Combined Search Over Heterogeneous Repositories

Mirit Shalem
Combined Search Over Heterogeneous Repositories

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Mirit Shalem

Submitted to the Senate of the
Technion - Israel Institute of Technology

Adar 5773 Haifa February 2013
The research thesis was done under the supervision of
Asst.Prof. Yaron Kanza - Faculty of Computer Science, Technion
and
Asst.Prof. Ziv Bar-Yossef - Faculty of Electrical Engineering, Technion and Google Haifa
Engineering Center.

Acknowledgements

This dissertation would not have been possible without the help and support of many people, and I am delighted and honored to acknowledge them here.

I wish to thank, first and foremost, my two mentors, Yaron Kanza and Ziv Bar-Yossef. It has been an exceptional privilege to work with such devoted, knowledgeable, understanding, and insightful advisors. No words can express my appreciation for your outstanding guidance, advice and encouragement throughout my academic path. Yaron, I would also like to thank you for your infinite support throughout the years, and for your patience during numerous meetings in which my (third) baby was an active participant.

I owe my deepest gratitude to my family, the most devoted, supportive, and caring family. I would like to thank my parents and my wonderful husband for the unconditional support, persistent encouragement, inspirational advice, and endless love. To my three amazing children, who have dealt with my absence during many weekends with a smile, I thank you for your love and understanding. Special thanks go to my parents-in-law, Ruti and Shaul, for the warm hospitality whenever I needed a quiet place to study, and for treating me as their own daughter. To my wonderful sister Ronit, for your great advice and encouragement, I am deeply indebted to you. Finally, sincere thanks to all my dear friends, for their support and faith in me.

The generous financial help of the Technion and the Department of Computer Science is gratefully acknowledged.
List of Publications


• [69] Mirit Shalem, Yaron Kanza: How to choose combinations in a join of search results. WWW (Companion Volume) 2011: 119-120

• [68] Mirit Shalem, Yaron Kanza: Computing the top-k maximal answers in a join of ranked lists. CIKM 2010: 1381-1384

Contents

Abstract 1

1 Introduction 7
  1.1 Data Extraction From XML Repositories 8
  1.2 Maximal Join of Ranked Lists 9
  1.3 Ranking Joined Search Results 10

2 Data Extraction From XML Repositories 11
  2.1 Introduction 11
     2.1.1 Our results 11
     2.1.2 Our techniques 13
     2.1.3 Chapter Outline 14
  2.2 Related work 14
  2.3 Preliminaries 15
     2.3.1 Data model 15
     2.3.2 XPath fragment 15
     2.3.3 Evaluation model 16
     2.3.4 Modes of evaluation 17
  2.4 Lower bounds 17
     2.4.1 Techniques 17
     2.4.2 Filtering 19
     2.4.3 Full-fledged evaluation 22
     2.4.4 Pattern matching 25
  2.5 The TMC model 29
     2.5.1 The TMC model 29
     2.5.2 Properties of the TMC model 31
     2.5.3 Lower bound for Reverse set disjointness 32
     2.5.4 Lower bound for delayed intersection 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.5</td>
<td>Lower bound for tensor product</td>
<td>35</td>
</tr>
<tr>
<td>2.6</td>
<td>Algorithms</td>
<td>37</td>
</tr>
<tr>
<td>2.6.1</td>
<td>The filtering algorithm</td>
<td>37</td>
</tr>
<tr>
<td>2.6.2</td>
<td>The full-fledged evaluation algorithm</td>
<td>43</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Pattern matching upper bound</td>
<td>46</td>
</tr>
<tr>
<td>2.A</td>
<td>Filtering lower bound</td>
<td>47</td>
</tr>
<tr>
<td>3</td>
<td>Maximal Join of Ranked Lists</td>
<td>51</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>51</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Chapter Outline</td>
<td>53</td>
</tr>
<tr>
<td>3.2</td>
<td>Framework</td>
<td>53</td>
</tr>
<tr>
<td>3.3</td>
<td>Algorithms</td>
<td>55</td>
</tr>
<tr>
<td>3.3.1</td>
<td>The Vertical Nested Loop Algorithm</td>
<td>57</td>
</tr>
<tr>
<td>3.3.2</td>
<td>The Horizontal Nested Loop Algorithm</td>
<td>59</td>
</tr>
<tr>
<td>3.3.3</td>
<td>The TMax Algorithm</td>
<td>62</td>
</tr>
<tr>
<td>3.4</td>
<td>Analysis and comparison</td>
<td>66</td>
</tr>
<tr>
<td>3.4.1</td>
<td>No Associations</td>
<td>67</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Everything is Associated</td>
<td>68</td>
</tr>
<tr>
<td>3.4.3</td>
<td>One-to-one Associations</td>
<td>68</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Conclusions from the Analysis</td>
<td>69</td>
</tr>
<tr>
<td>3.5</td>
<td>Instance Optimality and approximation</td>
<td>70</td>
</tr>
<tr>
<td>3.6</td>
<td>The Hybrid Approach (VHNL)</td>
<td>74</td>
</tr>
<tr>
<td>3.7</td>
<td>Experimental Evaluation</td>
<td>76</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Methodology</td>
<td>76</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Results</td>
<td>78</td>
</tr>
<tr>
<td>3.8</td>
<td>Related Work</td>
<td>80</td>
</tr>
<tr>
<td>4</td>
<td>Ranking Joined Search Results</td>
<td>83</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>83</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Chapter Outline</td>
<td>85</td>
</tr>
<tr>
<td>4.2</td>
<td>Framework</td>
<td>85</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Search Queries and Joined Search Results</td>
<td>86</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Coverage and Optimality Ratio</td>
<td>87</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Existing Semantics</td>
<td>89</td>
</tr>
<tr>
<td>4.2.4</td>
<td>New Semantics</td>
<td>90</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Relationships Between the Semantics</td>
<td>92</td>
</tr>
</tbody>
</table>
4.2.6 Imprecise Ranking and $h$-Optimality. ................................. 93
4.3 Algorithms ................................................................. 93
  4.3.1 Producing a pipeline of $\text{Join}(Q)$ ................................. 94
  4.3.2 The FilterSorted ($\text{FS}$) Algorithm ................................. 94
  4.3.3 The FilterUnSort$\text{ed}$ ($\text{FUS}$) Algorithm .......................... 95
  4.3.4 The Top-$h$-Per-Item ($\text{TopHI}$) Algorithm ......................... 96
4.4 Experimental Evaluation .................................................. 98
  4.4.1 Methodology ......................................................... 98
  4.4.2 Experiments and their results ...................................... 99
4.5 Related Work .............................................................. 107

5 Conclusions ................................................................. 109

References ................................................................. 111
List of Figures

2.1 Example XML document (right) and twig query (left) for the XPath query: //Book>Title=“Web” and /Year=2003]. ............................................. 16
2.2 The document $D(x,y)$. ......................................................... 21
2.3 The document $G(s,t,u,v)$. .................................................. 23
2.4 The document $E(x,y)$. ....................................................... 26
2.5 The document $F(x,y)$. ....................................................... 28
2.6 A 2-dimensional mesh, with $m_1 = m_2 = n$, in the TMC model. ............. 30
2.7 An example XML document (right) and query (left). ................................ 38
2.8 A document demonstrating the sub-optimality of TwigStack and TwigStackList on the query /a/b and /c]. ................................................................. 46
2.9 A schematic illustration of a basic twig query that contains a unique “/a/b”. ....... 48
2.10 A schematic illustration of the document constructed by the reduction from RDISJn to FILTERQ. ................................................................. 49
3.1 Example of three lists of two items each. Connections between items denote associations. ........................................................................... 56
3.2 Effect of association degree ($m = 1000$, $t = 4$, $k = 100$): running time. ........ 75
3.3 Effect of association degree ($m = 1000$, $t = 4$, $k = 100$): access cost. .......... 76
3.4 Effect of association degree ($m = 1000$, $t = 4$, $k = 100$): join cost. ............ 77
3.5 Join cost of VHNL as a function of $h$ ($m = 1000$, $t = 4$, $k = 20$, $p = 5\%$) ........ 78
3.6 Performance as a function of $k$ ($m = 1000$, $t = 4$, $p = 2\%$) ...................... 79
3.7 Performance as a function of $k$ over Yahoo! Local data .......................... 80
4.1 An illustration of relationships between researchers. The shapes represent two researchers who are experts in Search ($S_e$), six researchers who are experts in Harmonic Analysis ($H_e$) and six researchers who are experts in Machine Learning ($M_e$). We represent relationships as proximity, i.e., we assume here that researchers are related if and only if they are in the same cluster. .............................................. 86
4.2 Results of the search queries $q_1$, $q_2$ and $q_3$ ................. 86
4.3 The coverage, optimality ratio and the harmonic mean of top-k results under various semantics over Yahoo! Local data, as a function of $k$ ................. 101
4.4 Measures of the top-50 results under various semantics over Yahoo! Local data, as a function of $h$ ................. 102
4.5 Running times for computing \textit{ALL} as a function of the association degree ($m = 100$, $t = 4$) ................. 103
4.6 Running times for computing \textit{ALL} as a function of the association degree ($m = 1000$, $t = 4$) ................. 103
4.7 Running times for computing \textit{EXISTS} as a function of the list length ($m = 1000$, $t = 4$, \(p = 10\%\)) ................. 104
4.8 Running times for computing \textit{ALL}, as a function of $h$ ($m = 1000$, $t = 4$, $p = 30\%$) ................. 104
4.9 User satisfaction measures (item satisfaction assigned by score) ................. 106
4.10 User satisfaction measures (random satisfying hotels) ................. 106
List of Tables

2.1 Summary of Our Results ................................................. 12

3.1 Summary of the complexity analysis ................................. 69

4.1 Parameters of result sets over Yahoo! Local data ................. 100
Abstract

In complex search tasks that utilize information from several data sources, it is often required to pose several basic search queries, join the answers to these queries, where each answer is given as a ranked list of items, and return a ranked list of combinations.

Example: A tourist Alice wants to find a festive event that is related to classical music, a good Italian restaurant and an underground station that contains an elevator, such that all three will be located within a short walking distance. A combined search should return relevant triples (event, restaurant, station), ordered according to the combined scores of the items.

Evaluation of such complex queries consists of several parts. The first part requires extracting, for each basic query independently, the relevant information from a suitable repository. Next, the results of the basic queries should be integrated, i.e., joined according to the join conditions of the complex query. Finally, the join result should be filtered and ranked, to provide the user with a manageable sized answer that has a high probability of satisfying her.

This thesis studies several aspects of complex query evaluation. We begin with the question how to extract data efficiently from XML documents. Current algorithms incur high memory costs on queries that involve child-axis nodes. In this work we provide an analytical explanation for this phenomenon. We present a large-scale study of the space complexity of evaluating twig queries (a fragment of XPath) over indexed XML documents.

The next part of the complex search is to join the ranked results of the basic queries, and return the "best" results. We discuss two problems of the top-k join of ranked lists. First, a join is a lossy operation, and over heterogeneous data sources some highly-ranked items may not appear in any combination, although they may be relevant to the user. We introduce top-k maximal join, which solves this problem by allowing maximal (incomplete) combinations in the answer, i.e., combinations that cannot be extended. We present novel algorithms for computing the top-k maximal combinations. One of the algorithms is instance optimal w.r.t. algorithms that compute a $\theta$-approximate answer. A second algorithm is much more efficient than adaptations of existing algorithms for top-k maximal join.

Second, the top-k join result may include too many repetitions of items, and repetitions may reduce user satisfaction. The main task is to choose a subset of the join result that would maximize the proba-
bility to satisfy the user. To that end, we propose two measures for estimating the quality of result sets, namely, coverage and optimality-ratio. We present new semantics for complex search queries, aiming at providing high coverage and high optimality ratio. We show empirically that our semantics outperform top-k and other existing semantics, w.r.t. user satisfaction.
## List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q$</td>
<td>Query</td>
</tr>
<tr>
<td>$D$</td>
<td>XML document [size]</td>
</tr>
<tr>
<td>$d$</td>
<td>XML document depth</td>
</tr>
<tr>
<td>$\tilde{O}$</td>
<td>Upper bound notation that suppresses factors that are linear in the query size or logarithmic in the document size</td>
</tr>
<tr>
<td>$T_a$</td>
<td>The index stream of a document label ’a’</td>
</tr>
<tr>
<td>$\phi$</td>
<td>A mapping from the nodes of a query to elements in an XML document</td>
</tr>
<tr>
<td>$\text{FILTER}_Q(D)$</td>
<td>The filtering of document $D$ using query $Q$</td>
</tr>
<tr>
<td>$\text{PM}_Q(D)$</td>
<td>The pattern matching of $Q$ in document $D$</td>
</tr>
<tr>
<td>$\text{FFE}_Q(D)$</td>
<td>The full-fledged evaluation of $Q$ on document $D$</td>
</tr>
<tr>
<td>$\text{MDSS}(f)$</td>
<td>The MDS space complexity of function $f$</td>
</tr>
<tr>
<td>$\text{DSS}(f)$</td>
<td>The (single) data stream space complexity of function $f$</td>
</tr>
<tr>
<td>$v^n$</td>
<td>An $n$-dimensional vector obtained by taking $n$ copies of bit $v$</td>
</tr>
<tr>
<td>$s \circ t$</td>
<td>Concatenation of vectors $s$ and $t$</td>
</tr>
<tr>
<td>$x \otimes y$</td>
<td>The tensor product of vectors $x$ and $y$</td>
</tr>
<tr>
<td>$o_p^l$</td>
<td>The $p$-th item in the $l$-th list</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Approximation ratio</td>
</tr>
<tr>
<td>$L_i$</td>
<td>A ranked list of items, which is the answer to a subquery $q_i$</td>
</tr>
<tr>
<td>$t$</td>
<td>Number of subqueries [lists] in $Q$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>Null, represents a missing item</td>
</tr>
</tbody>
</table>
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEL</td>
<td>Begin, End, Level (encoding)</td>
</tr>
<tr>
<td>DOE</td>
<td>Distinct Output Elements</td>
</tr>
<tr>
<td>MDS</td>
<td>Multiple Data Stream</td>
</tr>
<tr>
<td>TMC</td>
<td>Token-based Mesh Communication</td>
</tr>
<tr>
<td>RDISJ</td>
<td>Reverse set Disjointness</td>
</tr>
<tr>
<td>DINT</td>
<td>Delayed Intersection</td>
</tr>
<tr>
<td>HNL</td>
<td>Horizontal Nested Loop</td>
</tr>
<tr>
<td>VNL</td>
<td>Vertical Nested Loop</td>
</tr>
<tr>
<td>PBRJ</td>
<td>Pull Bound Rank Join</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Search is a fundamental service provided by numerous applications, on the World-Wide Web and in stand-alone information systems. An answer to a search query is a ranked list of items that are relevant to the query. Usually, a basic search application finds relevant items by posing a single query over a single data source. Yet, some complex search tasks require combining the answers of several basic search queries, by joining the ranked lists [16].

