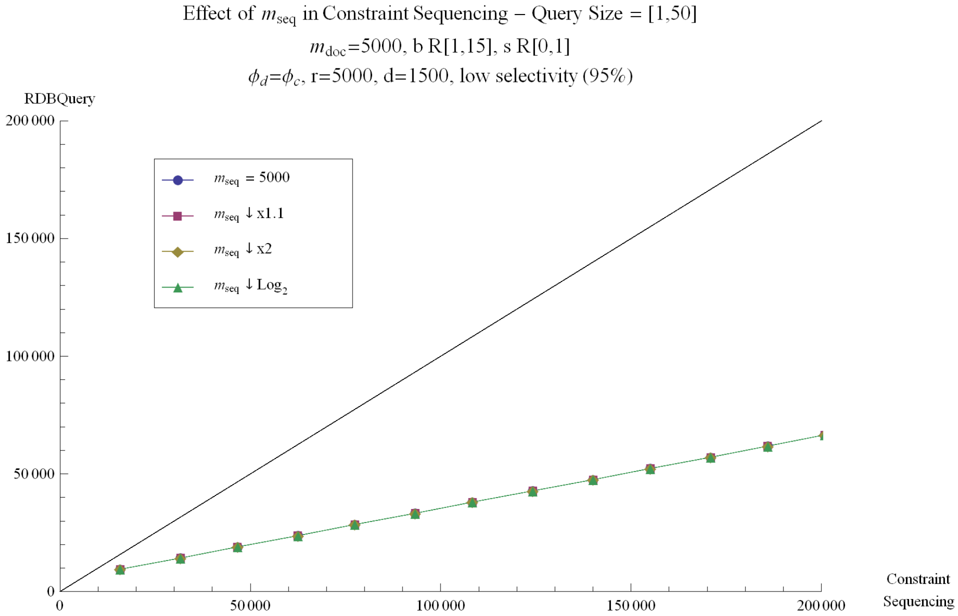


Figure H.14: CS, vary sequence size (low selectivity) - Wide

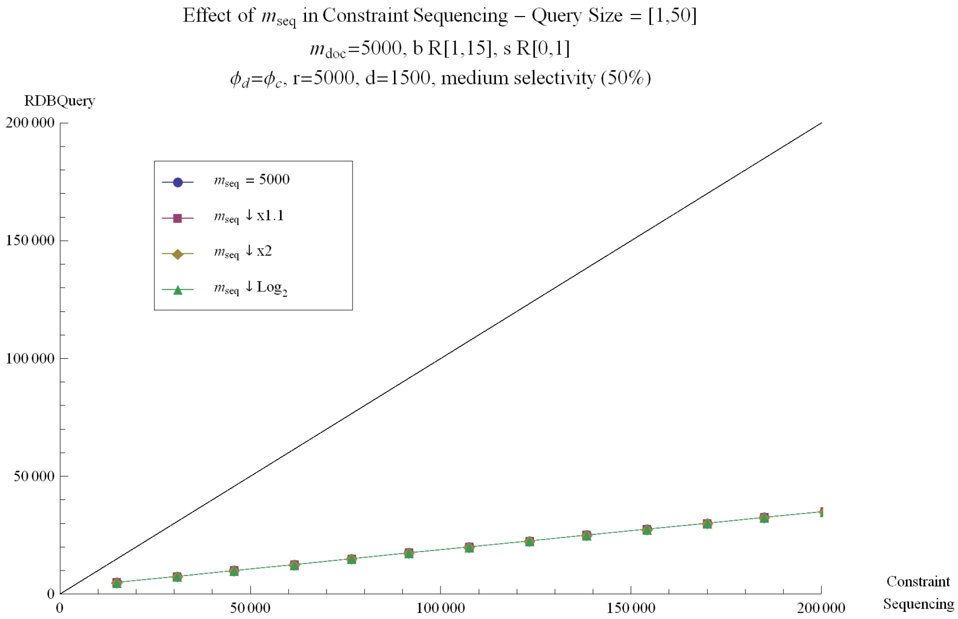


Figure H.15: CS, vary sequence size (medium selectivity, decreased s) - Wide

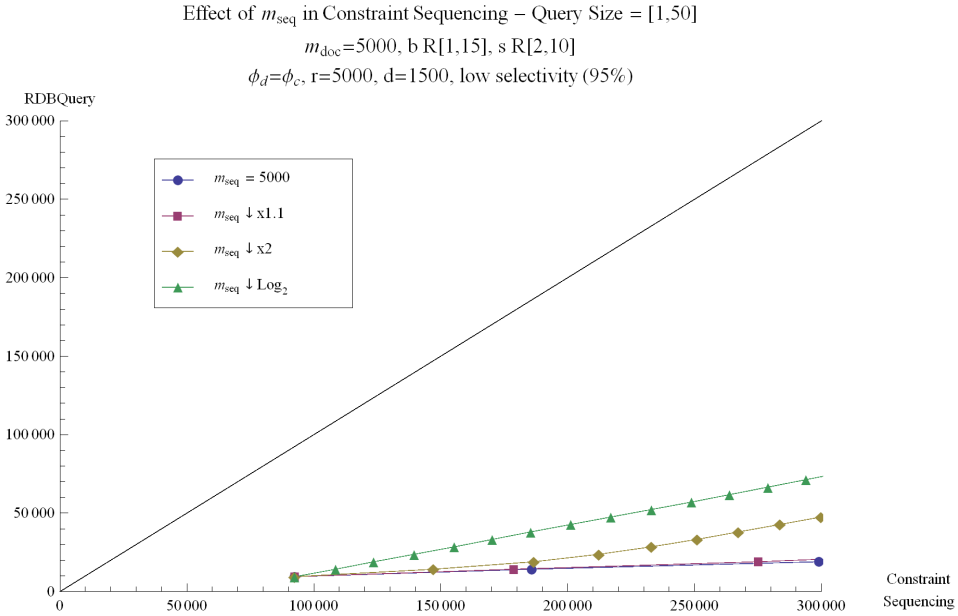


Figure H.16: CS, vary sequence size (low selectivity, increased s) - Wide

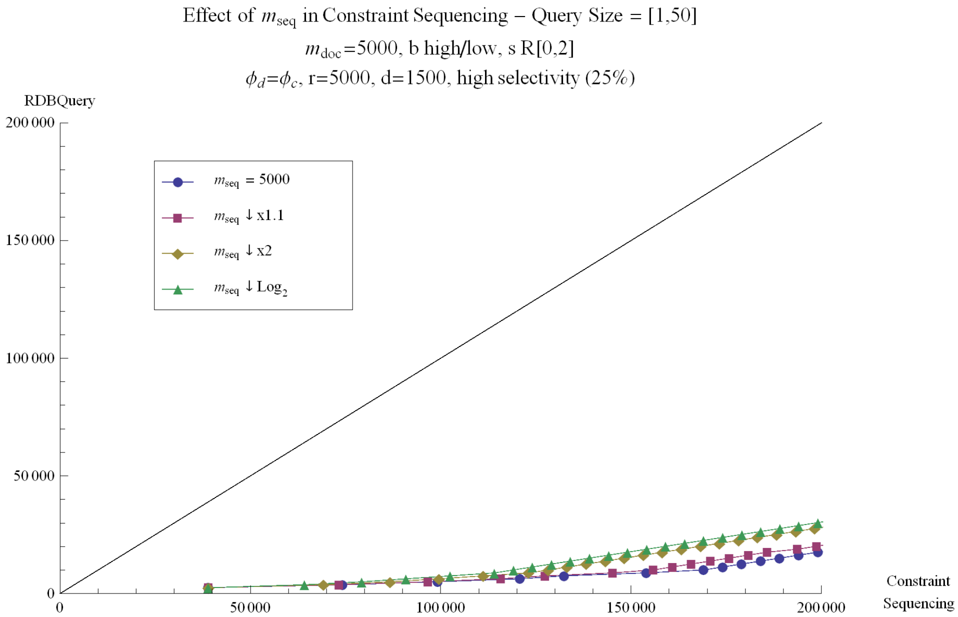


Figure H.17: CS, vary sequence size (high selectivity, b high/low) - Wide

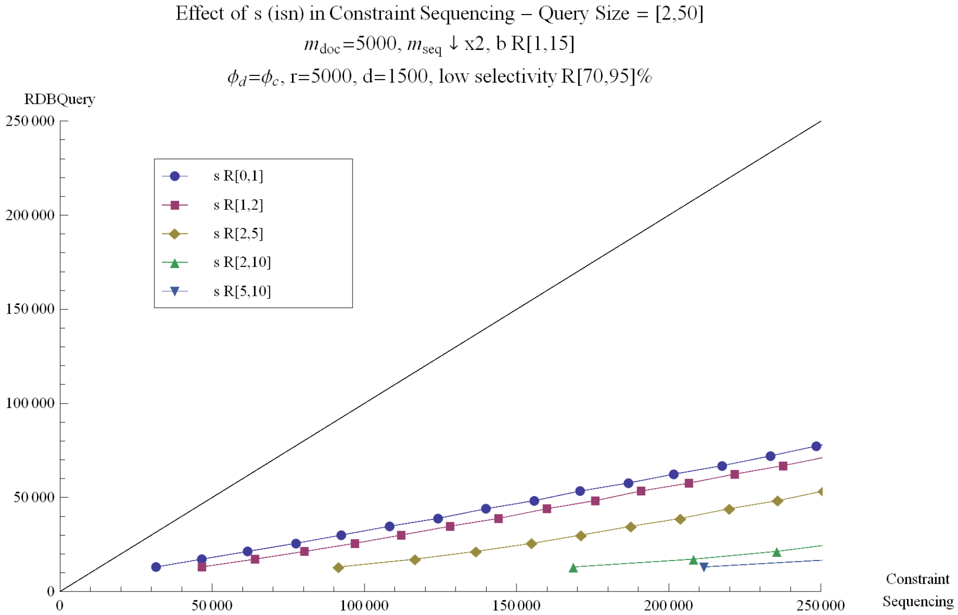


Figure H.18: CS, random identical sibling nodes (increased low selectivity) - Wide