Example 1.1 shows that combining search results can be useful when using a Web application to search over geospatial data.

Example 1.1. A businessperson, who plans a visit to some city, needs to find a hotel, where she could stay, and two restaurants near the hotel, where she could have lunch and supper. Her travel expenses should be restricted to her budget, i.e., the sum of (1) the hotel’s daily rate, (2) the cost of a business lunch in one restaurant, and (3) the cost of a supper in the other restaurant should not exceed the daily budget.

The businessperson can pose three independent search queries over Web sites that contain the relevant information: one query to search for a hotel, a second query to search for a restaurant that offers business lunch and a third query to search for a restaurant that is suitable for supper. However, integrating the answers to these independent queries while taking into account their ranks can be a hard task for a human user, because in each answer the items are sorted merely according to their relevance with respect to a single query and independently from their relationships with items in the answers to the other queries.

Combining the answers of search queries can be useful in additional scenarios. One example is when searching over a social network for a group of related people, where each person has a specific attribute (e.g., searching for a manager, a software engineer and a DBA who live in the same city and are

---

1There exist Web sites that for given parameters provide a list of hotels, or a list of restaurants, with their rank and their price, e.g., www.tripadvisor.com. Some of these Web sites provide the result as a Web service, e.g., Yahoo! Travel. This allows applications to easily apply the search over several sources and integrate the results.
employed in the same company). For the technical details of how to extract data from multiple sources, using Web services, see [12, 13].

Evaluation of such complex queries consists of several parts. The first part requires extracting, for each basic query independently, the relevant information from a suitable repository. Next, the ranked results of the basic queries should be integrated, i.e., joined according to the join conditions of the complex query. Finally, the join result should be filtered and ranked, to provide the user with a manageable sized answer that has a high probability of satisfying her.

This thesis studies several aspects of complex query evaluation. In the first chapter we analyze the problem of how to extract data efficiently from XML documents. This study, however, does not consider ranking the data extracted. There are various methods to rank XML query results, e.g., see [26, 72, 73]. For the rest of this thesis, we assume the result for each basic query is a ranked list of items.

The next step of complex search is to join the ranked results of the basic queries, and return the "best" results. The second and third chapters of this thesis discuss two problems of the top-k join of ranked lists. First, a join is a lossy operation, and over heterogeneous data sources some highly-ranked items may not appear in any combination, although they may be relevant to the user. In the second chapter we introduce top-k maximal join, which solves this problem by allowing maximal (incomplete) combinations in the answer, i.e., combinations that cannot be extended. The second problem is that the top-k join result may include too many repetitions of items, and repetitions may reduce user satisfaction. The third and last part of this thesis studies the problem of how to choose a subset of the join result that would maximize the probability to satisfy the user.

In the following we introduce each of the three parts in more detail, and review the main contributions of this thesis.

### 1.1 Data Extraction From XML Repositories

XQuery and XPath [23] queries are typically represented as node-labeled twig patterns (i.e., small trees). Evaluating a twig pattern over an XML document is therefore a core database operation. As with relational databases, creating an index over the XML document at a pre-processing step can significantly reduce the costs (time, space) of query evaluation. Similarly to text search, an index for an XML document consists of posting lists or streams, one for each XML label that occurs in the document. The stream consists of positional encodings of all the elements that have this label, in document order. In this study we focus on the most popular encoding scheme, the BEL encoding [14], in which each element is encoded as a (Begin,End,Level) tuple. The BEL encoding, although being compact, enables simple testing of structural relationships between elements.
Over the past decade, many algorithms for evaluating twig queries over indexed XML documents have been proposed (e.g., [14, 21, 32, 47, 57, 60, 78]). Much progress has been made in supporting wider fragments of XPath and XQuery and in achieving better performance in terms of running time, memory usage, and I/O costs.

Many of the existing algorithms follow two trends. The first trend is the tendency to achieve good performance on queries that involve descendant-axis only nodes, while suffering from poor performance on queries that involve child-axis nodes. The second trend relates to the mode of evaluation: many current algorithms find all possible matches of the whole query in the document (“pattern matching”), even though they are required to output only the matches of the query’s output node(s) (“full-fledged evaluation”) or to simply return a bit indicating whether there is at least one match of the query in the document (“filtering”).

These trends raise two natural questions. First, is the space overhead incurred by child-axis nodes inherent or is it an artifact of the way existing algorithms work? Second, does the pattern matching evaluation mode incur any overhead relative to full-fledged evaluation and/or filtering?

In order to address the above questions, we embark on a large-scale study of the space complexity of evaluating twig queries over indexed documents. We show the space to depend on three factors: (1) whether the query is a path or a tree; (2) the types of axes occurring in the query and their occurrence pattern; and (3) the mode of query evaluation (filtering, full-fledged, or pattern matching). We prove lower bounds via a new model of communication complexity and matching upper bounds that for some queries are exceedingly more efficient than existing algorithms.

### 1.2 Maximal Join of Ranked Lists

Complex search tasks that utilize information from several data sources, or that produce groups of related items over a single source, are answered by integrating the results of distinct basic search queries. In such integration, each basic query returns a ranked list of items, and the main task is to compute the join of these lists, returning the top-k combinations. Computing the top-k join of ranked lists has been studied extensively for the case where the answer comprises merely complete combinations. However, a join is a lossy operation, and over heterogeneous data sources some highly-ranked items, from the results of the basic queries, may not appear in any combination. Yet, such items and the partial combinations in which they appear may still be relevant answers and should not be discarded categorically.

In this work we consider a join where combinations are padded by nulls for missing items, as in outer-join. A combination is maximal if it cannot be extended by replacing a null by an item. We present two novel algorithms for computing the top-k maximal combinations and depict two modifications of common top-k algorithms, which serve as benchmarks. The algorithms differ in the order by which
they produce the complete and the incomplete combinations and in the way they manage the required data structures. The first novel algorithms is instance optimal over the class of algorithms that compute a $\theta$-approximation to the answer. The second novel algorithm is more efficient than the other three algorithms (the first novel one and the two modifications), as shown by our analytic comparison and experimental evaluation.

1.3 Ranking Joined Search Results

The join result may include too many repetitions of items, and hence, frequently the entire join is too large to be useful. This can be solved by choosing a small subset of the join result. The focus of this study is on how to choose this subset. We propose two measures for estimating the quality of result sets, namely, coverage and optimality ratio. Intuitively, maximizing the coverage aims at including in the result as many as possible appearances of items in their optimal combination, and maximizing the optimality ratio means striving to have each item appearing only in its optimal combination, i.e., only in the most highly ranked combination that contains it. One of the difficulties, when choosing the subset of the join in a complex search, is that there is a conflict between maximizing the coverage and maximizing the optimality ratio.

In this study, we introduce the measures coverage and optimality ratio. We present new semantics for complex search queries, aiming at providing high coverage and high optimality ratio. We examine the quality of the results of existing and the novel semantics, according to these two measures, and we provide algorithms for answering complex search queries under the new semantics. Finally, we present an experimental study, using Yahoo! Local Search Web Services, of the efficiency and the scalability of our algorithms, showing that complex search queries can be evaluated effectively under the proposed semantics.
Chapter 2

Data Extraction From XML Repositories

In this chapter we present a study of the space complexity of evaluating twig queries (a fragment of XPath) over indexed XML documents.

2.1 Introduction

Over the past decade, many algorithms for evaluating twig queries over indexed XML documents have been proposed (e.g., [14, 21, 32, 47, 57, 60, 78]). Much progress has been made in supporting wider fragments of XPath and XQuery and in achieving better performance in terms of running time, memory usage, and I/O costs. Many of the existing algorithms follow two trends. The first trend is the tendency to achieve good performance on queries that involve descendant-axis only nodes, while suffering from poor performance on queries that involve child-axis nodes. The second trend relates to the mode of evaluation: many current algorithms find all possible matches of the whole query in the document (“pattern matching”), even though they are required to output only the matches of the query’s output node(s) (“full-fledged evaluation”) or to simply return a bit indicating whether there is at least one match of the query in the document (“filtering”).

These trends raise two natural questions. First, is the space overhead incurred by child-axis nodes inherent or is it an artifact of the way existing algorithms work? Second, does the pattern matching evaluation mode incur any overhead relative to full-fledged evaluation and/or filtering?

2.1.1 Our results

In order to address the above questions, we embark on a large-scale study of the space complexity of evaluating twig queries over indexed documents. Our lower bound results are quite strong, since they apply to the instance data complexity [10], rather than to the standard data complexity. That is, we fix any query, not just a worst-case query, and then prove lower bounds for evaluating this query. Therefore,
our lower bounds are given in terms of properties of the query as well as parameters of the document. Our analysis shows that the space complexity of twig query evaluation depends on three parameters: (i) whether the query is a path or a tree; (ii) the types of the axes in the query and their occurrence pattern; and (iii) the mode of evaluation: filtering, full-fledged evaluation, or pattern matching.

Table 2.1: Summary of Our Results

<table>
<thead>
<tr>
<th>Axis pattern</th>
<th>Path queries</th>
<th>Tree queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Upper bound</td>
<td>Lower bound</td>
</tr>
<tr>
<td>(//)<em>/(//)</em></td>
<td>$\tilde{O}(1)_{\text{2.6.1}}$</td>
<td>$\Omega(1)$</td>
</tr>
<tr>
<td>(//)<em>(///)</em></td>
<td>$O(d)$ [14]</td>
<td>$\Omega(d)_{\text{3.2.4.2}}$</td>
</tr>
</tbody>
</table>

(a) Filtering

<table>
<thead>
<tr>
<th>Axis pattern</th>
<th>Path queries</th>
<th>Tree queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Upper bound</td>
<td>Lower bound</td>
</tr>
<tr>
<td>(//)<em>(///)</em></td>
<td>$O(1)_{\text{2.6.2}}$</td>
<td>$\Omega(1)$</td>
</tr>
<tr>
<td>(//)<em>(///)</em></td>
<td>$O(d)$ [14]</td>
<td>$\Omega(d)_{\text{3.2.4.2}}$</td>
</tr>
</tbody>
</table>

(b) Full-fledged evaluation

<table>
<thead>
<tr>
<th>Axis pattern</th>
<th>Path queries</th>
<th>Tree queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Upper bound</td>
<td>Lower bound</td>
</tr>
<tr>
<td>(//)<em>(///)</em></td>
<td>$O(d)$ [14]</td>
<td>$\Omega(1)$</td>
</tr>
<tr>
<td>(//)<em>(///)</em></td>
<td>$O(d)$ [14]</td>
<td>$\Omega(d)_{\text{3.2.4.1.1}}$</td>
</tr>
<tr>
<td>(//)<em>(///)</em></td>
<td>$O(d)$ [14]</td>
<td>$\Omega(d)_{\text{3.2.4.2.2}}$</td>
</tr>
</tbody>
</table>

(c) Pattern matching

$^*$ $D$, $d$, and DOE denote the document size, its depth, and the number of Distinct Output Elements, resp.

Table 2.1 summarizes our results (marked in shaded background) as well as previously known bounds. We analyze each evaluation mode separately. We also categorize the queries according to the axis pattern of paths in the query (represented as a regular expression). To classify a query, we check if at least one path in the query fits the regular expression, starting from the lowest row of the table upwards. As the query size is typically small relative to the document size or the output size, we did not focus on it as a parameter. We use the $\tilde{O}$ notation to suppress factors that are linear in the query size or logarithmic in the document size.

Our results provide two theoretical explanations for the difficulty in handling queries with child-axis nodes. The first explanation applies to all evaluation modes and to queries that contain the (//)(/) pattern (i.e., ones that consist of at least one descendant-axis node that is followed by a child-axis node, such as //all; see the last row in all three tables)$^1$. We show that the space needed to evaluate such queries is $\Omega(d)$, where $d$ is the document’s depth. The lower bound follows from the need to simultaneously hold in memory candidate matches of the descendant-axis node that are nested within each other. Thus, when evaluating the query on highly recursive documents (ones that consist of long chains of same-label elements), $\Omega(d)$ space may be needed. The second, and possibly more significant, explanation

$^1$Our lower bounds for queries that contain the (//)(/) pattern require that the labels of the two corresponding nodes (i.e., the descendant axis node and the child axis node below it) should be different and not appear elsewhere in the query.
applies to the full-fledged evaluation and pattern matching modes and to (a subset of) the tree queries that contain the (//)(/) pattern (see the lower right corner at the second and third tables). We prove that processing such queries additionally requires $\Omega(DOE)$ space, where 'DOE' is the number of Distinct Output Elements. As the output size can be as large as the document itself, it may be unavoidable to use a lot of memory on such queries.²

Our study reveals another notable phenomenon. On tree queries that do not contain the (//)(/) pattern (i.e., ones that consist of descendant-axis nodes only or ones in which child-axis nodes always precede descendant-axis nodes; see the upper right corners in all three tables), pattern matching is subject to an $\Omega(DOE)$ lower bound, while the other modes are not. We present a new twig join algorithm that is adapted for the filtering and full-fledged evaluation modes and uses only constant space for these queries. Thus, our algorithm demonstrates that working in the pattern matching mode, while only filtering or full-fledged evaluation are needed, incurs significant space overhead.