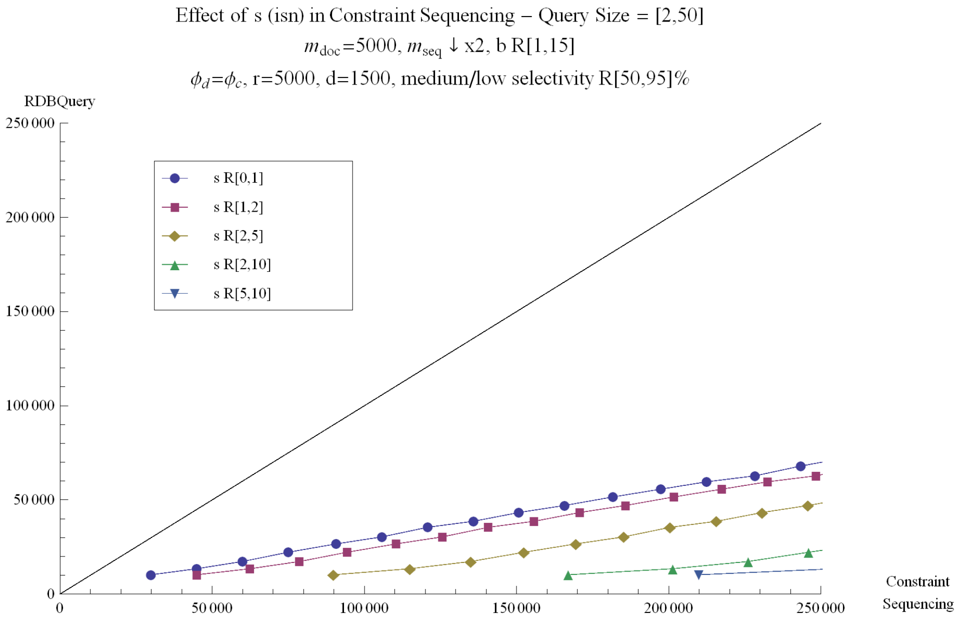


Figure H.19: CS, random identical sibling nodes (medium/low selectivity) - Wide

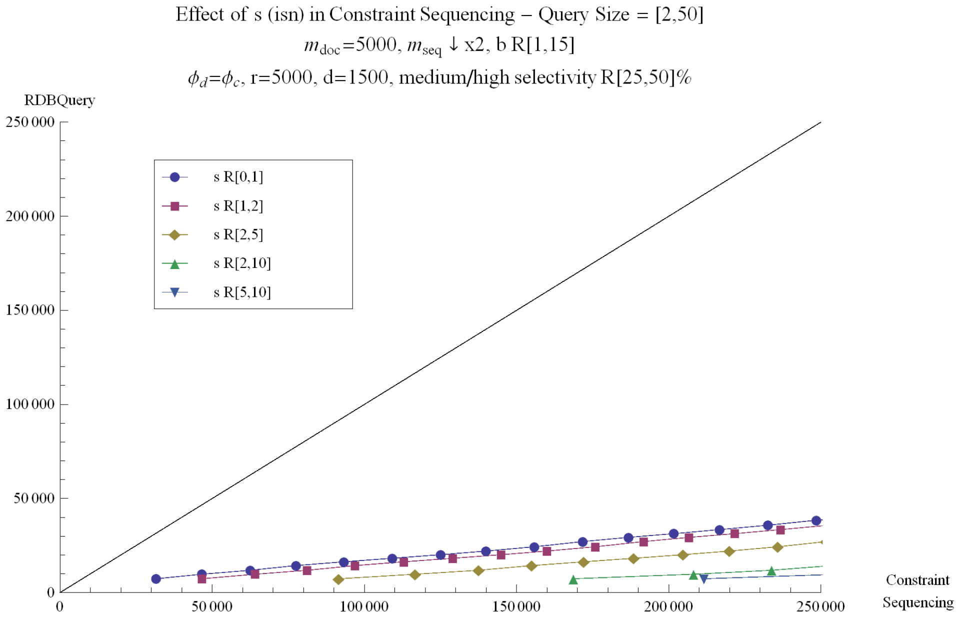


Figure H.20: CS, random identical sibling nodes (medium/high selectivity) - Wide

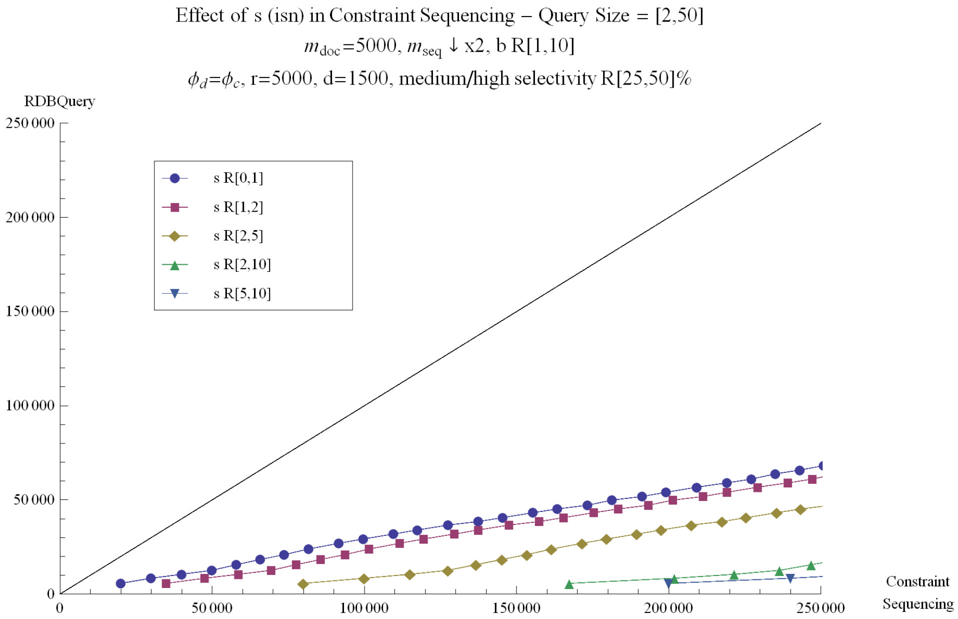


Figure H.21: CS, random identical sibling nodes (medium/high selectivity, decreased b) - Wide

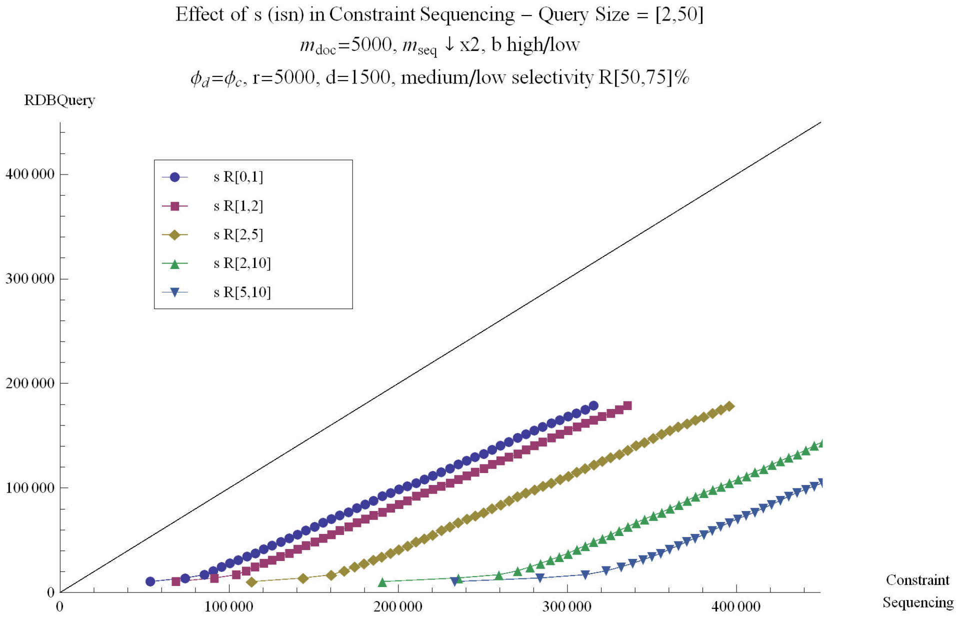


Figure H.22: CS, random identical sibling nodes (medium/low selectivity, b high/low) - Wide

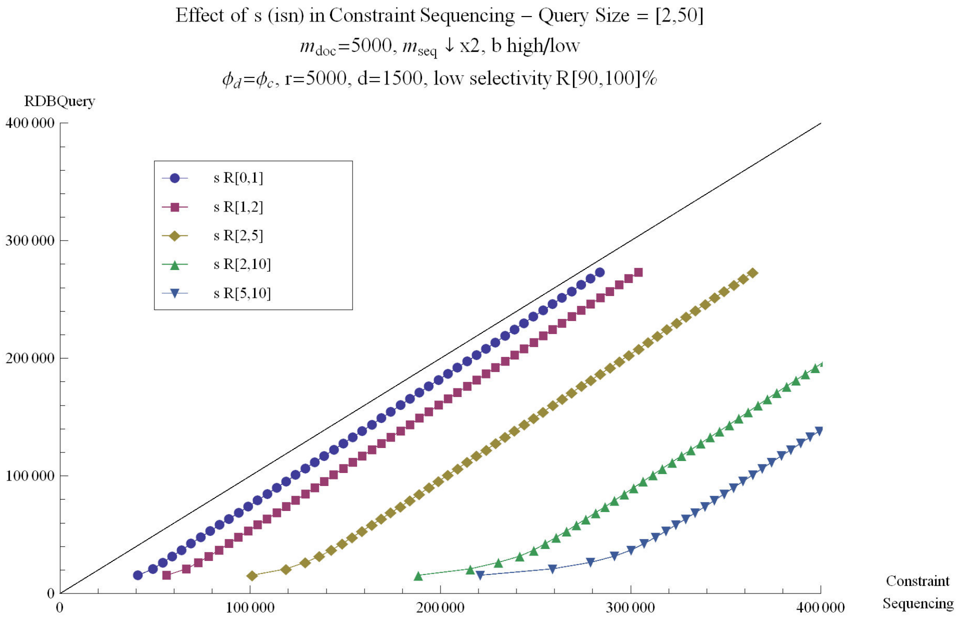


Figure H.23: CS, random identical sibling nodes (low selectivity, b high/low) - Wide

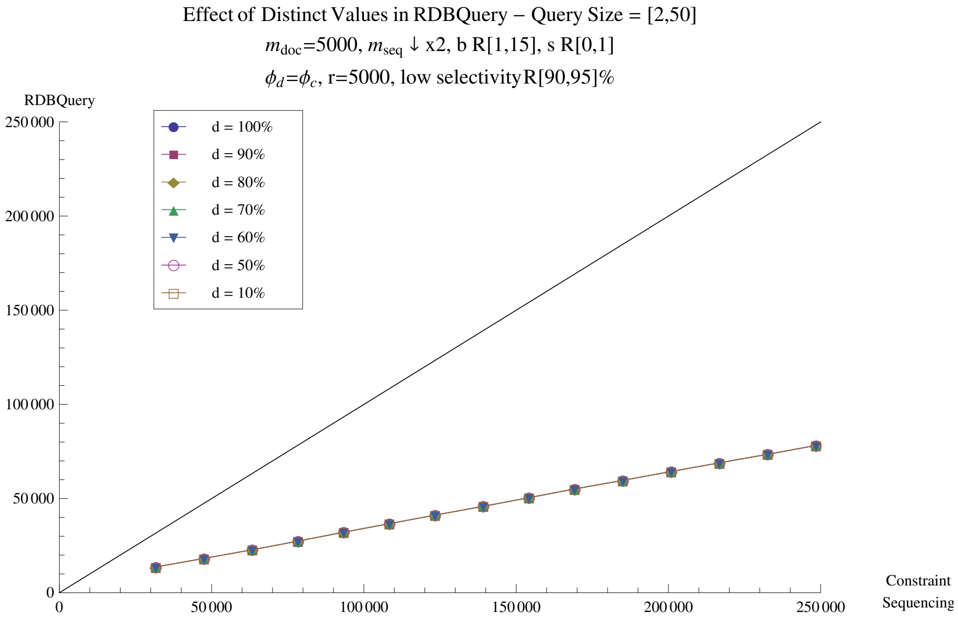


Figure H.24: RDBQuery, vary distinct values (low selectivity, low s) - Wide

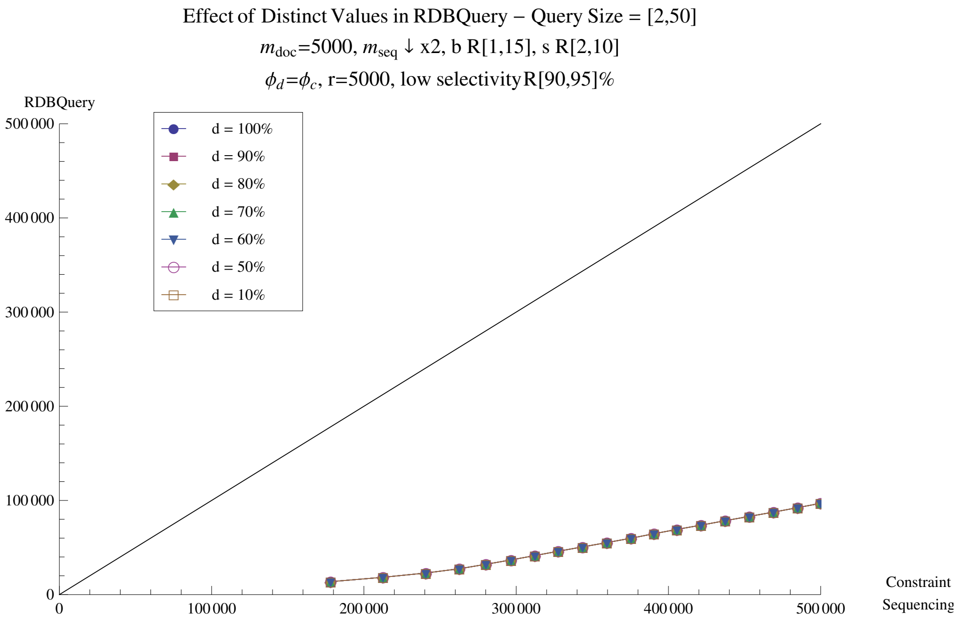


Figure H.25: RDBQuery, vary distinct values (low selectivity, high s) - Wide

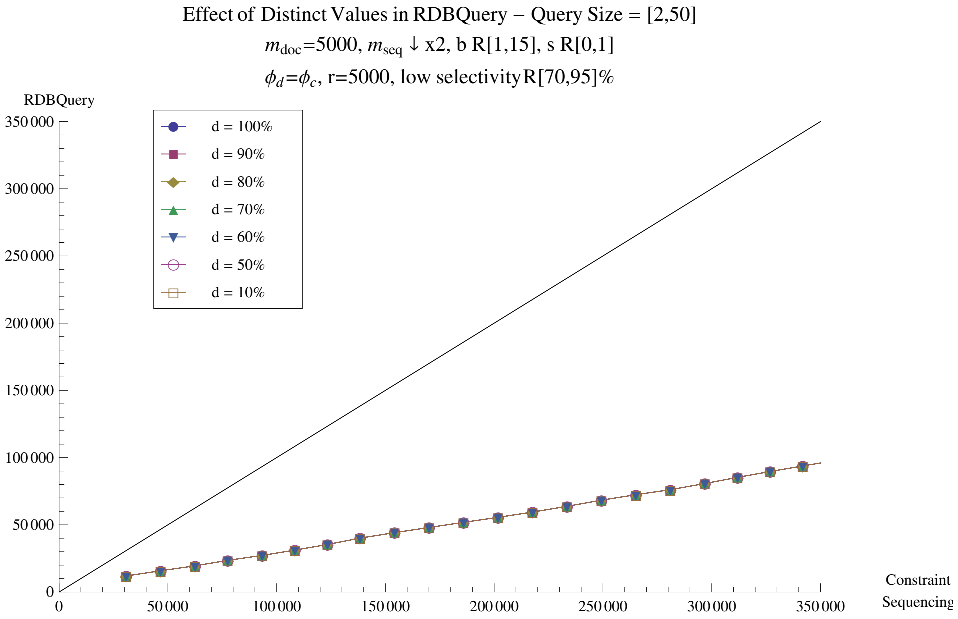


Figure H.26: RDBQuery, vary distinct values (decreased low selectivity, low s) - Wide