The $\tilde{O}(d)$ upper bound for path queries (in all evaluation modes) follows from the PathStack algorithm [14]. The $\tilde{O}(d)$ upper bound for filtering tree queries follows from the TurboXPath algorithm [52]³ (see also [10,47]). The tight upper bounds in the pattern matching mode (see the second row of the third table) follow from extensions we propose to the TwigStack algorithm [14]. Obtaining space-optimal algorithms for tree queries that contain the (//)(/) pattern remains an open problem. ($\tilde{O}(D)$ is the space needed by an in-memory algorithm that simply stores the whole document in main memory.) Another open problem is the difference between the lower and upper bounds for pattern matching of path queries with the (//)?//? pattern.

2.1.2 Our techniques

Space lower bounds for data stream algorithms are normally proved via communication complexity. It turns, however, that the standard communication complexity model [54] is inadequate for proving lower bounds for multiple data stream (MDS) algorithms. We therefore introduce a new model of multi-party communication complexity—the token-based mesh communication model (TMC)—which enables proving space lower bounds for MDS algorithms. The model allows a clean abstraction of the information-theoretic arguments made in the lower bound proofs. It also enables us to recycle arguments that are repeatedly used in the proofs, thus making them more modular. We prove communication lower bounds in the TMC model for two variants of the set-disjointness problem and for the tensor product problem. Our space lower bounds for twig query evaluation are obtained via reductions from these problems.

²Note that in general the algorithm does not need to allocate expensive main memory storage for the output, since the output can be written to a write-once output device.
³TurboXPath is designed for XML streams, yet it can be made to work on indexed XML documents with a constant factor space overhead.
successors, our algorithm is “holistic”, as it treats the whole query as one unit. Yet, unlike TwigStack, our algorithm is not “document-driven”, but rather “query-driven”. That is, rather than traversing the elements in document order and at each step looking for the largest query subtree that is matched by the current document subtree, our algorithm traverses the query top-down and advances the stream cursors to the next match. We include detailed theoretical correctness and performance analysis of our algorithm. As the main thrust of this chapter is the analytical study of the space complexity of processing twig queries, we do not include empirical analysis of our algorithm.

2.1.3 Chapter Outline

We begin by presenting background information and a formal definition of the evaluation model. In Section 2.4 we prove our lower bounds, and describe the generic reduction scheme we use in our proofs. Section 2.5 presents the TMC model and communication lower bounds for three problems. These lower bounds are used in Section 2.4. In Section 2.6 we discuss existing and new upper bounds, and in particular, we present our twig join algorithm with its full analysis.

2.2 Related work

Starting with the seminal work of Bruno et al. [14] on holistic twig join algorithms, there have been many follow-up studies that presented improvements in the I/O and memory costs (e.g., [32, 48, 57, 60]) or extended the supported fragment of queries (e.g., [47, 78]). However, none of these papers presents a systematic study of lower bounds as we do.

The only previous work to address space lower bounds for processing twig queries was a paper by Choi et al. [22]. They state that any algorithm evaluating the query //a[/a and /a] requires super-constant memory. Our study does not address a single worst-case query, but provides lower bounds for evaluation of any query. Our lower bounds are also finer-grained and yield a quantitative characterization of the space complexity.

Chen et al. [21] compared three different indexing schemes: by label, by label and level, and by ancestors’ labels. They demonstrate the impact of the chosen scheme on the classes of twig patterns that can be evaluated “optimally”, i.e., without redundant intermediate results. However, their focus is on the two latter schemes, and not on lower bounds for the first scheme, which is the subject of study in this thesis.

Several previous works proved space lower bound for evaluating XPath queries in other models. Gottlob, Koch, and Pichler [37], Segoufin [66], and Götz, Koch and Martens [38] studied the complexity of evaluating XPath queries over XML documents stored in main memory. Grohe, Koch, and

---

4While this statement is true, we suspect the proof included in [22] to be flawed, as it relies on a reduction to, and not from, evaluation of Select-Project-Join queries over continuous data streams [7].
Schweikardt [41] proved lower bounds for XPath evaluation on external memory machines with limited random accesses. As the models studied in these works are completely different from the model studied in this thesis, their lower bounds are not applicable to our setting. Bar-Yossef, Fontoura, and Josifovski [9, 10] showed space lower bounds for evaluating XPath queries over a single XML stream. Lower bounds in our model derive the same lower bounds in their model, while upper bounds in their model also apply in our model.

There is extensive literature on massive data sets computations in general, and on multiple data streams in particular. Various models have been presented and analyzed, e.g. [2, 7, 8, 35, 40–42]. All these models are different from the multiple data stream model studied in this thesis, either because they are stronger (and thus admit weaker lower bounds) or because they focus on other complexity measures than space.

The multiple-cursor multiple data stream model was analyzed in [39] and a lower bound for reverse set-disjointness was provided. Yet, the paper focuses on relational algebra queries and not on XPath.

2.3 Preliminaries

2.3.1 Data model

XML documents are modeled as ordered rooted trees. Each node in the tree is called an element and is labeled by a name or a text value. The edges represent direct element-subelement or element-value relationships. Every document has an (invisible) root whose label we denote by “$”. Figure 2.1 depicts an example document tree.

Similarly to previous papers on twig joins, we assume only leaf elements in the document may contain text. This makes the relationship element-value easier to represent and evaluate.

2.3.2 XPath fragment

We focus on a fragment of XPath, which we call basic twig queries. Many existing algorithms focus on this type of queries [14, 48, 57]. The syntax of a basic twig query is defined as follows:

Twig ::= Step | Step Twig
Step ::= Node [Predicate]?  
Path ::= Node | Node Path
Node ::= Axis label
Axis ::= /|//
Predicate ::= Twig | Path = textvalue | Predicate and Predicate

A basic twig query can be represented as a tree, where each internal node is marked by a label and each leaf is marked by a label or by a text value. Every edge in the tree represents the axis of the node.
Figure 2.1: Example XML document (right) and twig query (left) for the XPath query: //Book [Title="Web" and Year=2003].

below it, and can be either a child axis (/) or a descendant axis (//). Similarly to documents, every query has an invisible root labeled by “$”. One of the tree’s nodes is designated as the output node. Figure 2.1 depicts an example basic twig query. The output node is pointed by an arrow.

2.3.3 Evaluation model

We consider query evaluation over indexed XML documents. An XML document is represented in positional encoding. Each document element is encoded as a triple: (Begin, End, Level), based on its position in the document. “Begin” and “End” are the positions of the beginning and the end of the element, respectively, and “Level” is the nesting depth. Positional encoding is the most popular format for representing XML documents, since it is simple and compact, yet it allows for efficient evaluation of structural relationships between document nodes.

An indexed XML document consists of a collection of index streams, one stream for every label that occurs in the document. For every label ‘a’, stream T_a contains positional encodings of all elements with label ‘a’ in the document, sorted by the “Begin” attribute. Each stream ends with a special “EndOfStream” tuple with dummy values. Each query node u is associated with a cursor in the corresponding stream T_u. An algorithm can read from a cursor position many times, until it decides to advance it. Cursors can be advanced only forwards, and not backwards. The output is written to a write-only stream. If two query nodes u, v share the same label, the algorithm maintains two separate cursors.

The output node in an XPath query is always the path’s leaf. Yet, in the tree representation, this leaf may become any labeled node in the tree. For example, the two queries //a/b and //a[b]c are represented by the same tree, but their output nodes are different.

Since data values may be located only within leaf elements, they can be attached to the BEL encoding of the element which contains them. This makes the relationship element-value easy to evaluate.
on streams $T_u$ and $T_v$, which represent the same stream. We therefore abuse notation and use $T_u$ to
denote the *cursor* on the stream corresponding to $u$. The algorithms we consider are restricted to access
only streams corresponding to labels that occur in the query. All known twig join algorithms conform
to this restriction.

When analyzing the space complexity of an algorithm that runs over an indexed XML document, we
do not take into account the space used for storing the input streams, the cursors, or the output stream.

### 2.3.4 Modes of evaluation

We consider three modes of query evaluation: *filtering*, *full-fledged*, and *pattern matching*. The underlying notion in all modes is a *match*.

For a query $Q$ and a node $u \in Q$, we denote by $Q_u$ the sub-query rooted at $u$. Similarly, for a
document $D$ and an element $e \in D$, we denote by $D_e$ the sub-document rooted at $e$.

**Definition 2.1** (Sub-query match). A match of a sub-query $Q_u$ in a sub-document $D_e$ is a mapping $\phi$ from the nodes of $Q_u$ to elements in $D_e$ satisfying the following: (1) *root match*: $\phi(u) = e_u$, (2) *labels match*: $w$ and $\phi(w)$ have the same label, for every $w \in Q_u$, and (3) *structural match*: the structural relationship between $\phi(w)$ and $\phi(\text{parent}(w))$ matches the axis of $w$, for every $w \in Q, w \neq u$.

A match of a query $Q$ in a document $D$ is a match of $Q_{\text{root}(Q)}$ in $D_{\text{root}(D)}$. Given a query $Q$ and a
document $D$, the *filtering* of $D$ using $Q$, denoted $\text{FILTER}_Q(D)$, is a bit indicating whether $Q$ has at least
one match in $D$. The *pattern matching* of $Q$ in $D$, denoted $\text{PM}_Q(D)$, is the collection of all matches of
$Q$ in $D$. The *full-fledged evaluation* of $Q$ on $D$, denoted $\text{FFE}_Q(D)$, is the collection of elements $\phi(t)$, for all matches $\phi$ of $Q$ in $D$ ($t$ is the output node of $Q$).

**Remark 2.1.** Extending the results of this study to wider fragments of XPath is an open problem. For example, supporting wildcards (*) requires changing the model to having more than one cursor per stream and this affects all the proofs.

### 2.4 Lower bounds

In this section we present our lower bounds for evaluating basic twig queries, for the three evaluation
modes. The proof of each lower bound is based on a reduction from another problem, whose lower
bound we prove separately in Section 2.5. We start by describing our generic reduction scheme, and
then we use it to prove three lower bounds, one for each evaluation mode.

**2.4.1 Techniques**

Our lower bounds are proved via reductions from problems in the *multiple data streams* (MDS) model:
2.4.1.1 The MDS model

In the multiple data streams (MDS) model, the input data \( x \) is divided into several read-only streams, and the required output, \( f(x) \), is written to a write-only output stream. Each of the input streams is associated with a cursor that can move only in the forward direction. The cursor specifies which part of the stream has already been read. An algorithm can read from a cursor position many times, until it decides to advance it, and can use a read/write random access working tape. When the entire input has been read, the output stream contains \( f(x) \).

The space of an algorithm is the number of bits used in the working tape. The algorithm does not 'know' the cursor positions, and therefore if it needs to use a cursor position it must compute and store it in its working tape.

This model generalizes our evaluation model for basic twig queries.

**Definition 2.2 (MDSS).** The MDS space complexity of function \( f \), denoted as \( \text{MDSS}(f) \), is the minimum space required for \( A \), over all algorithms \( A \) that compute \( f \) in the MDS model.

**Definition 2.3 (DSS).** Let \( \text{DSS}(f) \) denote the (single) data stream space complexity of function \( f \), where \( f \) is a function whose input is given in one stream, i.e., \( \text{DSS}(f) = \text{MDSS}(f) \).

2.4.1.2 An MDS-reduction

Let \( f \) be a function in the MDS model over \( k \) streams: \( s_1, \ldots, s_k \), and let \( g \) be a function in the MDS model over \( k+l \) streams: \( t_1, \ldots, t_k \) and \( c_1, \ldots, c_l \). We say \( r = (r_{in}^1, \ldots, r_{in}^k, r_{ou}^1, \ldots, r_{ou}^l, r_{out}) \) is an MDS reduction from \( f \) to \( g \), denoted \( f \leq_{\text{MDS}} g \), if \( r \) satisfies the following:

1. \( r \) is a tuple of functions in the MDS model
2. \( \forall i, 1 \leq i \leq k : r_{in}^i : s_i \rightarrow t_i \)
3. \( \forall i, 1 \leq i \leq l : r_{ou}^i : \epsilon \rightarrow c_i \) (the input is empty)
4. \( r_{out} : \text{Outputs}(g) \rightarrow \text{Outputs}(f) \)
5. \( \forall (s_1, \ldots, s_k) \in \text{Inputs}(f) : r_{out}(g(r_{in}^1(s_1), \ldots, r_{in}^k(s_k), r_{ou}^1(\epsilon), \ldots, r_{ou}^l(\epsilon))) = f(s_1, \ldots, s_k) \)

**Lemma 2.1.** Suppose there exists an MDS reduction \( r = (r_{in}^1, \ldots, r_{in}^k, r_{ou}^1, \ldots, r_{ou}^l, r_{out}) \) from \( f \) to \( g \). Then: \( \text{MDSS}(f) \leq \text{MDSS}(g) + \sum_{i=1}^{k} \text{DSS}(r_{in}^i) + \sum_{i=1}^{l} \text{DSS}(r_{ou}^i) + \text{DSS}(r_{out}) \)

**Proof.** Assume the space optimal algorithms for \( g, r_{in}^1, \ldots, r_{in}^k, r_{ou}^1, \ldots, r_{ou}^l, \) and \( r_{out} \), are \( A_g, A_{r_{in}^1}, \ldots, A_{r_{in}^k}, A_{r_{ou}^1}, \ldots, A_{r_{ou}^l} \), and \( A_{r_{out}} \), respectively. We now build an algorithm \( A_f \) for \( f \), whose space complexity is exactly \( \text{MDSS}(g) + \sum_{i=1}^{k} \text{DSS}(r_{in}^i) + \sum_{i=1}^{l} \text{DSS}(r_{ou}^i) + \text{DSS}(r_{out}) \), by computing:

\[
r_{out}(g(r_{in}^1(s_1), \ldots, r_{in}^k(s_k), r_{ou}^1(\epsilon), \ldots, r_{ou}^l(\epsilon)))
\]
Specifically, we simulate $Ar_{out}$, whose input is the output stream of $g$. Therefore, whenever $Ar_{out}$ advances the cursor of the input stream and reads the next bit, we simulate $Ag$ until it outputs the next bit. But in order to simulate $Ag$, we need to generate on-the-fly its input streams: $t_1, \ldots, t_k$ and $c_1, \ldots, c_l$ (each time we generate the next requested bit). Therefore, whenever $Ag$ advances the cursor of some input stream, say $t_i$, and reads its next bit, we simulate $Ar_{in}^i$ until it outputs the next bit. Note that the input stream of $Ar_{in}^i$ is $s_i$, which is given as the input for $Af$. If $Ag$ advances the cursor of a $c_i$ input stream, then we simulate $Ar_{i,c}^i$, whose input is $\epsilon$.