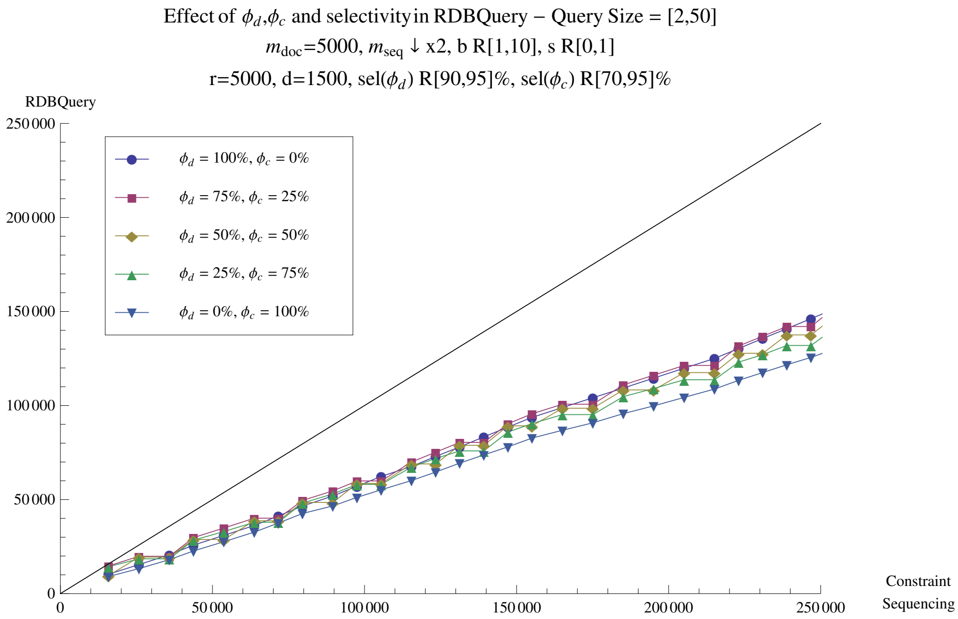


Figure H.27: RDBQuery, sel(ϕ_d) < sel(ϕ_c) (low selectivity, b decreased) - Wide

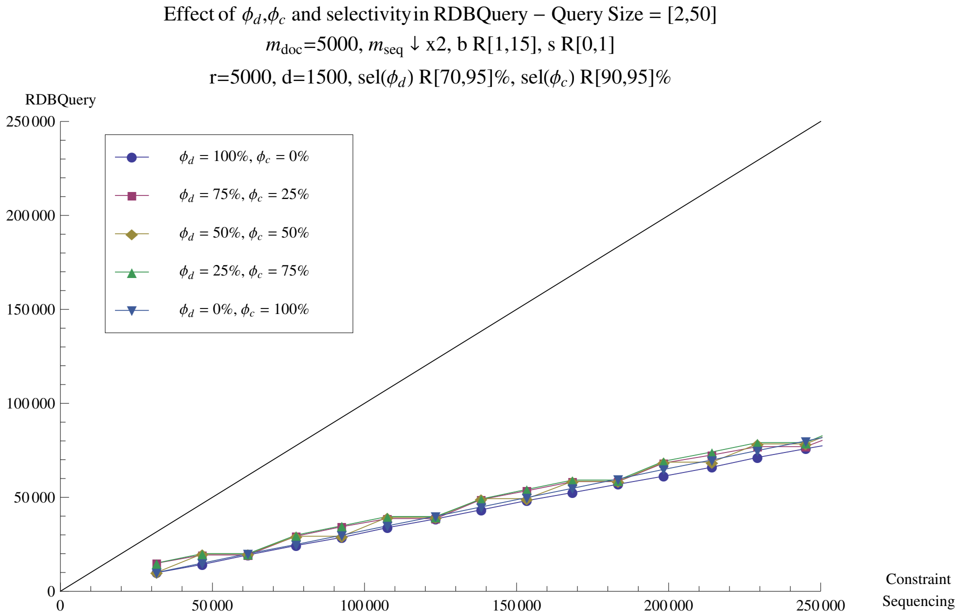


Figure H.28: RDBQuery, sel(ϕ_d) > sel(ϕ_c) (low selectivity) - Wide

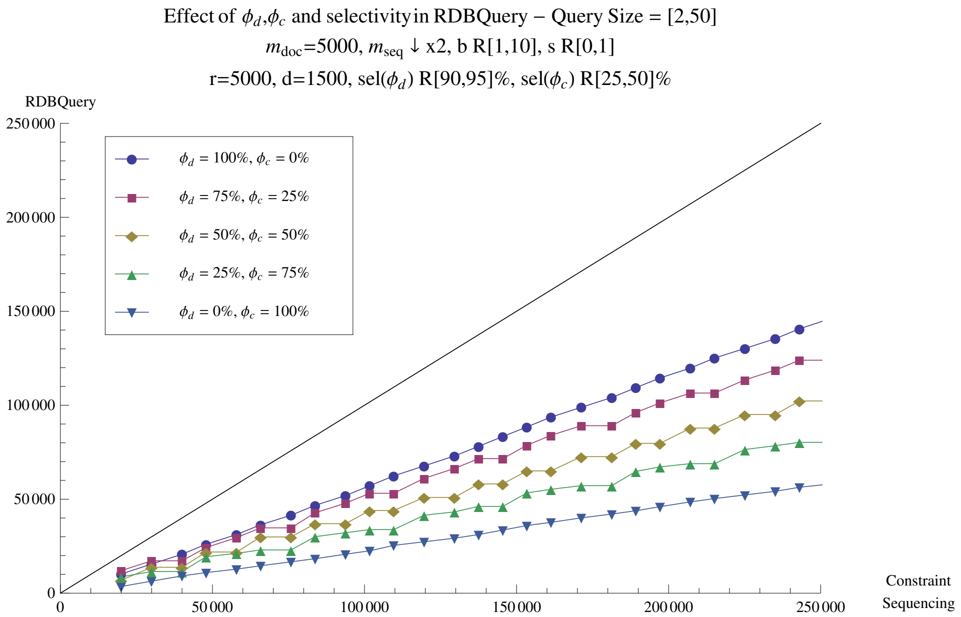


Figure H.29: RDBQuery, low sel(ϕ_d) < medium/high sel(ϕ_c) (b decreased) - Wide

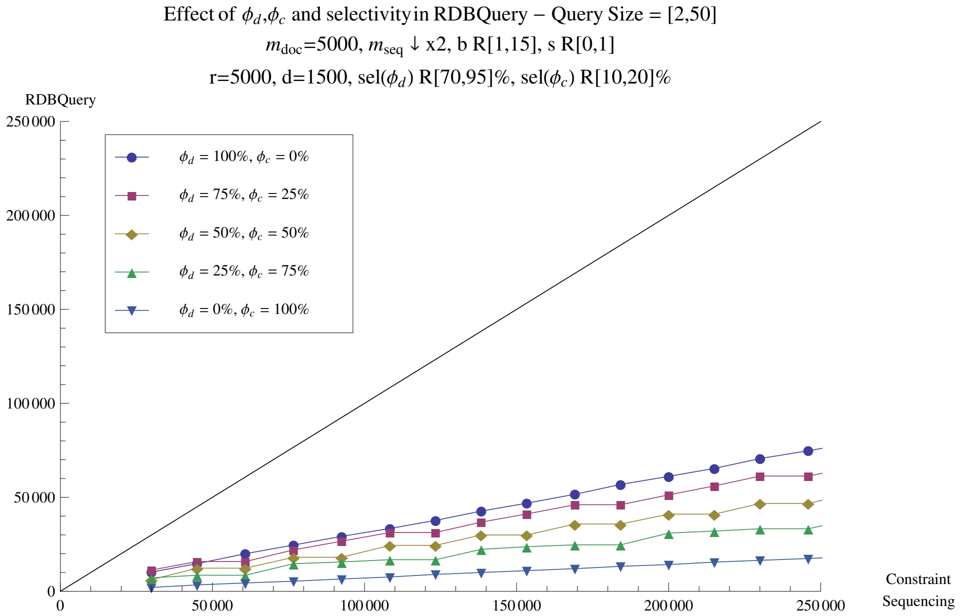


Figure H.30: RDBQuery, low sel(ϕ_d) < high sel(ϕ_c) - Wide

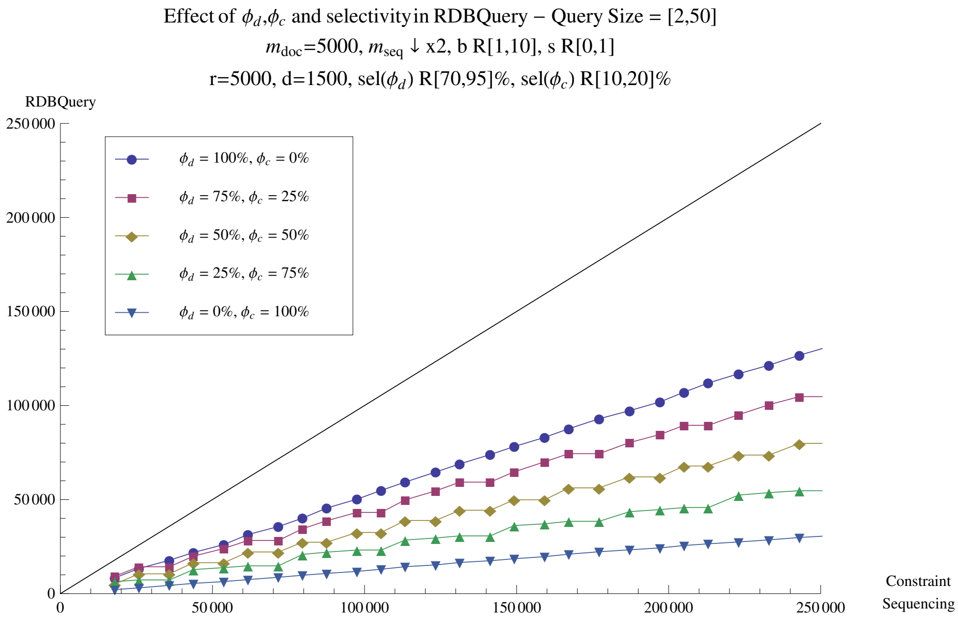


Figure H.31: RDBQuery, low sel(ϕ_d) < high sel(ϕ_c) (b decreased) - Wide

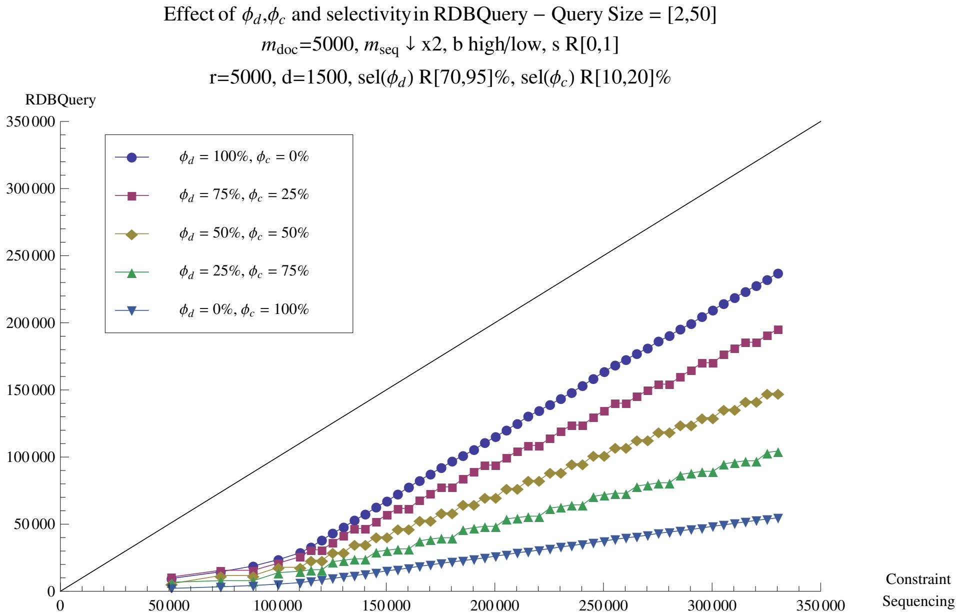


Figure H.32: RDBQuery, low sel(ϕ_d) < high sel(ϕ_c) b high/low) - Wide

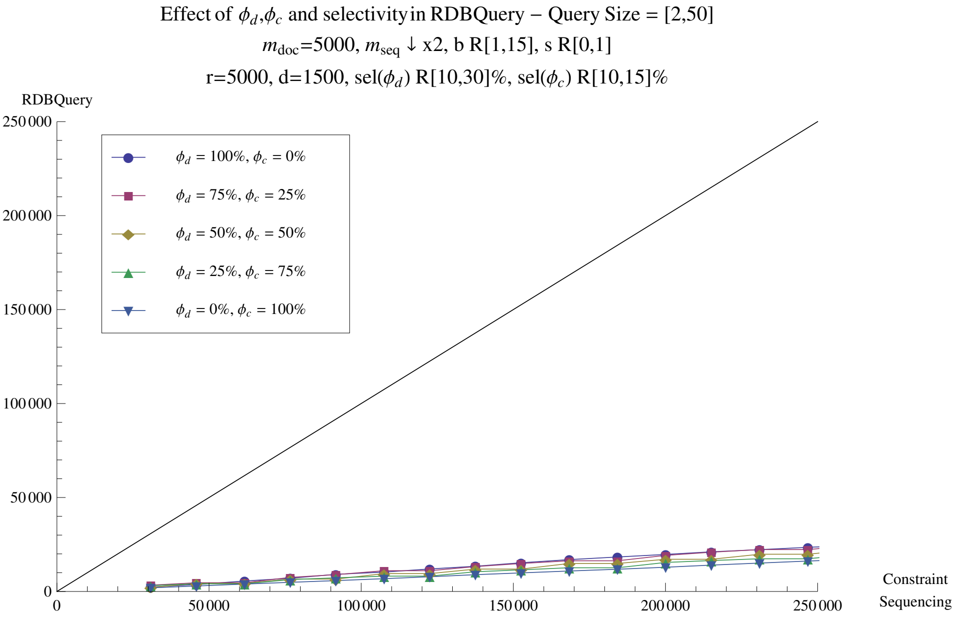


Figure H.33: RDBQuery, high sel(ϕ_d) < high sel(ϕ_c) - Wide

Appendix I

CS/RDBQuery Comparison Graphs (Similar Depth and Breadth)