The algorithm $Af$ we described correctly computes $f$ (by the MDS reduction properties), and its space complexity is $MDSS(g) + \sum_{i=1}^{k} DSS(r_{in}^i) + \sum_{i=1}^{l} DSS(r_{c}^i) + DSS(r_{out})$, since we simulate simultaneously all the corresponding algorithms.

Corollary 2.1. Suppose there exists an MDS reduction from $f$ to $g$. Then: $MDSS(g) \geq MDSS(f) - \sum_{i=1}^{k} DSS(r_{in}^i) - \sum_{i=1}^{l} DSS(r_{c}^i) - DSS(r_{out})$

2.4.2 Filtering

In this section we prove an $\Omega(docDepth)$ lower bound for filtering mode, for any query that contains the $(//)(/)$ pattern (e.g., $//a/b$). Since filtering is easier than the other two modes of evaluation, the $\Omega(docDepth)$ lower bound applies also to both full-fledged evaluation and pattern matching of queries that consist of the $(//)(/)$ pattern. This lower bound is matched by PathStack [14] for pattern matching of path queries, and by TurboXPath [52] for filtering any query.

Theorem 2.1. Let $a,b$ be any two labels, and let $Q$ be any basic twig query that contains the path segment $//a/b$. Furthermore, assume $a \neq b$ and that $a,b$ do not appear elsewhere in $Q$. Then, for every algorithm for $\text{FILTER}_Q$ and for every $d \geq 1$, there exists a document of depth at most $d - 1 + \text{depth}(Q)$, on which the algorithm uses at least $d - O(|Q| \log(|Q| \cdot d))$ bits of space.

We first prove the theorem for the special case $Q = //a/b$. This proof captures the main technical challenges of the general case. We then present the proof of the general case.

The difficulty in finding whether the query $//a/b$ has a match or not emanates from recursive documents that contain multiple nodes named ‘$a$’ nested within each other. For example, consider the document structure in Figure 2.2. Assume that some of the $s_i$ elements are labeled ‘$a$’, and some of the $t_j$ elements are labeled ‘$b$’. Any filtering algorithm will have to match the list of nested ‘$a$’ elements read from the stream $T_a$ against the list of ‘$b$’ elements read from the stream $T_b$. The query has a match if and only if one of these ‘$b$’ elements is a child of an ‘$a$’ element. Due to the pre-order organization of elements in the two streams, $T_a$ has to be matched against the reverse of $T_b$. This implies that no algorithm can match the two streams on the fly, but rather has to store at least one of them in memory.
As the length of these streams can be almost as long as the document’s depth, this implies the space lower bound.

Formally, the theorem is proven by an MDS reduction from the problem of reverse-set-disjointness in the MDS model.

2.4.2.1 Reverse set disjointness

The reverse of a binary vector \( y = (y_1, \ldots, y_n) \in \{0, 1\}^n \) is the binary vector \( y^R = (y_n, \ldots, y_1) \), i.e., \( y^R_i = y_{n+1-i} \).

Given two binary vectors: \( x, y \in \{0, 1\}^n \), the reverse set disjointness function, \( \text{RDISJ}_n(x, y) \), is defined to be 1 if \( \exists i \), s.t. \( x_i = y^R_i = 1 \), and 0 otherwise. When computed in the MDS model, \( x \) is given on one stream and \( y \) on another stream.

**Theorem 2.2.** For any \( n \geq 7 \), the space complexity of \( \text{RDISJ}_n \) in the MDS model is at least \( n - \log(n + 1) - 3 \).

The proof appears in Section 2.5.3.

2.4.2.2 The reduction

We prove Theorem 2.1 for the case \( Q = \langle a/b \rangle \) by an MDS reduction from \( \text{RDISJ}_n \) to \( \text{FILTER}_Q \). Note that \( \text{FILTER}_Q \) can also be described as a function over two streams: \( \text{FILTER}_Q(T_a, T_b) \), as we restricted ourselves to algorithms that access only streams corresponding to labels that occur in the query (see Section 2.3). Let \( n = d - 1 \). The MDS reduction is based on the following functions:

- \( r_{\text{in}}^1 \) and \( r_{\text{in}}^2 \) construct the index streams \( T_a \) and \( T_b \), respectively, of the XML document \( D(x, y) \) (see Figure 2.2). Specifically, \( r_{\text{in}}^1(x) = T_a \) and \( r_{\text{in}}^2(y) = T_b \). \( D(x, y) \) is the same for all \( (x, y) \), except for the labeling of elements. When \( x_i = 1 \), the corresponding element \( s_i \) is labeled ’a’, and otherwise it is labeled ’c’. When \( y_i = 1 \), the corresponding element \( t_i \) is labeled ’b’, and otherwise it is labeled ’d’.

- \( r_{\text{out}}(q) = q \) (note that the output of both \( \text{RDISJ}_n \) and \( \text{FILTER}_Q \) is one bit). Therefore \( \text{DSS}(r_{\text{out}}) = 0 \).

**Claim 2.1.** \( \text{DSS}(r_{\text{in}}^1) = \text{DSS}(r_{\text{in}}^2) \leq \log n \).

**Proof.** We describe algorithms \( A \) and \( B \) for \( r_{\text{in}}^1 \) and \( r_{\text{in}}^2 \), resp. In order to output the next tuple in \( T_a \) [\( T_b \)], \( A \) [\( B \)] advances the stream \( x \) [\( y \)] to the next set bit. If the position of this bit is \( i \), the algorithm creates the tuple \( (i, 4n - 3i + 3, i) \) in \( T_a \) [(\( n + 3i - 2, n + 3i - 1, n + 2 - i \)) in \( T_b \)]. If no set bit is found in \( x \) [\( y \)], the algorithm creates a \( T_a, \text{EndOfStream} \) [\( T_b, \text{EndOfStream} \)] tuple.
It is easy to check that the index streams constructed are well-formed, i.e., sorted by the “Begin” attribute, and that they represent the document $D(x, y)$ whose depth is $d$. The space needed for $A[B]$ is $\log n$ bits for keeping the current position in $x[y]$.

**Lemma 2.2.** $\text{RDISJ}_n(x, y) = \text{FILTER}_Q(D(x, y))$.

**Proof.** $\text{RDISJ}_n(x, y) = 1$ if and only if there exists some index $1 \leq i \leq n$, such that both $x_i$ and $y_i^R$ (i.e., $y_{n+1-i}$) are 1. This means that in $D(x, y)$ the label of $s_i$ is ‘$a$’ and the label of $t_{n+1-i}$ is ‘$b$’. Since $t_{n+1-i}$ is a child of $s_i$, the latter happens iff there is a match of $Q$ in $D(x, y)$. 

Since $r_1^i$ and $r_2^i$ construct the index streams $T_a$ and $T_b$ of $D(x, y)$, it follows that:

$$r_{out}(\text{FILTER}_Q(r_1^i(x), r_2^i(y))) = \text{FILTER}_Q(r_1^i(x), r_2^i(y)) = \text{RDISJ}_n(x, y)$$

Therefore, by Corollary 2.1, $\text{MDSS}(\text{FILTER}_Q) \geq \text{MDSS}(\text{RDISJ}_n) - \text{DSS}(r_1^i) - \text{DSS}(r_2^i) - \text{DSS}(r_{out}) \geq n - O(\log n) = d - O(\log d)$.

Now that we have proved Theorem 2.1 for the special case $Q = /ab$, we present a proof sketch of the general case. A full proof of the theorem appears in Appendix 2.A.

**Proof sketch of Theorem 2.1.** Similarly to the proof of the special case, the functions $r_1^i$ and $r_2^i$ construct the index streams $T_a$ and $T_b$, respectively, of an XML document. The document’s labels, as before, depend on $x$ and $y$. However, since now the query is some arbitrary tree that contains the path segment //ab, the document structure should match the query tree except, maybe, for the segment //ab. This means that the document structure will resemble that of the special case (see Figure 2.2), but we add all the other query nodes ”around” these potentially $a(s_i)$ and $b(t_i)$ nodes. This way the existence of a match of $Q$ in this document depends only on the labels of $s_i$ and $t_i$ nodes, i.e., on the set bits in $x$ and $y$. Note that since the query may consist of additional labels, the reduction should build their index streams too. However, these index streams do not depend on $x, y$, and can be "encoded" in the reduction. 

21
Remark 2.2 (Regarding the case $a = b$). Theorem 2.1 required that the labels $a, b$ are different and unique in $Q$. If this requirement is not met, then proving the filtering lower bound requires a lower bound for a variant of the reverse-set-disjointness problem. The reason is that now the streams $T_a$ and $T_b$ are the same, i.e., the cursors of both $a$ and $b$ point to the same stream. The corresponding variant of the reverse-set-disjointness problem (in order to use the same MDS reduction shown above) is where both streams contain $x \circ y$. We do not have a lower bound proof for this case.

2.4.3 Full-fledged evaluation

The $\Omega(\text{docDepth})$ lower bound we proved for filtering applies also to full-fledged evaluation of queries that consist of the $(//)/(\land)$ pattern. In this section we prove that full-fledged evaluation of some of these queries is subject to an additional $\Omega(\text{DOE})$ lower bound. Strictly speaking, the lower bound does not apply to all queries that contain $(//)/(\land)$, but rather only to a subset of them, depending on the location of the output node in the query. These two lower bounds are combined to an $\Omega(\max(\text{docDepth}, \text{DOE}))$ lower bound for this fragment of queries, as shown in Table 2.1 (see the middle table, lower right corner).

For a query $Q$ and a document $D$, we define the DOE of $\text{FFE}_Q(D)$, denoted by $\text{DOE}(\text{FFE}_Q(D))$, to be the number of distinct document elements to which the query’s output node can be matched.

Theorem 2.3 (DOE lower bound). Let $Q$ be any basic twig query that contains the path segment $(//)$. Furthermore, assume the following: (1) the output node is a descendant of the node labeled $z$ but not of the node labeled $b$ (this is where we require $Q$ to be a tree and not a path); (2) the output node’s label and $z, b$ are distinct and do not appear elsewhere in $Q$. Then, for every algorithm for $\text{FFE}_Q$ and for every $S \geq 1$, there exists a document $D$, for which $\text{DOE}(\text{FFE}_Q(D)) \leq S$ and on which the algorithm uses at least $\Omega(S)$ bits of space.

Here, we prove the theorem for the special case $Q = //z[/ba$. This proof captures the main technical challenges of the general case. Formally, the theorem is proven by an MDS reduction from the following variant of the set-disjointness problem, which we call delayed intersection:

2.4.3.1 Delayed intersection

Given three binary vectors $s, t, u \in \{0, 1\}^n$ and a bit $v \in \{0, 1\}$, the delayed intersection function, $\text{DINT}_n(s, t, u, v)$, is defined as $(s \cap v^n) \circ (t \cap u)$, where $\cap$ denotes bitwise-and, $v^n$ is the $n$-dimensional vector obtained by taking $n$ copies of $v$, and $\circ$ denotes concatenation of vectors. For example, $\text{DINT}_n(101, 011, 101, 1)$ is 101001. When computed in the MDS model, $s \circ t$ is given on one stream and $u \circ v$ on another stream.

---

7DOE denotes the number of distinct output elements.
Theorem 2.4. For any \( n \geq 7 \), the space complexity of \( \text{DINT}_n \) in the MDS model is at least \( n - \log(n + 1) - 3 \).

The proof appears in Section 2.5.4.

2.4.3.2 The reduction

We prove Theorem 2.3 for the case \( Q = \ll z/[b]/a \) by an MDS reduction from \( \text{DINT}_n \) to \( \text{FFE}_Q \). Let \( n = S/2 \). The MDS reduction is based on the following functions:

- \( r_1^1(s \circ t) \) and \( r_2^2(u \circ v) \) construct the index streams \( T_a \) and \( T_b \), respectively, of an XML document \( G(s,t,u,v) \). The document structure, which is presented in Figure 2.3, is the same for all inputs \( (s,t,u,v) \), except for the labeling of elements. When \( s_i = 1 \) or \( t_i = 1 \), the corresponding element \( s_i \) or \( t_i \) is labeled 'a', and otherwise it is labeled 'c'. When \( u_i = 1 \) or \( v = 1 \), the corresponding element \( u_i \) or \( v_0 \) is labeled 'b', and otherwise it is labeled 'd'.
- \( r_1^1 \) constructs the index stream \( T_z \) of the document \( G(s,t,u,v) \). Note that \( T_z \) is fixed and does not depend on the input.
- \( r_{out} \): its input is a stream of \( a \)-elements from the document \( G \), and it constructs a \( 2n \)-bit vector, whose set bits correspond to the position of the \( a \)-elements. Specifically, an \( a \) element may be an \( (s_i) \) or a \( (t_i) \) node, and accordingly the position of the set bit is \( i \) or \( n + i \), respectively. The position \( i \) may be computed from the element’s tuple, and therefore \( \text{DSS}(r_{out}) \leq \log n \).