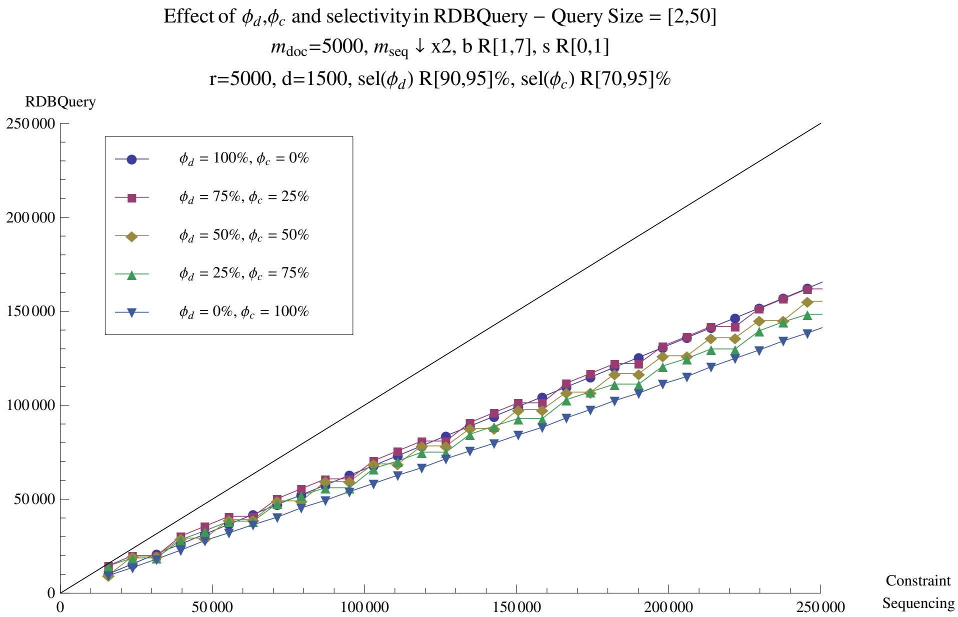


Figure I.1: RDBQuery, sel(ϕ_d < sel(ϕ_c) (low selectivity) - Similar Depth/Breadth

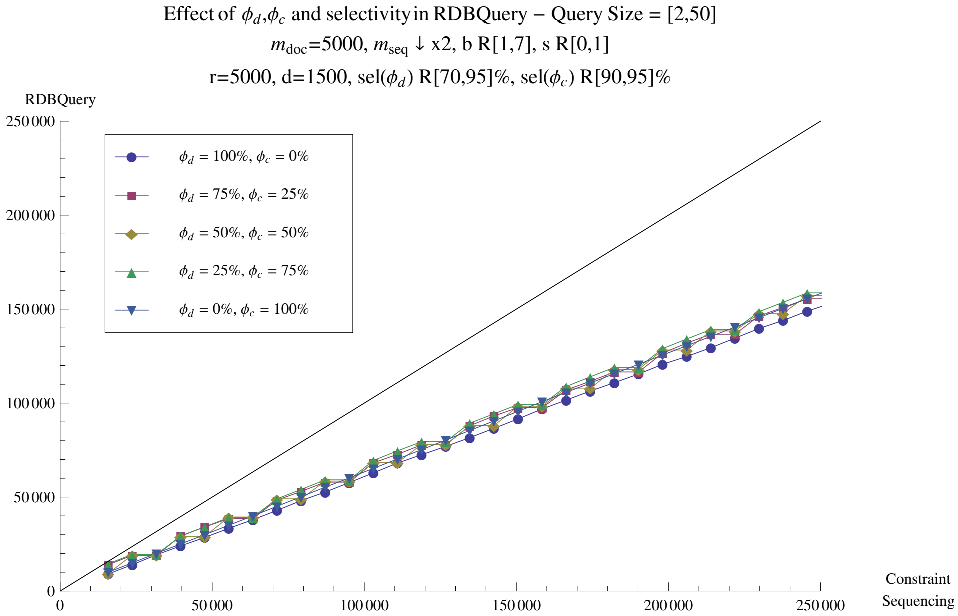


Figure I.2: RDBQuery, sel(ϕ_d > sel(ϕ_c) (low selectivity) - Similar Depth/Breadth

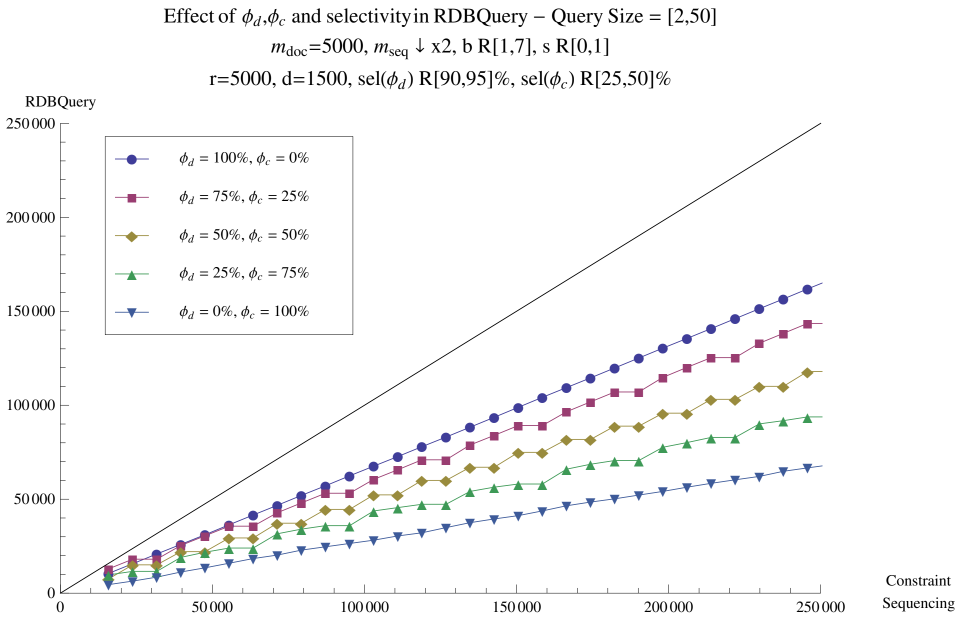


Figure I.3: RDBQuery, low sel(ϕ_d) < medium/high sel(ϕ_c) - Similar Depth/Breadth

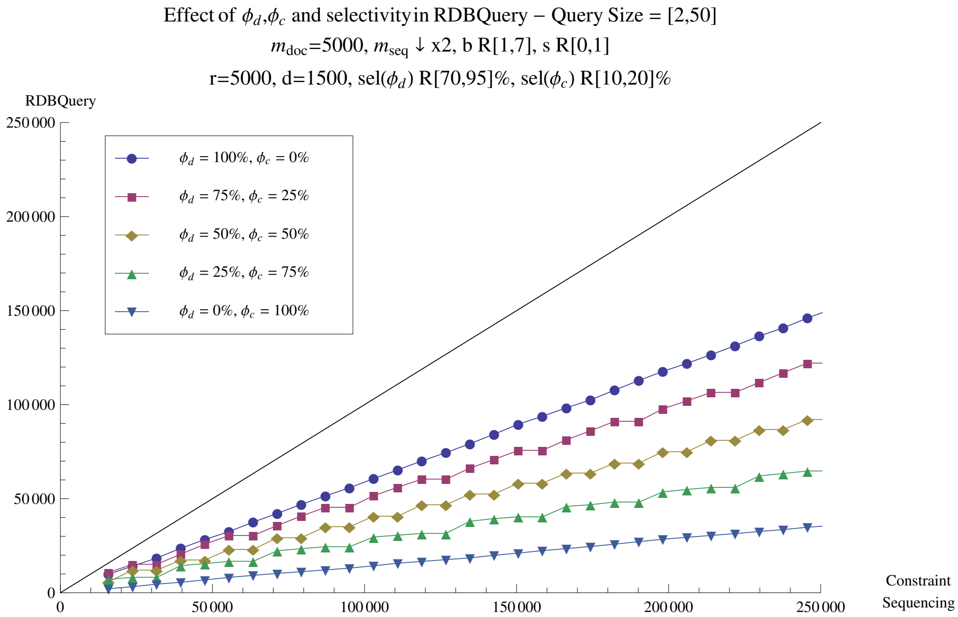


Figure I.4: RDBQuery, low sel(ϕ_d) < high sel(ϕ_c) - Similar Depth/Breadth

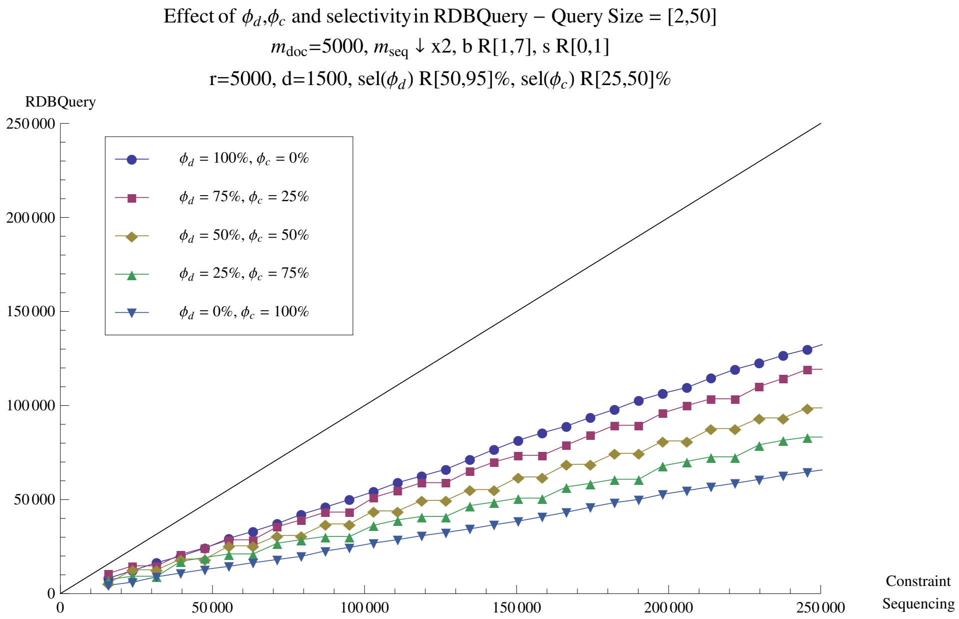


Figure I.5: RDBQuery, medium/low sel(ϕ_d) < medium/high sel(ϕ_c) - Similar Depth/Breadth

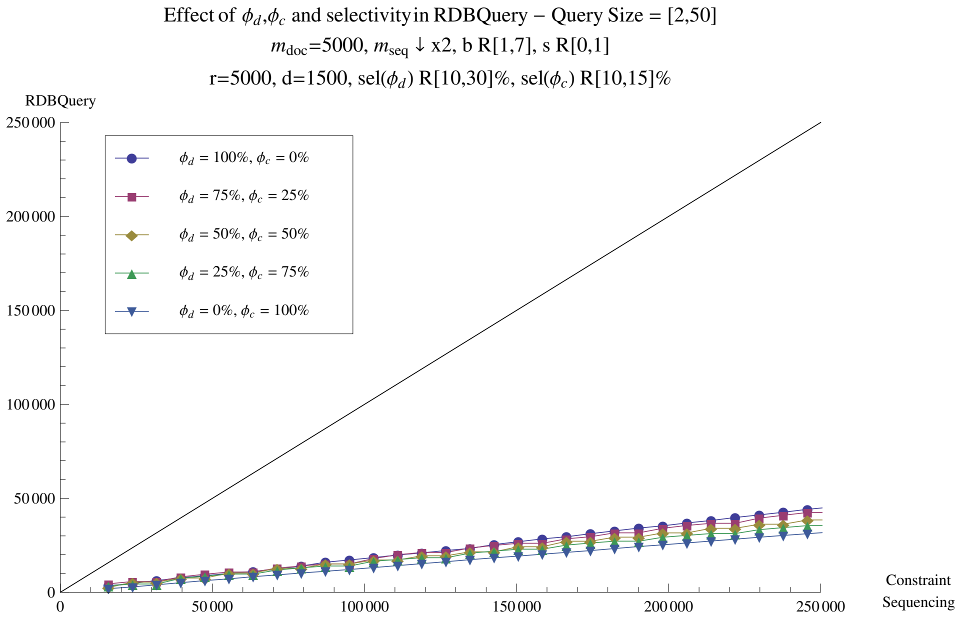


Figure I.6: RDBQuery, high sel(ϕ_d) < high sel(ϕ_c) - Similar Depth/Breadth

Appendix J

CS/RDBQuery Comparison Graphs (DBLP XML Dataset)