![Figure 2.3: The document \( G(s,t,u,v) \).](image)

Claim 2.2. \( \text{DSS}(r_1^1) = \text{DSS}(r_2^2) \leq \log n \).

Proof. We describe algorithms \( A \) and \( B \) for \( r_1^1 \) and \( r_2^2 \), resp. In order to output the next tuple in \( T_a \) [\( T_b \), \( A \) [\( B \)] advances the stream \( s \circ t \) [\( u \circ v \)] to the next set bit. The tuple created is a simple function of the position of this bit.
It is easy to check that the index streams constructed are well-formed, i.e., sorted by the “Begin” attribute, and that they represent $T_a$ and $T_b$ of the document $G(s, t, u, v)$. The space needed for $A[B]$ is $\log n$ bits for keeping the current position in the input stream $s \circ t [u \circ v]$.

\[
\text{Claim 2.3. } \text{dss}(r_1^c) \leq \log n
\]

\textbf{Proof.} The index stream $T_2$ can be generated on-the-fly, based on the position of the required tuple. Therefore, only $\log n$ bits of space are needed, to maintain this position.

The following proposition proves the validity of the reduction:

\textbf{Proposition 2.1.} Let $1 \leq k \leq 2n$. The $k$-th bit in $\text{dint}_n(s, t, u, v)$ is set iff:

(i) if $k \leq n$, $s_k \in \text{ffe}_Q(G(s, t, u, v))$, and (ii) if $k > n$, $t_{k-n} \in \text{ffe}_Q(G(s, t, u, v))$.

\textbf{Proof.} If $k \leq n$, then the $k$-th bit is in $(s \cap v^a)$, and it is set iff both $s_k$ and $v$ are set. This means that the labels of elements $s_k$ and $v_0$ in the document $G$ are 'a' and 'b', respectively. The latter happens if and only if the mapping $\phi = (a \mapsto s_k, b \mapsto v_0)$ is a match of $Q$ in $G$. If $\phi$ is a match, then $s_k \in \text{ffe}_Q(G(s, t, u, v))$. We only have to prove now that if $s_k \in \text{ffe}_Q(G(s, t, u, v))$, then $\phi$ is a match, i.e., prove that the labels of elements $s_k$ and $v_0$ are 'a' and 'b', respectively. Since $s_k$ was output, then there is a match that maps $a \mapsto s_k$. As the only element in $G$ that is a child of $s_k$’s parent and may have a 'b' label is $v_0$, the only possible match is $\phi$.

The second case is when $k > n$. Now the corresponding bit is the $(k-n)$-th bit in $(t \cap u)$, which is set iff both $t_{k-n}$ and $u_{k-n}$ are set. The proof here is similar to the previous case, but with elements $t_{k-n}$ and $u_{k-n}$ instead of $s_k$ and $v_0$.

Since $(r_1^1, r_2^1, r_c^1)$ construct the index streams of $G(s, t, u, v)$, it follows that:

\[
r_{out}(\text{ffe}_Q(r_1^1(s \circ t), r_2^1(u \circ v), r_c^1(\epsilon))) = \text{dint}_n(s, t, u, v)
\]

Therefore, by Corollary 2.1,

\[
\text{mdss}(\text{ffe}_Q) \geq \text{mdss}(\text{dint}_n) - \text{dss}(r_1^1) - \text{dss}(r_2^1) - \text{dss}(r_1^1) - \text{dss}(r_{out}) \geq n - O(\log n) = \Omega(S)
\]

Note also that the DOI of $\text{ffe}_Q(G(s, t, u, v))$ is at most $2n = S$. This concludes the proof of Theorem 2.3 for the special case $Q = \|z[\{b]\}a$.

\textbf{Remark 2.3} (Regarding the proof of the general case). \textit{The proof of Theorem 2.3 for the general case, similarly to the proof of the special case given above, is also based on an MDS reduction from the delayed intersection problem. However, the document structure should be extended, similarly to the extension}
we made in the proof of Theorem 2.1 (see Appendix 2.A) to ensure all other nodes in the query (except for \( z, a, b \)) can be matched. Therefore all subtrees surrounding the \///z/b segment in the query should be replicated in the document around each \( z \)-element.

### 2.4.4 Pattern matching

In this section we present two lower bounds for computing all the matches of basic twig queries. First, we show that computing all the matches of path queries that have at least one non-leaf descendant-axis node requires \( \Omega(\text{docDepth}) \) space. This lower bound is matched by the PathStack algorithm of Bruno et al. [14]. Then, we prove that the situation with tree queries is quite different: computing the matches of any tree query (regardless of the axis pattern) requires \( \Omega(\text{DOE}) \) space.

#### 2.4.4.1 Pattern matching for path queries

We now prove that computing all the matches of a path query that has at least one non-leaf descendant-axis node requires \( \Omega(\text{docDepth}) \) space. For the case the non-leaf descendant-axis node is followed by a child-axis node, the lower bound immediately follows from the \( \Omega(\text{docDepth}) \) space lower bound in the filtering mode (Theorem 2.1). What we prove here is that even if the axis pattern of the path is \((//)^*//)\) (i.e., the path ends with two or more descendant-axis nodes), then \( \Omega(\text{docDepth}) \) space is needed. For such queries (e.g., //a//b), we therefore have a gap between evaluation in the filtering mode, in which constant space is sufficient, and evaluation in the pattern matching mode, where \( \Omega(\text{docDepth}) \) space is needed. The lower bound is matched by the PathStack and TwigStack algorithms of Bruno et al. [14].

To gain some intuition for the hardness of queries like //a//b, consider a document that contains a path of \( m \) a’s followed by \( n \) b’s. In order to output all the \( mn \) matches of the query //a//b in this document, any algorithm will have either to store all the a’s before starting to read the b’s or vice versa. This gives a \( \min\{m, n\} \) space lower bound. The intuition is formalized in the following theorem:

**Theorem 2.5.** Let \( Q \) be any path query of the form /c_1/c_2/.../c_\ell///a_1///a_2///a_k, where \( \ell \geq 0, k \geq 2 \), and \( c_1, \ldots, c_\ell, a_1, \ldots, a_k \) are distinct labels. Then, for every algorithm for \( \text{PM}_Q \) and for every \( d \geq 1 \), there exists a document of depth at most \( d \), on which the algorithm uses at least \( \Omega(d) \) bits of space.

We start with a proof of the theorem for the special case \( Q = //a//b \). We then extend the proof to deal with arbitrary queries in the above XPath fragment. The proof is based on an MDS reduction from the tensor product problem:

**Definition 2.4 (Tensor product).** Given two binary vectors \( x, y \) of lengths \( m \) and \( n \), respectively, the tensor product of \( x \) and \( y \), denoted \( x \otimes y \), is a vector of length \( mn \) whose \((i, j)\)-th entry is \( x_i \cdot y_j \). The
required output is a list of indices of the set bits in \( x \otimes y \), in arbitrary order. The tensor product of \( k \) vectors \( x_1, \ldots, x_k \) is defined similarly by induction.

The following is a space lower bound for the tensor product problem in the MDS model. We assume \( x \) is given on one stream and \( y \) on another stream. The proof appears in Section 2.5.5.

**Theorem 2.6.** The space complexity of computing \( x \otimes y \) (\( x \) and \( y \) are of lengths \( m \) and \( n \), respectively) in the MDS model is at least \( \min(m, n) - 3 \) bits.

**The reduction** We prove Theorem 2.5 for the case \( Q = \ll a b \rr \) by an MDS reduction from the tensor product problem to \( \text{PM}_{\ll a b} \). Let \( m = n = d/2 \) (\( m, n \) are the lengths of the input to \( x \otimes y \)). The MDS reduction is based on the following functions:

- \( r_{in}^1(x) \) and \( r_{in}^2(y) \) construct the index streams \( T_a \) and \( T_b \), respectively, of the XML document \( E(x, y) \) (see Figure 2.4). \( E(x, y) \) is the same for all \( (x, y) \), except for the labeling of elements. When \( x_i = 1 \), the corresponding element \( s_j \) is labeled ‘\( a \)’, and otherwise it is labeled ‘\( c \)’. When \( y_i = 1 \), the corresponding element \( t_i \) is labeled ‘\( b \)’, and otherwise it is labeled ‘\( d \)’.

- \( r_{out} \): its input is a stream of matches, i.e., pairs of \((a, b)\)-elements, from the document \( E \), and the output is a list of indices that correspond to the position of the \( a \) and \( b \) elements in the input. Specifically, an input match \((s_i, t_j)\) is translated on-the-fly to the output: “\((x \otimes y)_{i,j} = 1\)”. Therefore, \( \text{dss}(r_{out}) \leq \log n \).

Figure 2.4: The document \( E(x, y) \).
Claim 2.4. \( \text{DSS}(r_{in}^1) = \text{DSS}(r_{in}^2) \leq \log n. \)

Proof. We describe algorithms \( A \) and \( B \) for \( r_{in}^1 \) and \( r_{in}^2 \), resp. In order to output the next tuple in \( T_a \) \([T_b]\), \( A \) \([B]\) advances the stream \( x[y] \) to the next set bit. If the position of this bit is \( i \), the algorithm creates the tuple \( (i, 2m + 2n + 1 - i, i) \) in \( T_a \) \([T_b]\) \([(m + j, m + 2n + 1 - j, m + j)\) in \( T_b\)]. If no set bit is found in \( x[y] \), the algorithm creates a \( T_a \).EndOfStream \([T_b\).EndOfStream\) tuple.

It is easy to check that the index streams constructed are well-formed, i.e., sorted by the “Begin” attribute, and that they represent the document \( E(x, y) \) whose depth is \( d \). The space needed for \( A \) \([B]\) is \( \log n \) bits for keeping the current position in \( x[y] \). \( \square \)

Proposition 2.2. \((x \otimes y)_{i,j} = 1 \) iff the match \((a \mapsto s_i, b \mapsto t_j)\) belongs to \( PM_Q(E(x, y)) \).

Proof. \((x \otimes y)_{i,j} = 1 \) if and only if \( x_i = y_j = 1 \). This means that in \( E(x, y) \) the label of \( s_i \) is 'a' and the label of \( t_j \) is 'b'. The latter happens if and only if \((a \mapsto s_i, b \mapsto t_j)\) is a match of \( l/a/l/b \) in \( E(x, y) \). \( \square \)

Since \( r_{in}^1 \) and \( r_{in}^2 \) construct the index streams \( T_a \) and \( T_b \) of \( E(x, y) \), it follows from the proposition above that:

\[
r_{out}(PM_Q(r_{in}^1(x), r_{in}^2(y))) = r_{out}(PM_Q(E(x, y))) = x \otimes y
\]

Therefore, by Corollary 2.1,

\[
\text{MDSS}(PM_Q) \geq \text{MDSS}(x \otimes y) - \text{DSS}(r_{in}^1) - \text{DSS}(r_{in}^2) - \text{DSS}(r_{out}) \geq n - 3 - O(\log n) = \Omega(d)
\]

We have proved Theorem 2.5 for the special case \( Q = l/a/l/b \). The extension of the proof to arbitrary path queries of the above form is quite straightforward. We therefore provide only a proof sketch. The proof relies on a reduction from the tensor product of \( k \) vectors. For the latter, we have the following lower bound, which is proven in Section 2.5.5:

Theorem 2.7. Given \( k \) vectors \( x_1, x_2, \ldots, x_k \) of dimensions \( m_1, \ldots, m_k \), respectively, the space complexity of computing \( x_1 \otimes x_2 \otimes \cdots \otimes x_k \) in the MDS model is at least \( \sum_{i=1}^k m_i - \max_i \{m_i\} - \log k - k + 1 \).

Proof of Theorem 2.5 (Sketch). We use a very similar MDS reduction from the tensor product of \( k \) vectors \( x_1, \cdots, x_k \) of lengths \( m_1, \cdots, m_k \), where \( \sum_{i=1}^k m_i = d - \ell \). Similarly to the document \( E(x, y) \) we constructed above (see Figure 2.4), the document \( E(x_1, \cdots, x_k) \) we construct now is a path. There are two differences, however: (1) \( E(x_1, \ldots, x_k) \) consists of \( k \) nested chains of elements (corresponding to \( a_1, \ldots, a_k \)) rather than just two; and (2) the path begins with the prefix /\( c_1/c_2/\ldots/c_\ell \). \( \square \)

2.4.4.2 Pattern matching for tree queries

As opposed to the filtering and the full fledged evaluation modes, pattern matching of all tree queries incurs high space costs. For a query \( Q \) and a document \( D \), we define the \( \text{DOE} \) of \( PM_Q(D) \) to be the
The number of distinct elements that occur in the matches in $PM_Q(D)$.

**Theorem 2.8.** Let $Q$ be any basic twig query which is a tree, and has at least two leaves which are labeled by distinct labels that do not occur elsewhere in $Q$. Then, for every algorithm for $PM_Q$ and for every $s \geq 1$, there exists a document $D$, for which the DOE of $PM_Q(D)$ is at most $s$ and on which the algorithm uses at least $\Omega(s)$ bits of space.

This theorem too is proven via an MDS reduction from the tensor product problem. We first prove the theorem for the special case $Q = /a[/b$ and $]c]$.  

![Figure 2.5: The document $F(x, y)$.](Figure2.5)

**The reduction** Let $n = m = \frac{s-1}{2}$ ($m, n$ are the lengths of $x, y$, resp.). The MDS reduction is based on the following functions:

- $r_{in}^1(x)$ and $r_{in}^2(y)$ construct the index streams $T_b$ and $T_c$, respectively, of the XML document $F(x, y)$ (see Figure 2.5). $F(x, y)$ is the same for all $(x, y)$, except for the labeling of elements. When $x_i = 1$, the corresponding element $s_i$ is labeled ’$b$’, and otherwise it is labeled ’$c$’. When $y_i = 1$, the corresponding element $t_i$ is labeled ’$c$’, and otherwise it is labeled ’$f$’.