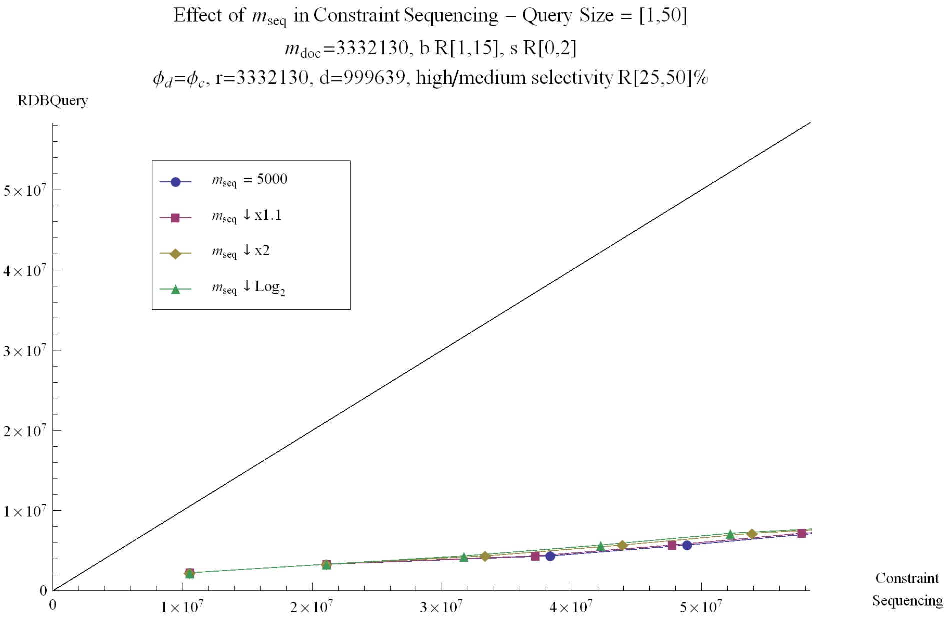


Figure J.1: CS, vary sequence size (medium/high selectivity) - DBLP

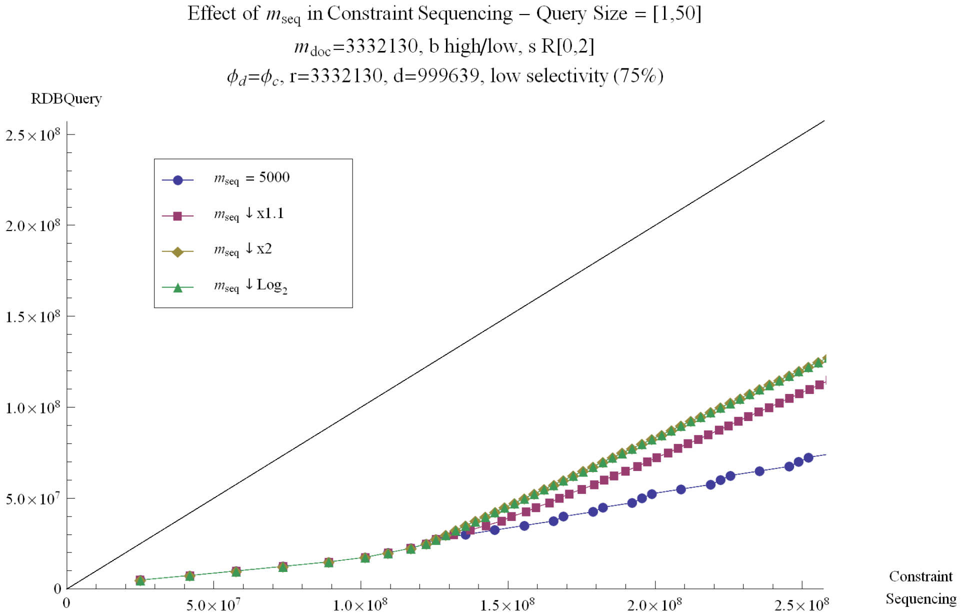


Figure J.2: CS, vary sequence size (low selectivity, *b* high/low) - DBLP

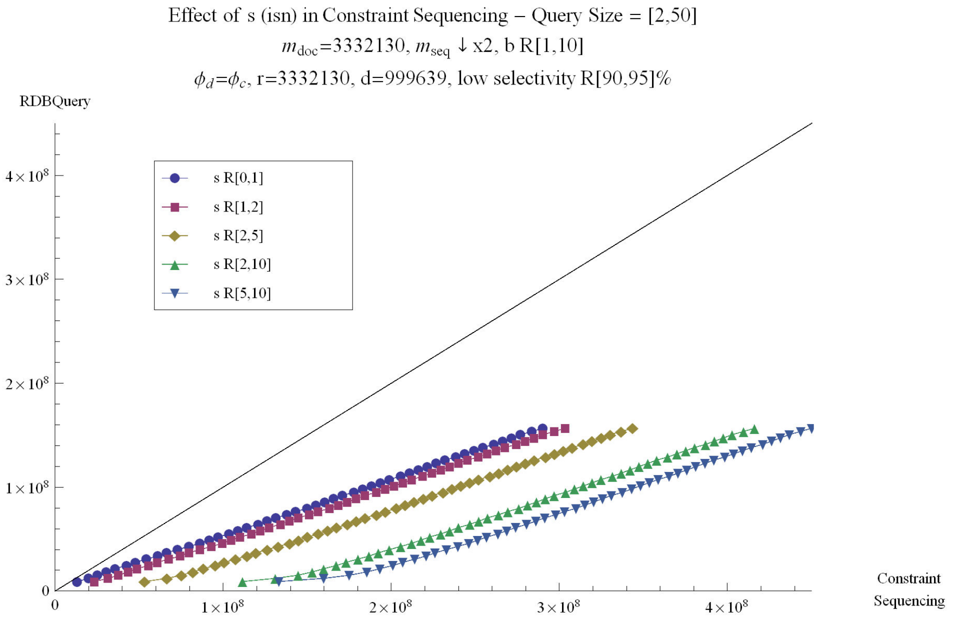


Figure J.3: CS, identical sibling nodes increasing (low selectivity) - DBLP

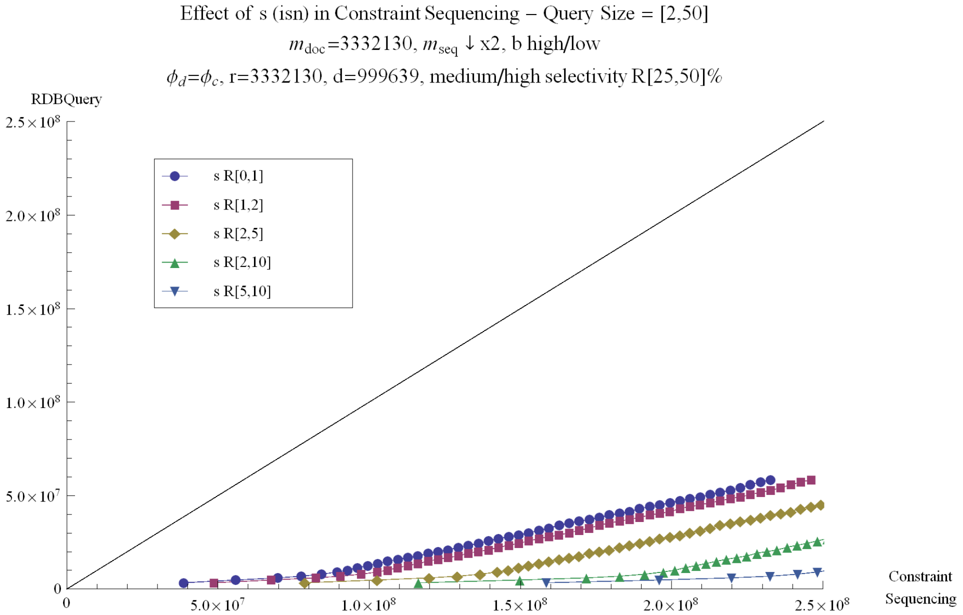


Figure J.4: CS, identical sibling nodes increasing (medium/high selectivity, b high/low) - DBLP

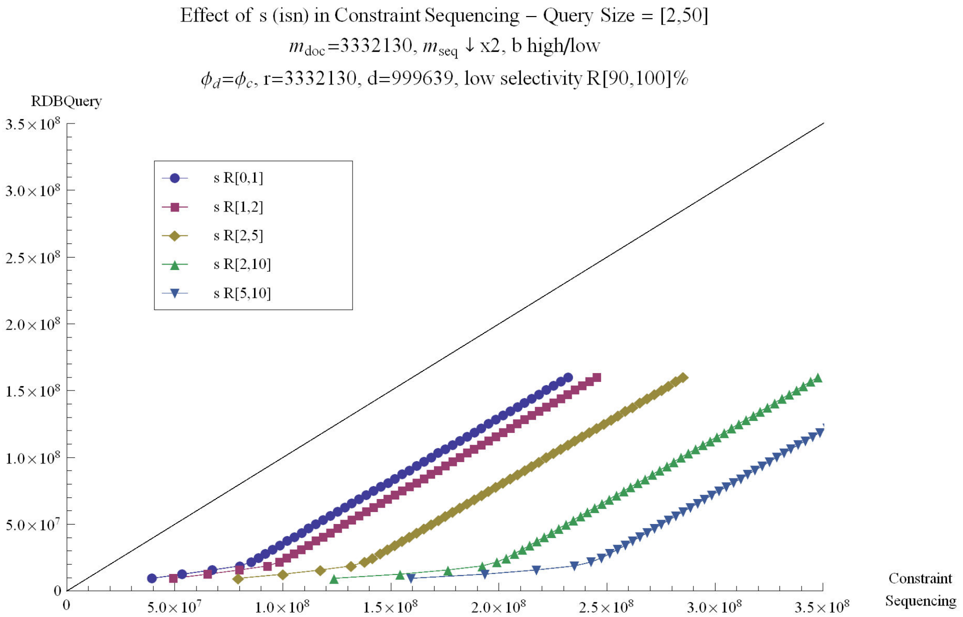


Figure J.5: CS, identical sibling nodes increasing (low selectivity, b high/low) - DBLP