- $r_{c}$ constructs the index stream $T_a$ of the document $F(x, y)$, which consists of only one tuple. Note that $T_a$ is fixed and does not depend on the input, and therefore $DSS(r_{c}) = O(\log n)$.

- $r_{out}$: its input is a stream of matches, i.e., tuples of $(a, b, c)$-elements, from the document $E$, and the output is a list of indices that correspond to the position of the $b$ and $c$ elements in the input. Specifically, an input match $(a, s_i, t_j)$ is translated on-the-fly to the output: “$(x \otimes y)_{i,j} = 1$”. Therefore, $DSS(r_{out}) = O(\log n)$.

**Claim 2.5.** $DSS(r_{in}^1) = DSS(r_{in}^2) \leq \log n$.

**Proof.** We describe algorithms $A$ and $B$ for $r_{in}^1$ and $r_{in}^2$, resp. In order to output the next tuple in $T_b[T_c]$, $A [B]$ advances the stream $x[y]$ to the next set bit. The corresponding tuple can be easily computed based on the position of that bit. If no set bit is found in $x[y]$, the algorithm creates a $T_b.EndOfStream [T_c.EndOfStream]$ tuple.
It is easy to check that the index streams constructed are well-formed, i.e., sorted by the “Begin” attribute, and that they represent the document $F(x,y)$. Note also that the DOE of $\text{PM}_Q(F(x,y))$ is at most $2m + 1 = s$, as required by the theorem. The space needed for $A[B]$ is $O(\log n)$ bits for keeping the current position in $x[y]$.

**Proposition 2.3.** $(x \otimes y)_{i,j} = 1$ iff the match $(a \leftrightarrow a, b \leftrightarrow s_i, c \leftrightarrow t_j)$ belongs to $\text{PM}_Q(F(x,y))$.

**Proof.** $(x \otimes y)_{i,j} = 1$ if and only if $x_i = y_j = 1$. This means that in $F(x,y)$ the label of $s_i$ is ’b’ and the label of $t_j$ is ’c’. The latter happens if and only if $(a \leftrightarrow a, b \leftrightarrow s_i, c \leftrightarrow t_j)$ is a match of $/a[/b$ and $/c]$ in $F(x,y)$.

Since $r^1_c, r^1_{in}, r^2_{in}$ construct the index streams of $F(x,y)$, it follows from the proposition above that:

$$r_{\text{out}}(\text{PM}_Q(r^1_c, r^1_{in}(x), r^2_{in}(y))) = r_{\text{out}}(\text{PM}_Q(F(x,y))) = x \otimes y$$

Therefore, by Corollary 2.1,

$$\text{MDSS}(\text{PM}_Q) \geq \text{MDSS}(x \otimes y) - \text{DSS}(r^1_{in}) - \text{DSS}(r^2_{in}) - \text{DSS}(r^1_c) - \text{DSS}(r_{\text{out}}) \geq n - 3 - O(\log n) = \Omega(s)$$

In order to prove Theorem 2.8 for any tree query $Q$ with two distinct leaves, we slightly change the reduction shown in the special case. Instead of the document $F(x,y)$ built on-the-fly (see Figure 2.5), now the document will be of the same structure and labels of $Q$, except for the two distinct query leaves, which will be replaced with the nodes $s_1, \ldots, s_m$ and $t_1, \ldots, t_m$ (as in the document $F(x,y)$).

### 2.5 The TMC model

In this section we present a new deterministic model of communication, the *token-based mesh communication model* (TMC), which can be used to prove space lower bounds in the MDS model. After investigating basic properties of protocols in the model, we use them to prove lower bounds for three problems: reverse set disjointness (Theorem 2.2), delayed intersection (Theorem 2.4), and tensor product (Theorems 2.6, 2.7). We note that these lower bounds could have been proved directly through the MDS model, without using the new TMC model. However, we believe that the TMC model presents in an explicit way the various possible computations of a multiple data stream algorithm, i.e., the various cursor configurations, and enables cleaner and more modular proofs.

#### 2.5.1 The TMC model

In the *token-based mesh communication* (TMC) model, there are $n$ players, who wish to jointly compute a function $f$ on a shared input $x \in \{0,1\}^m$. The input $x$ is viewed as a concatenation of $d$ strings
of lengths $m_1, m_2, \ldots, m_d$, respectively, where $x_i \in \{0, 1\}^{m_i}$. The players are placed on nodes of a network, whose underlying topology is a $d$-dimensional mesh. Specifically, the set of nodes is $V = [m_1] \times [m_2] \times \cdots \times [m_d]$, where $[m_i] = \{0, 1, \ldots, m_i\}$. Node $(i_1, i_2, \ldots, i_d)$ has an outgoing edge to $(i_1, i_2, \ldots, i_d) + e_j$, for all $j$ for which $i_j < m_j$. Here, $e_j$ is the $d$-dimensional $j$-th standard unit vector.

Every player receives as input the bits from the input strings that correspond to its position in the mesh, i.e., a player on node $(i_1, i_2, \ldots, i_d)$ receives $(b_{1,i_1}, b_{2,i_2}, \ldots, b_{d,i_d})$, where $b_{j,k}$ is the $k$-th bit in $x_j$, for $1 \leq k \leq m_j$, and is 0, for $k = 0$.

The communication in the network is not in broadcast, as is in the more standard models, but is token-based. At each round of the protocol, a single player holds a “token”, indicating she is the only one who can send messages in the round. She sends a single private message to one of her outgoing neighbors. The neighbor who receives the message holds the token at the next round. The communication is deterministic, and always starts at the node $s = 0^d$ (the “start player”) and ends at the node $t = (m_1, m_2, \ldots, m_d)$ (the “end player”). All players share a write-only output stream, to which only a player who holds the token can write. The stream should contain the value $f(x)$ by the end of the protocol.

The max communication cost of a protocol $P$ in this model is the length of the longest message sent during execution of $P$ on the worst-case choice of input $x$.

Figure 2.6 shows a 2-dimensional mesh, with $m_1 = m_2 = n$.

![2-dimensional mesh](image)

**Figure 2.6:** A 2-dimensional mesh, with $m_1 = m_2 = n$, in the TMC model.

The following shows a reduction from the TMC model to the MDS model:

**Lemma 2.3** (Reduction lemma). Let $f : \{0, 1\}^{m_1} \times \{0, 1\}^{m_2} \times \cdots \times \{0, 1\}^{m_d} \rightarrow B$. If there exists an algorithm that computes $f$ in the MDS model with $S$ bits of space, then there exists a protocol that
computes \( f \) in the \( d \)-dimensional TMC model whose max communication cost is at most \( S \) bits.

**Proof.** Let \( A \) be an algorithm that computes \( f \) in the MDS model with \( S \) bits of space. The input of \( A \) is \( d \) streams \( \{x_1, \ldots, x_d\} \) of sizes \( \{m_1, \ldots, m_d\} \), respectively. In the TMC model we have a \( d \)-dimensional mesh, where every dimension corresponds to one stream. Each setting of the \( d \) cursors in the MDS model corresponds to one player in the \( d \)-dimensional mesh. We now describe a protocol \( P \) that computes \( f \) in the TMC model. At the beginning, player \( s \) (i.e., \( 0^d \)) holds the token, and it has received no input (by definition). It starts to execute the algorithm \( A \). Whenever \( A \) moves a cursor, say the cursor of stream \( x_i \), the player who currently holds the token, denoted as \( (i_1, i_2, \ldots, i_d) \), sends the token together with the current content of the memory of \( A \) (\( S \) bits) to player \( (i_1, i_2, \ldots, i_d) + e \). The player who receives the token and the memory-content continues the execution of \( A \) at the same way. Whenever \( A \) writes to the output stream, the simulating player does the same. Player \( t \) is the last to execute \( A \), and it finishes the simulation. Note that \( P \) correctly computes \( f \) (because \( A \) does) and its max communication is \( S \). □

### 2.5.2 Properties of the TMC model

We now investigate some basic properties of the TMC model, which are crucial to our lower bound proofs. For simplicity of exposition, we focus on 2-dimensional meshes, yet the definitions and results can be easily extended to \( d \)-dimensional meshes as well.

Let \( P \) be a protocol that computes \( f(x, y) \) in the 2-dimensional TMC model.

**Definition 2.5** (Communication path). The communication path of protocol \( P \) on input \( (x, y) \), denoted \( \text{PATH}(x, y) \), is the sequence of players \( \{(i, j)\} \) through whom the token passes during the execution of \( P \) on \( (x, y) \).

The following fact states that all communication paths must pass through the diagonals \( i + j = C \):

**Proposition 2.4.** \( \forall 1 \leq C \leq \min(m_1, m_2) \) and \( \forall x, y, \exists 0 \leq i \leq C, \text{ such that } (i, C - i) \in \text{PATH}(x, y) \).

**Proof.** The diagonal \( i + j = C \) is an \((s, t)\)-cut, and thus any path from \( s \) to \( t \) crosses this cut. □

**Definition 2.6** (Passing set). The passing-input-set of protocol \( P \) w.r.t. player \( (i, j) \), denoted \( \text{PASS}(i, j) \), is the set of all inputs \( (x, y) \) s.t. \( (i, j) \in \text{PATH}(x, y) \).

Let \( \text{PREF}_i(x) \) denote the first \( i \) bits of \( x \), and let \( \text{SUFF}_i(x) \) denote the last \( i \) bits of \( x \), for \( 0 \leq i \leq m_1 \). For \( i = m_1 + 1 \), \( \text{SUFF}_{m_1+1}(x) = 0 \circ x \). (The same goes for \( m_2 \).)

Let player \( (i, j) \in \text{PATH}(x, y) \). \( \text{MSG}_{i,j}(x, y) \) denotes the message sent by player \( (i, j) \) during the execution of \( P \) on input \( (x, y) \). \( \text{SUCC}_{i,j}(x, y) \) denotes the successor of \( (i, j) \) in \( \text{PATH}(x, y) \).

**Definition 2.7** (Packet). Let player \( (i, j) \in \text{PATH}(x, y) \). The packet sent by \( (i, j) \), denoted \( \text{PACKET}_{i,j}(x, y) \), is defined as the combination of \( \text{MSG}_{i,j}(x, y) \) and \( \text{SUCC}_{i,j}(x, y) \).
Definition 2.8 (Prefix set). The passing-prefix-set of protocol P w.r.t. player \((i,j)\), denoted \(\text{PREF}(i,j)\), is the set: \(\{(\text{PREF}_i(x), \text{PREF}_j(y)) \mid (x,y) \in \text{PAS}(i,j)\}\)

Proposition 2.5. \(|\text{PAS}(i,j)| = |\text{PREF}(i,j)| \cdot 2^{(m_1-i)+(m_2-j)}\)

Proof. For every \((\alpha,\beta) \in \text{PREF}(i,j)\), \(\alpha \in \{0,1\}^i\) and \(\beta \in \{0,1\}^j\), there are \(2^{(m_1-i)+(m_2-j)}\) different continuations to inputs in \(\text{PAS}(i,j)\).

The following lemma states that the execution (i.e., packets and output) from any given node on the communication path depends solely on the message received by that node and on the input bits from that node and forward. Given that, the future execution is conditionally independent of the history, i.e., of previous messages and input bits.

Lemma 2.4 (History independence). Let \((x,y), (x',y') \in \text{PAS}(i,j)\). If \(\text{PACKET}_{i,j}(x,y) = \text{PACKET}_{i,j}(x',y')\), \(\text{SUFF}_{m_1-i+1}(x) = \text{SUFF}_{m_1-i+1}(x')\), and \(\text{SUFF}_{m_2-j+1}(y) = \text{SUFF}_{m_2-j+1}(y')\), then the output written from the time player \((i,j)\) sent his packet and until the end is the same on both inputs.

Proof. Let \(\text{PATH}_k(x,y)\) denote the \(k\)-th player in \(\text{PATH}(x,y)\), for \(0 \leq k \leq m_1 + m_2\). Note that, by the structure of the mesh, \((i,j) = \text{PATH}_{i+j}(x,y) = \text{PATH}_{i+j}(x',y')\). We now show that from the time player \((i,j)\) sends his packet and until the end, the computation (i.e., packets) on both input pairs is exactly the same. It then follows that the output written during this time is also the same.

Specifically, we prove by induction on \(k\), \(k = i + j, \ldots, m_1 + m_2\), that (i) \(\text{PATH}_k(x,y) = \text{PATH}_k(x',y')\); and if \((i',j')\) is the \(k\)-th player in both communication paths, then (ii) \((i',j')\) sends the same packet on both input pairs.

For \(k = i + j\), \(\text{PATH}_{i+j}(x,y) = \text{PATH}_{i+j}(x',y') = (i,j)\), and we know that \(\text{PACKET}_{i,j}(x,y) = \text{PACKET}_{i,j}(x',y')\). By the induction assumption, for any \(k < (m_1 + m_2)\), \(\text{PATH}_k(x,y) = \text{PATH}_k(x',y') = (i',j')\), where \(i' \geq i\) and \(j' \geq j\), and \(\text{PACKET}_{i',j'}(x,y) = \text{PACKET}_{i',j'}(x',y')\). We prove for \(k + 1\). (i) Since \(\text{PACKET}_{i',j'}(x,y) = \text{PACKET}_{i',j'}(x',y')\), and it contains the destination node, then \(\text{PATH}_{k+1}(x,y) = \text{PATH}_{k+1}(x',y')\). Let \((p,q)\) be the player at position \(k+1\) in both communication paths. (ii) The packet that player \((p,q)\) sends depends only on the message it has received from \((i',j')\) (and it is the same in both cases), and on its input bits, which are \(x_p\) or \(x'_p\), and \(y_q\) or \(y'_q\). Since \(p \geq i\) and \(q \geq j\), and both inputs have the same \((m_1 - i + 1)\) and \((m_2 - j + 1)\) suffixes, then \(x_p = x'_p\) and \(y_q = y'_q\). Therefore player \((p,q)\) sends the same packet.