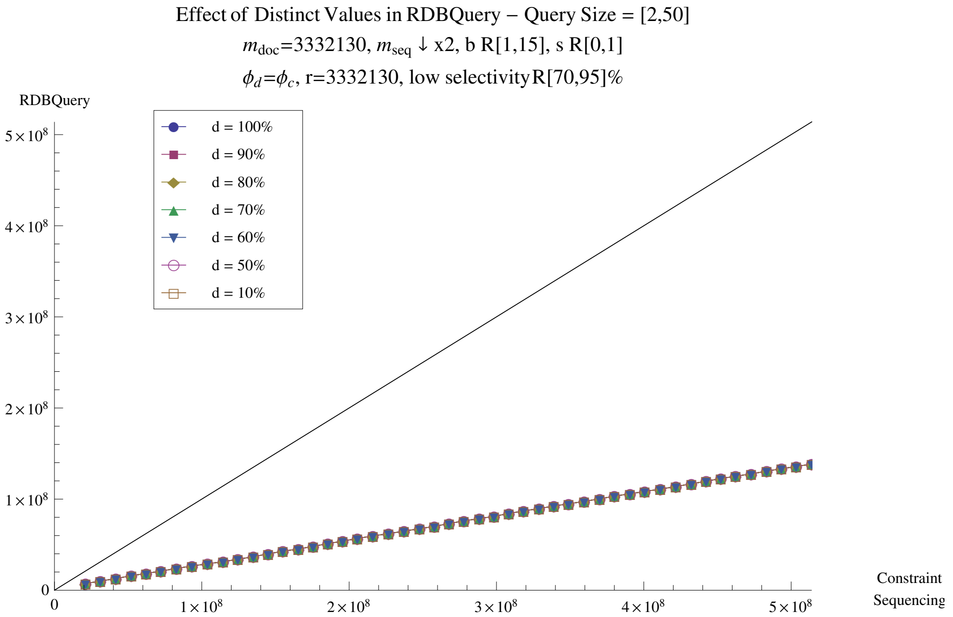


Figure J.6: RDBQuery, vary distinct values (low selectivity) - DBLP

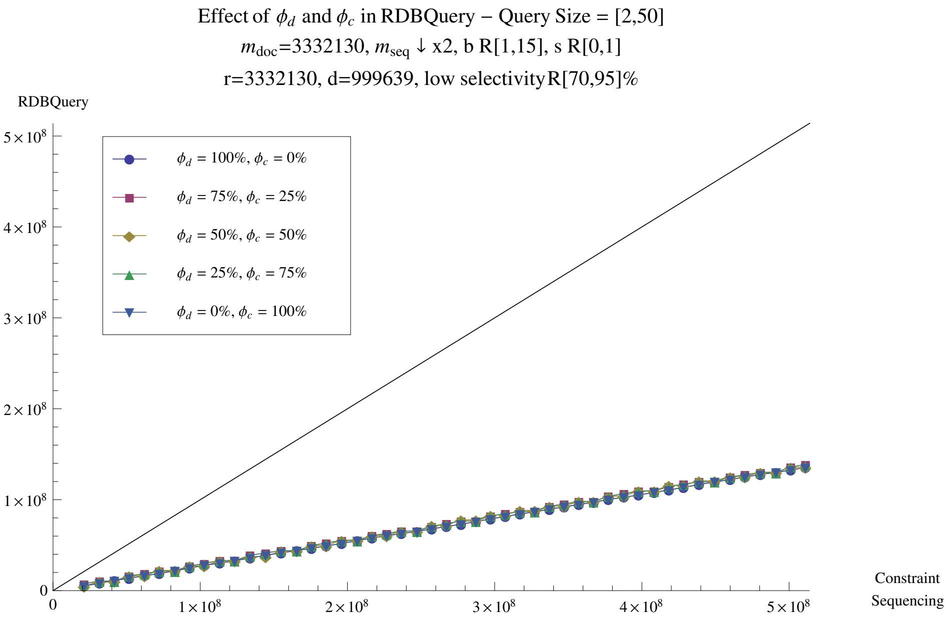


Figure J.7: RDBQuery, query edge distribution (low selectivity) - DBLP

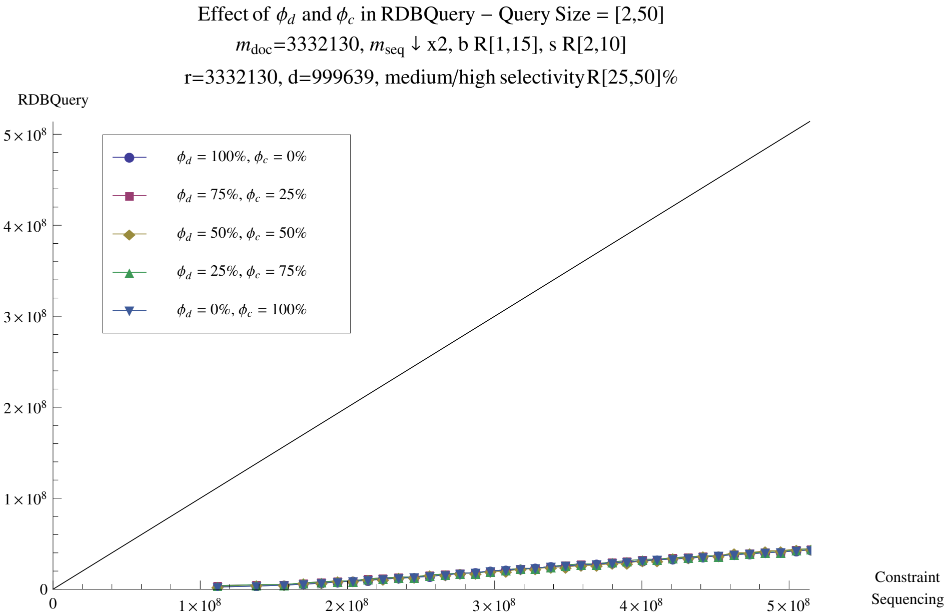


Figure J.8: RDBQuery, query edge distribution (medium/high selectivity) - DBLP

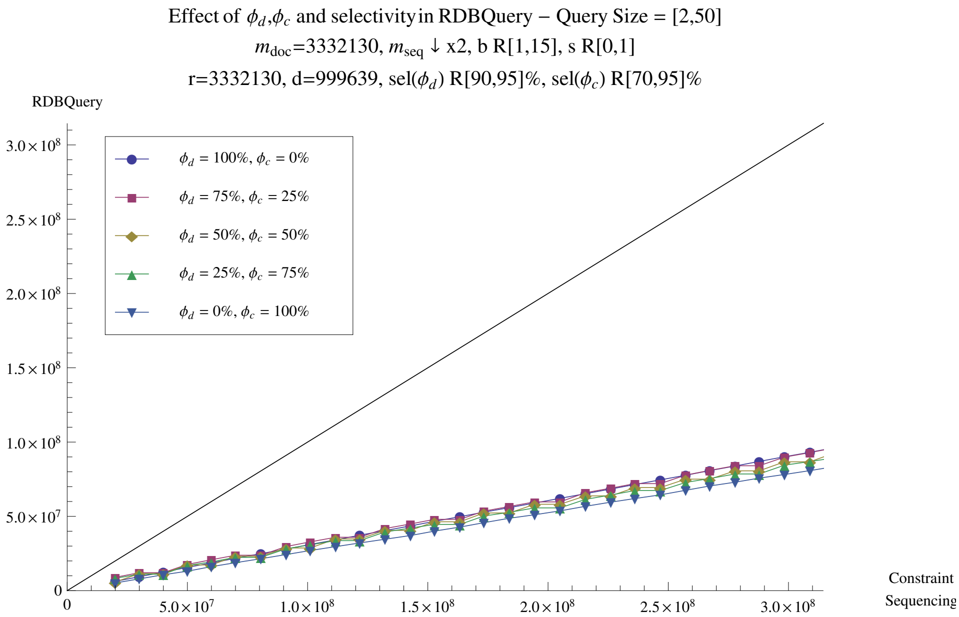


Figure J.9: RDBQuery, sel(ϕ_d) < sel(ϕ_c) (low selectivity) - DBLP

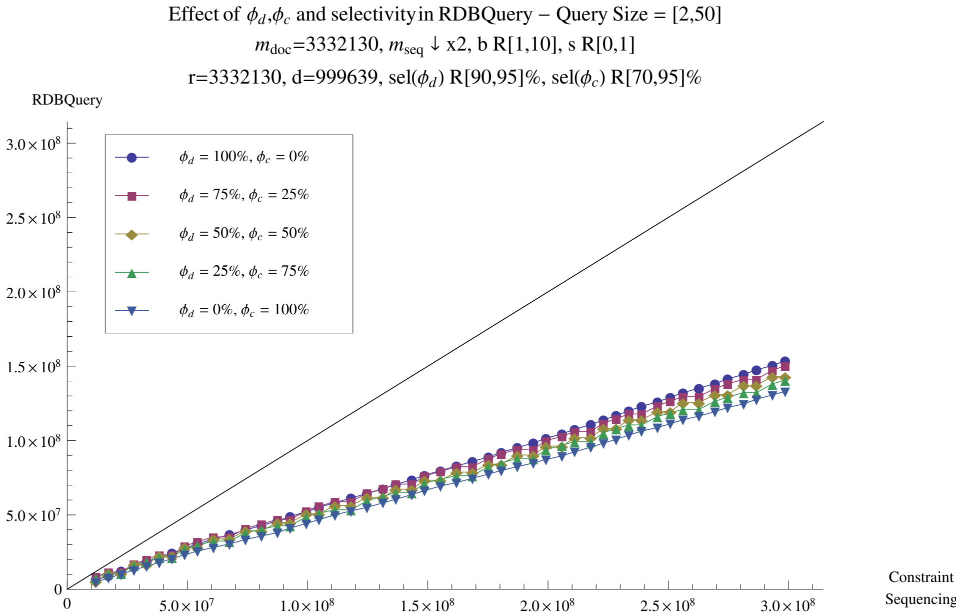


Figure J.10: RDBQuery, sel(ϕ_d) < sel(ϕ_c) (low selectivity, decreased b) - DBLP

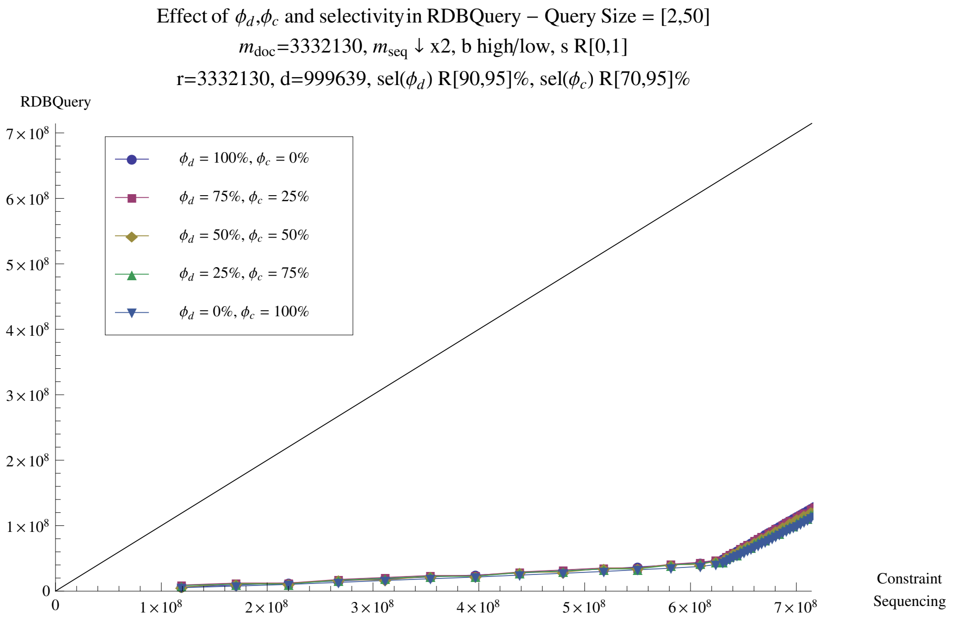


Figure J.11: RDBQuery, sel(ϕ_d) < sel(ϕ_c) (low selectivity, b high/low) - DBLP

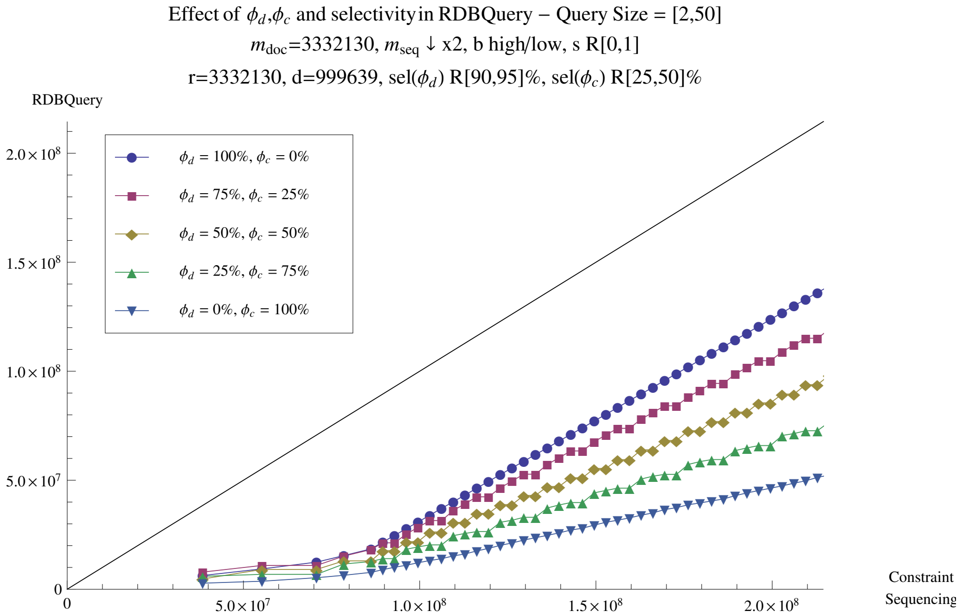


Figure J.12: RDBQuery, low sel(ϕ_d) < medium/high sel(ϕ_c) (b high/low) - DBLP

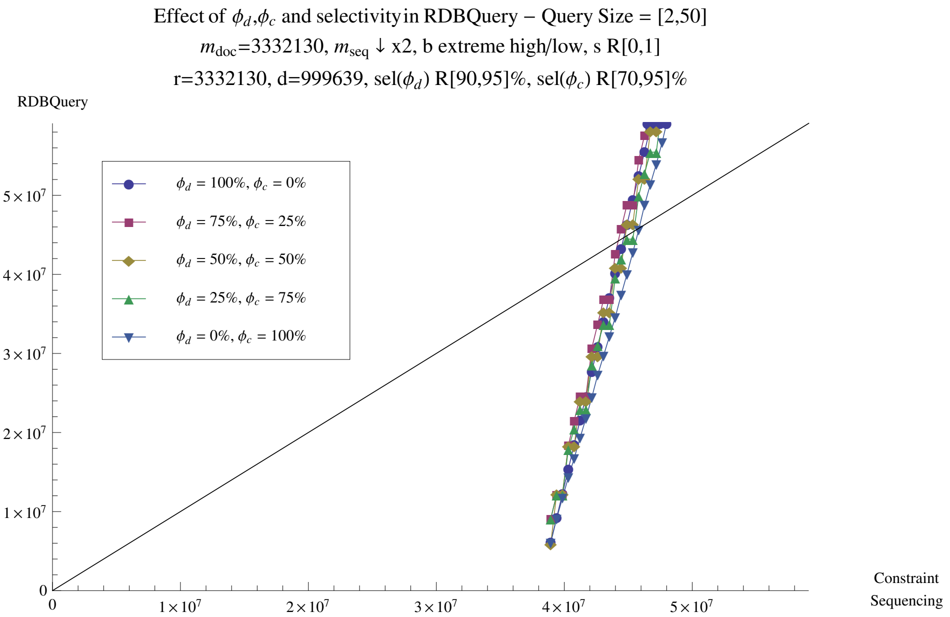


Figure J.13: RDBQuery, low sel(ϕ_d) < low sel(ϕ_c) (b extreme high/low) - DBLP

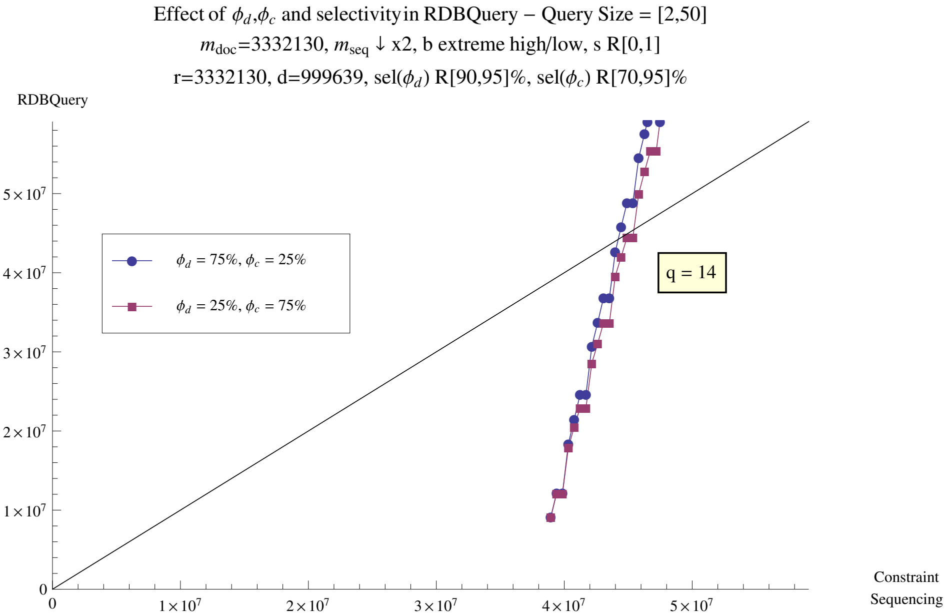


Figure J.14: RDBQuery, low sel(ϕ_d) < low sel(ϕ_c) (b extreme high/low) - DBLP (Zoom)

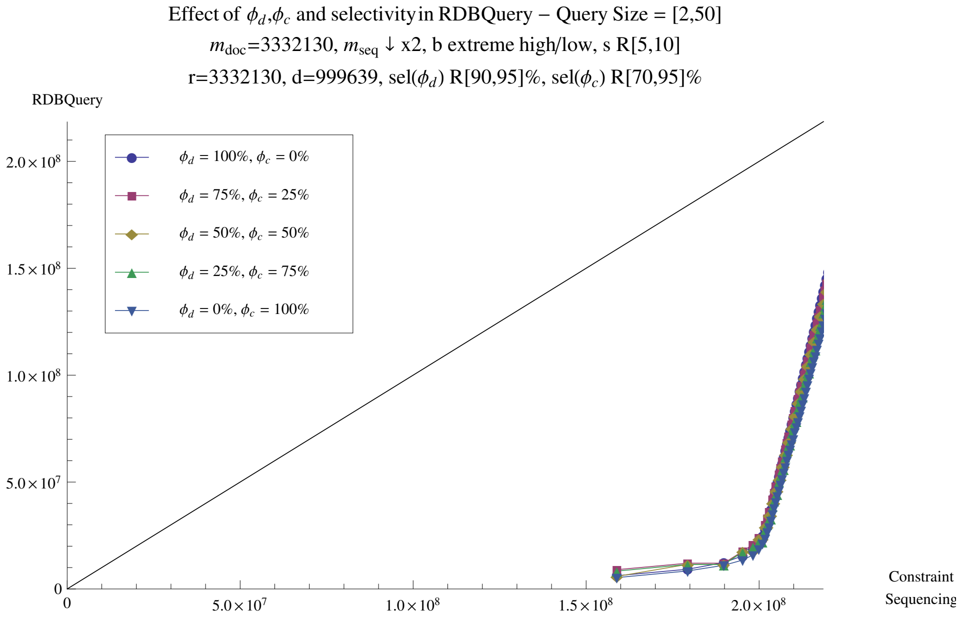


Figure J.15: RDBQuery, low sel(ϕ_d) < low sel(ϕ_c) (b extreme high/low, increased s) - DBLP