2.5.3 Lower bound for Reverse set disjointness

We now use the TMC model to prove the space lower bound for reverse set disjointness in the MDS model.
Theorem 2.2 (restated) For any \( n \geq 7 \), the space complexity of RDISJ\(_n\) in the MDS model is at least \( n - \log(n+1) - 3 \).

Proof. We will prove that any 2-dimensional TMC protocol computing the reverse set disjointness problem has a max communication cost of at least \( m = n - \log(n+1) - 3 \) bits. Using Lemma 2.3, this will imply the same lower bound in the MDS model.

To reach a contradiction, we assume there exists a protocol \( P \) that solves RDISJ\(_n\) with max communication of \( m \) bits, where \( m < n - \log(n+1) - 3 \). We will prove that there must be an input on which \( P \) errs.

According to Proposition 2.4, all communication paths go through the diagonal \( i + j = n \). There are \( 2^n \) different inputs \( (x, y) \), which means there are \( 2^n \) communication paths, while there are \( n+1 \) players in this diagonal, i.e., players of the form \((i, n-i)\). Therefore, by the pigeonhole principle, there exists \( 0 \leq i \leq n \), s.t. \(|\text{PASS}(i, n-i)| \geq \frac{2^n}{n+1}\). We call this player \((i, n-i)\) the congested player.

Claim 2.6. Let \((i, n-i)\) be the congested player of \( P \). Then \(|\text{PREF}(i, n-i)| \geq \frac{2^n}{n+1}\) \(\square\)

Proof. By Proposition 2.5, \(|\text{PREF}(i, n-i)| = \frac{|\text{PASS}(i, n-i)|}{2^n} \geq \frac{2^n}{n+1}\). Let \((x, y)\) be any input in \(\text{PASS}(i, n-i)\). Consider the \(i\)-th bit of \(x\) and the \((n-i)\)-th bit of \(y\). There are four possible settings for these bits, inducing a partition of \(\text{PREF}(i, n-i)\) into four sets. By the pigeonhole principle, one of these sets is of size at least \(\frac{2^n}{n+1}\). Call this set \(A\). Since the message sent by the congested player \((i, n-i)\) has at most \(m\) bits, where \(m < n - \log(n+1) - 3\), and since this player has only two neighbors, the number of possible packets it can send is less than \(\frac{2^n}{n+1}\). Since there are at least \(\frac{2^n}{n+1}\) different prefix pairs in \(A\), then by the pigeonhole principle, there exist two pairs of prefixes \((\alpha', \beta'), (\alpha'', \beta'')\) \(\in A\) s.t. \(\text{PACKET}_{i, n-i}(\alpha', \beta') = \text{PACKET}_{i, n-i}(\alpha'', \beta'')\), and \(\alpha'_i = \alpha''_i\), and \(\beta''_{n-i} = \beta''_{n-i}\).

The above prefix pairs \((\alpha', \beta')\) and \((\alpha'', \beta'')\) differ in at least one bit. W.l.o.g., we assume this is the \(k\)-th bit in \(\alpha'\) and \(\alpha''\), where \(k < i\). W.l.o.g., assume \(\alpha'_k = 1\) and \(\alpha''_k = 0\). We now define two different inputs for \(P\), on one of which \(P\) must err. Let \(\gamma = 0^{i-k}\) and \(\delta = 0^{i-k} \circ 1 \circ 0^{k-1}\).

Claim 2.7. \(P\) outputs the same answer on the two inputs: \((\alpha' \circ \gamma, \beta' \circ \delta)\) and \((\alpha'' \circ \gamma, \beta'' \circ \delta)\).

Proof. First examine the execution (and possible output) until player \((i, n-i)\) sends its packet. Since the length of \(\alpha'\) and \(\alpha''\) is \(i\), and the length of \(\beta'\) and \(\beta''\) is \(n-i\), then there is no \(j\) s.t. both \((\alpha' \circ \gamma)_j\) and \((\beta' \circ \delta)_j\) belong to the prefix pair \((\alpha', \beta')\) (the same goes for \(\alpha''\) and \(\beta''\)). This implies that the value of RDISJ\(_n\) on these two input pairs is not necessarily 1. In addition, recall that \(\alpha'_k = 1\), implying that the value of RDISJ\(_n\) on \((\alpha' \circ \gamma, \beta' \circ \delta)\) is not necessarily 0. Therefore, in order to determine RDISJ\(_n\)\((\alpha' \circ \gamma, \beta' \circ \delta)\), one needs to know \((\beta' \circ \delta)_k = \delta''_k\), which is unavailable yet. Hence the output bit on \((\alpha' \circ \gamma, \beta' \circ \delta)\) was not written yet.
Now recall that $\text{PACKET}_{i,n-1}(\alpha',\beta') = \text{PACKET}_{i,n-1}(\alpha''',\beta''')$, and that the $(n-i,i)$-suffix pairs (i.e., $(\alpha_i' \circ \gamma, \beta''_{n-i} \circ \delta)$) are the same for the two executions. Therefore, by Lemma 2.4, $P$ outputs the same value. \hfill \Box

Claim 2.8. $\text{RDISJ}_n(\alpha' \circ \gamma, \beta' \circ \delta) \neq \text{RDISJ}_n(\alpha'' \circ \gamma, \beta'' \circ \delta)$.

Proof. Since $\gamma$ is all zero, and the only '1' bit in $\delta$ is in position $(i+1-k)$, which is the $k$-th bit in $\delta^R$, then $\text{RDISJ}_n(\alpha' \circ \gamma, \beta' \circ \delta) = 1$ iff $\alpha_k'$ is 1. Similarly, $\text{RDISJ}_n(\alpha'' \circ \gamma, \beta'' \circ \delta) = 1$ iff $\alpha_k''$ is 1. Recall that $\alpha'$ and $\alpha''$ differ in the $k$-th bit. Therefore, only one of $\text{RDISJ}_n(\alpha' \circ \gamma, \beta' \circ \delta)$ and $\text{RDISJ}_n(\alpha'' \circ \gamma, \beta'' \circ \delta)$ equals 1. \hfill \Box

Claims 2.7 and 2.8 prove that there is no protocol that solves $\text{RDISJ}_n$ with less than $n - \log(n+1) - 3$ bits of space. This concludes the proof of Theorem 2.2. \hfill \Box

2.5.4 Lower bound for delayed intersection

Theorem 2.4 (restated) For any $n \geq 7$, the space complexity of $\text{DINT}_n$ in the MDS model is at least $n - \log(n+1) - 3$.

Proof. We will prove that any 2-dimensional TMC protocol computing the delayed intersection problem has a max communication cost of at least $m = n - \log(n+1) - 3$ bits. Using Lemma 2.3, this will imply the same lower bound in the MDS model.

To reach a contradiction, we assume there exists a protocol $P$ that solves $\text{DINT}_n$ with max communication of $m$ bits, where $m < n - \log(n+1) - 3$. We will prove that there must be an input on which $P$ errs.

According to Proposition 2.4, all communication paths go through the diagonal $i+j = n$. There are $2^{3n+1}$ different inputs $(s, t, u, v)$, which means there are $2^{3n+1}$ communication paths, while there are $n+1$ players in this diagonal, i.e., players of the form $(i, n-i)$. Therefore, by the pigeonhole principle, there exists $0 \leq i \leq n$, s.t. $|\text{PASS}(i, n-i)| \geq \frac{2^{3n+1}}{n+1}$. We call the player $(i, n-i)$ the congested player.

By Proposition 2.5, $|\text{PREFIX}(i, n-i)| = \frac{|\text{PASS}(i,n-i)|}{2^{m+1}} \geq \frac{n}{n+1}$ (note that here $m_1 = 2n, m_2 = n+1$). Now consider the $i$-th bit of $s$ and the $(n-i)$-th bit of $u$ in any input in $\text{PREFIX}(i, n-i)$. There are four possible settings for these bits, inducing a partition of $\text{PREFIX}(i, n-i)$ into four sets. By the pigeonhole principle, one of these sets is of size at least $\frac{2^{n-2}}{n+1}$. Call this set $A$. Since the message sent by the congested player $(i, n-i)$ has at most $m$ bits, where $m < n - \log(n+1) - 3$, and since this player has only two neighbors, the number of possible packets it can send is less than $\frac{2^{2n-3}}{n+1}$. There are $\frac{2^{n-2}}{n+1}$ different

---

*We claimed that the output bit was not written by the time player $(i, n-i)$ sends its packet only on $(\alpha' \circ \gamma, \beta' \circ \delta)$. Note that it is possible that $P$ on $(\alpha'' \circ \gamma, \beta'' \circ \delta)$ has already written the output bit before player $(i, n-i)$ sends its packet. However, since $P$ on the two inputs writes the same output after player $(i, n-i)$ sends its packet, and it has to write the output bit for $(\alpha' \circ \gamma, \beta' \circ \delta)$, it means that $P$ writes two output bits on $(\alpha'' \circ \gamma, \beta'' \circ \delta)$, and therefore obviously errs. Hence we can discard this case.*

34
prefix pairs in \( A \), therefore by the pigeonhole principle, there exist two prefix pairs \((\alpha', \beta'), (\alpha'', \beta'')\) in \( A \) such that \( \text{PACKET}_{i,n-i}(\alpha', \beta') = \text{PACKET}_{i,n-i}(\alpha'', \beta'') \), and \( \alpha'_i = \alpha''_i \), and \( \beta'_{n-i} = \beta''_{n-i} \).

We now define two different inputs for \( P \), on one of which \( P \) must err. The above prefix pairs \((\alpha', \beta')\) and \((\alpha'', \beta'')\) differ in at least one bit. There are two cases, based on the position of this bit:

- First assume this is the \( k \)-th bit in \( \alpha' \) and \( \alpha'' \), where \( k < n \). W.l.o.g., assume \( \alpha'_k = 1 \). We will show that \( P \) did not output the \( k \)-th bit of \( \text{DINT}_n \), denoted \( \text{DINT}_n[k] \), on \((\alpha', \beta')\) by the time player \((i, n-i)\) sends its packet. For every input \((s, t, u, v)\) s.t. \( \text{PREF}_i(s ot) = \alpha' \) and \( \text{PREF}_{n-i}(u ov) = \beta' \), \( \text{DINT}_n[k](s, t, u, v) = (s_k \land v) = (\alpha'_k \land v) = v \). Note that \( v \) is the \( n + 1 \)-st bit in \((u \circ v)\) and the length of \( \beta' \) is \( n - i \leq n \). Therefore, \( v \) is not a part of \( \beta' \), implying that after seeing \((\alpha', \beta')\), no algorithm can determine \( v = \text{DINT}_n[k](s, t, u, v) \).

Let \( \gamma \in \{0, 1\}^{2n-1} \), \( \delta \in \{0, 1\}^i \times \{1\} \). We next show that \( P \) must output the same answer for \( \text{DINT}_n[k] \) on the two inputs: \((\alpha' \circ \gamma, \beta' \circ \delta)\) and \((\alpha'' \circ \gamma, \beta'' \circ \delta)\). Recall that \( \text{PACKET}_{i,n-i}(\alpha', \beta') = \text{PACKET}_{i,n-i}(\alpha'', \beta'') \), and that the suffixes \( \alpha'_i \circ \gamma \) and \( \beta'_{n-i} \circ \delta \) are the same for the two executions. Therefore, by Lemma 2.4, \( P \) outputs the same value.

On the other hand, we show that \( \text{DINT}_n[k](\alpha' \circ \gamma, \beta' \circ \delta) \neq \text{DINT}_n[k](\alpha'' \circ \gamma, \beta'' \circ \delta) \). This would imply that \( P \) err on at least one of the inputs. Recall that \( \text{DINT}_n[k](\alpha' \circ \gamma, \beta' \circ \delta) = (\alpha'_k \land v) = \alpha'_k \), and similarly \( \text{DINT}_n[k](\alpha'' \circ \gamma, \beta'' \circ \delta) = \alpha''_k \). Since \( \alpha'_k \neq \alpha''_k \), then the corresponding value of \( \text{DINT}_n[k] \) is also different.

- The proof for the other case, where the two prefix pairs \((\alpha', \beta')\) and \((\alpha'', \beta'')\) differ in the \( k \)-th bit in \( \beta' \) and \( \beta'' \), is very similar. Either \( \beta'_k = 1 \) or \( \beta''_k = 1 \). For example, assume the former. The value of \( \text{DINT}_n[n+k] \) on inputs whose prefix pair is \((\alpha', \beta')\) depends on the \( k \)-th bit of \( t \), which is not available at the time the algorithm reads \((\alpha', \beta')\). Hence, the protocol \( P \) can not output \( \text{DINT}_n[n+k] \) on \((\alpha', \beta')\) by the time player \((i, n-i)\) sends its packet.

We choose a suffix pair \((\gamma, \delta)\) for the two prefixes, such that the \( k \)-th bit in \( t \), i.e., the \( n - i + k \)-th bit in \( \gamma \), is set. \( \text{DINT}_n[n+k] \) is different on the two inputs \((\alpha' \circ \gamma, \beta' \circ \delta)\) and \((\alpha'' \circ \gamma, \beta'' \circ \delta)\).

\[ \square \]

### 2.5.5 Lower bound for tensor product

We start with a proof of the lower bound for the tensor product of two vectors:

**Theorem 2.6 (restated)** The space complexity of computing \( x \otimes y \) (\( x \) and \( y \) are of lengths \( m \) and \( n \), respectively) in the MDS model is at least \( \min(m, n) - 3 \) bits.
Proof. We prove the lower bound in the TMC model. The corresponding lower bound in the MDS model will then follow from the reduction lemma (Lemma 2.3). To reach a contradiction, we assume there exists a protocol \( P \) that computes \( x \otimes y \) with max communication of \( s \) bits, where \( s < \min(m,n) - 3 \). We shall prove that there exists an input \((x,y)\) on which \( P \) must err.

We consider only inputs in which the last bit is set. The number of inputs is therefore \( 2^{m+n-2} \). For every input \((x,y)\), the communication path goes either through player \((m,n-1)\) or through \((m-1,n)\). This induces a partition of the inputs into two sets: \( \text{PASS}(m,n-1) \), \( \text{PASS}(m-1,n) \). By the pigeonhole principle, one of these sets is of size at least \( \frac{2^{m+n-2}}{2} \). W.l.o.g., we assume it is \( \text{PASS}(m,n-1) \). There are \( 2^n \) possible values for \( y \), which induces a partition of \( \text{PASS}(m,n-1) \) into \( 2^n \) sets. By the pigeonhole principle, one of these sets is of size at least \( \frac{2^{m+n-2}}{2^n} \). Call this set \( A \). Since the message sent by player \((m,n-1)\) has at most \( s \) bits, where \( s < \min(m,n) - 3 \), then by the pigeonhole principle, there exist two pairs of inputs \((x',y),(x'',y)\) \( \in A \) s.t. \( \text{PACKET}_{m,n-1}(x',y) = \text{PACKET}_{m,n-1}(x'',y) \). \( x' \) and \( x'' \) differ in at least one bit, let it be the \( k \)-th bit, \( k < m \).

Since \( y_n \) has not been read yet, the results of \((x \otimes y)_{s,n}\) could not have been written to the output stream yet. Recall that \( x'_m = x''_m = 1 \), \( y_n = 1 \), and \( \text{PACKET}_{m,n-1}(x',y) = \text{PACKET}_{m,n-1}(x'',y) \). Therefore, player \((m,n)\) writes the same output for the two inputs: \((x',y)\) and \((x'',y)\). But since \( x'_k \neq x''_k \), then \( (x' \otimes y)_{k,n} \neq (x'' \otimes y)_{k,n} \).

This proves that there is no algorithm that computes \( x \otimes y \) with less than \( \min(m,n) - 3 \) bits of space. \( \Box \)

We now extend the proof to deal with \( k \geq 2 \) vectors:

**Theorem 2.7 (restated)** Given \( k \) vectors \( x_1,x_2,\ldots,x_k \) of dimensions \( m_1,\ldots,m_k \), respectively, the space complexity of computing \( x_1 \otimes x_2 \otimes \cdots \otimes x_k \) in the MDS model is at least \( \sum_{i=1}^{k} m_i - \max_i \{m_i\} - \log k - k + 1 \).

**Proof.** We give a proof sketch, since it is an extension of the above proof for two vectors. We assume there exists a protocol \( P \) that computes \( x_1 \otimes \cdots \otimes x_k \) with max communication of \( s \) bits, where \( s < \sum_{i=2}^{k} m_i - \log k - k + 1 \). We shall prove that there must be an input on which \( P \) errs.

We consider only input vectors in which the last bit is set \((2^{\sum_{i=1}^{k}(m_i-1)}) \) inputs). The last position of the communication path, before reaching player \((m_1,\ldots,m_k)\), induces a partition of the inputs into \( k \) sets. One of these sets is of size at least \( \frac{2^{\sum_{i=1}^{k}(m_i-1)}}{k} \). W.l.o.g., we assume it is \( \text{PASS}(m_1-1,m_2,\ldots,m_k) \).

There are \( 2^{m_1-1} \) possible values for \( x_1 \), which induces another partition, in which one of the sets is of size at least \( \frac{2^{x_1} \sum_{i=2}^{k}(m_i-1)}}{k} \). Call this set \( A \). By the pigeonhole principle, there exist two inputs \( I_1 = (x_1,x'_2,\cdots,x'_k) \) and \( I_2 = (x_1,x''_2,\cdots,x''_k) \) in \( A \), s.t. the packet sent by player \((m_1-1,m_2,\cdots,m_k)\) is the same on these inputs. Therefore, the last player writes the same output for the two inputs, but since \( I_1 \) and \( I_2 \) differ in at least one bit, then their tensor products also differ in the corresponding entry. \( \Box \)
2.6 Algorithms

In this section we present our upper bounds for evaluating twig queries, for the three evaluation modes.

2.6.1 The filtering algorithm

We now present a constant space\(^9\) filtering algorithm for queries that do not contain the (//)(/) pattern. The algorithm, presented in Algorithm 2.1, computes \(\text{FILTER}_Q(D)\) by trying to find a match of \(Q\) in \(D\). The basic procedure used in the algorithm is \(\text{NextMatchUnderSelf}(u, e_u)\), which gets as input a query node \(u\) and the element \(e_u\), on which the cursor of the stream \(T_u\) is currently positioned\(^10\), and returns true if and only if the sub-query \(Q_u\) has a match in the sub-document \(D_{e_u}\). Moreover, if such a match exists, the procedure advances the stream cursors to the positions that indicate the match.

NextMatchUnderSelf works by recursively searching for matches of the sub-queries rooted at the children \(v\) of \(u\). To this end, it calls the procedure \(\text{NextMatchUnderParent}(v, e_u)\). The latter gets as input a query node \(v\) and the element \(e_u\), on which the cursor of \(T_u\) (\(u = \text{parent}(v)\)) is currently positioned, and returns true if and only if \(Q_v\) has a match in \(D_{e_v}\), where \(e_v\) is a descendant of \(e_u\) whose relationship with \(e_u\) matches the axis of \(v\)\(^11\). This procedure works by repeatedly advancing the cursor of \(T_v\), until finding the desired element \(e_v\). If a match is found, the cursors of the corresponding streams are advanced to positions that indicate the match.

The axis pattern of \(Q\) allows the algorithm to decide locally whether an element participates in a match or not, without having to remember elements for later use. The intuition is that a child-axis node can be matched only to elements of the same level, since all of its query-ancestors are also child-axis nodes. Therefore recursions in the document, which are the cause of difficulty in the pattern //a/b, are irrelevant for this case.

2.6.1.1 Example run

In order to illustrate how the filtering algorithm works, we provide an example run. Consider the document and the query presented in Figure 2.7. A subscript is added to each node to indicate its label. Another subscript, which is added to document elements only, indicates the order in the index stream. Initially, the three cursors point to \((e_{a_1}, e_{b_1}, e_{c_1})\). \(\text{NextMatchUnderSelf}(u_8, e_8)\) calls \(\text{NextMatchUnderParent}(u_a, e_8)\), which searches in the subtree of \(e_8\) for an \(a\) element that has \(b\) and \(c\) descendants. \(e_{a_1}\) is the first element checked. Now \(T_b\) and \(T_c\) are advanced separately, until the cursors point to \(b\) and \(c\) elements that are descendants of \(e_{a_1}\), or begin after \(e_{a_1}\) ends. Since \(e_{b_1}\) is not nested

\(^9\)The constant space is up to the \(\tilde{O}\)-notation, which suppresses factors that are linear in the query size or logarithmic in the document size.

\(^10\)As mentioned in Section 2.3, we use \(T_u\) to denote the cursor on the stream corresponding to \(u\). This way, if two query nodes \(u, v\) share the same label, then \(T_u\) and \(T_v\) denote two separate cursors on the same stream.

\(^11\)Note that the relationship between two elements can be easily tested using their BEL encoding.
Algorithm 2.1 Filtering algorithm for queries without \((/)(/))\).

1: \textbf{function} \textsc{Filter}(Q, D) \\
2: \hspace{1em} \textbf{return} \text{NextMatchUnderSelf}(\text{root}(Q), \text{root}(D)) \\
3: \textbf{end function} \\
1: \textbf{function} \textsc{NextMatchUnderSelf}(u, e_u) \\
2: \hspace{1em} \textbf{for} every child \(v\) of \(u\) \textbf{do} \\
3: \hspace{2em} \textbf{if} \(!\text{NextMatchUnderParent}(v, e_u)\) \textbf{then} \\
4: \hspace{3em} \textbf{return} false \\
5: \hspace{2em} \textbf{end if} \\
6: \hspace{1em} \textbf{end for} \\
7: \hspace{1em} \textbf{return} true \\
8: \textbf{end function} \\
1: \textbf{function} \textsc{NextMatchUnderParent}(v, e_v) \\
2: \hspace{1em} e_v := T_v.\text{ReadElement}() \\
3: \hspace{1em} \textbf{while} \((e_v \neq T_v.\text{EndOfStream}) \text{ and } (e_v.\text{Begin} < e_u.\text{End})\) \textbf{do} \\
4: \hspace{2em} \textbf{if} \((\text{relationship between } e_v \text{ and } e_u \text{ matches axis}(v)) \text{ and } \) \\
5: \hspace{3em} \text{(\text{NextMatchUnderSelf}(v, e_v))}\textbf{then} \\
6: \hspace{4em} \textbf{return} true \\
7: \hspace{1em} \textbf{end if} \\
8: \hspace{1em} T_v.\text{Advance}() \\
9: \hspace{1em} e_v := T_v.\text{ReadElement}() \\
10: \hspace{1em} \textbf{end while} \\
11: \hspace{1em} \textbf{return} false \\
12: \textbf{end function}

despite \(e_{a_1}\), \(T_b\) is advanced to \(e_{b_2}\), which matches the required axis. However, \(e_{c_1}\) begins after \(e_{a_1}\) ends, and therefore \(e_{a_1}\) is rejected as a possible match to \(u_a\), and \(T_a\) is advanced to \(e_{a_2}\). Now the three cursors point to \((e_{a_2}, e_{b_2}, e_{c_1})\). Again, we look for \(b\) and \(c\) descendants of \(e_{a_2}\). \(e_{b_2}\) is not nested within \(e_{a_2}\), and \(T_b\) is advanced to \(e_{b_3}\), which matches the required axis. \(e_{c_1}\) is already a descendant of \(e_{a_2}\). Therefore both calls to \textsc{NextMatchUnderParent}(\(u_b, e_{a_2}\)) and \textsc{NextMatchUnderParent}(\(u_c, e_{a_2}\)) return true, which means that \textsc{NextMatchUnderParent}(\(u_a, e_S\)) returns true. \textsc{Filter}(Q) returns true and the three cursors point to \((e_{a_2}, e_{b_3}, e_{c_1})\), which indicate the match found.

![Figure 2.7: An example XML document (right) and query (left).](image-url)
2.6.1.2 Complexity analysis

Space complexity:

**Proposition 2.6.** Let $Q$ be a basic twig query that does not contain the $(//)(/)$ pattern. Then, the space complexity of the algorithm $\text{Filter}(Q,D)$ is $\tilde{O}(1)$.

**Proof.** The recursion depth of the algorithm is equal to the query depth. Each level requires space for storing $O(1)$ document elements. Therefore, the space complexity is $\tilde{O}(1)$. \hfill $\Box$

**Time and I/O complexity:** The time and I/O complexity of the algorithm are both linear in the length of the input streams. Note that the I/O complexity may be substantially decreased by using B+ trees over the index streams, and “ForwardTo(pos)” methods in the $\text{NextMatchUnderParent}$ method, similarly to previous algorithms (e.g., see [48]). However, since the focus of this research is the memory requirements, and it is quite simple to integrate these I/O saving methods, we present here the simplified version.

2.6.1.3 Correctness analysis

We now provide a full analysis of the algorithm’s correctness:

**Theorem 2.9.** Let $Q$ be any basic twig query, which does not contain the $(//)(/)$ pattern. Then, the algorithm $\text{Filter}(Q,D)$ returns true if and only if there exists at least one match of $Q$ in $D$.

In order to prove Theorem 2.9, we need to show that the algorithm is both sound (returns true, only if a match exists) and complete (if a match exists, returns true).

**Cursor configurations** A notion that will play a crucial role in our analysis is cursor configurations. We denote by $Q$ and $D$ any basic twig query and any document, respectively. Let $u$ be a query node. A $Q_u$-cursor configuration (or $Q_u$-configuration, in short) is a setting of the cursors $\{T_v\}_{v \in Q_u}$. For a $Q_u$-configuration $C$ and for a node $v \in Q_u$, $C[v]$ denotes the position of the cursor $T_v$ as specified by $C$. We sometimes abuse notation and think of $C[v]$ as the document element pointed by this cursor.

A $Q_u$-configuration can be viewed as a mapping from $Q_u$ to elements of $D$ that preserves label matches. If this mapping is a match, we say that the configuration induces a match.

Let $C_1, C_2$ be two $Q_u$-configurations. $C_1$ is said to dominate $C_2$, denoted $C_1 \succeq C_2$, if for every $v \in Q_u$, $C_1[v] \geq C_2[v]$. As stream cursors move only in the forward direction, subsequent configurations encountered during an execution of an algorithm always dominate one another.

The $Q_u$-configuration at the time a function $f$ is called is the starting $Q_u$-configuration of $f$. The $Q_u$-configuration when $f$ returns is called the ending $Q_u$-configuration of $f$. By the above, the ending configuration always dominates the starting configuration.

---

12We slightly abuse notation, and use $T_v$ to denote both the stream $T_{\text{label}(v)}$ and the cursor on this stream corresponding to $v$.  

39
We next analyze the main subroutine, NextMatchUnderSelf, and show it is sound and complete. It is easy to verify that whenever NextMatchUnderSelf($u$, $e_u$) is called, then $e_u$ is the element on which the cursor $T_u$ is currently positioned.

**Soundness** The soundness of Filter($Q$, $D$) will follow from the soundness of the function NextMatchUnderSelf:

**Lemma 2.5** (Soundness). If NextMatchUnderSelf($u$, $e_u$) returns true, then its ending $Q_u$-configuration induces a match of $Q_u$ in $D_{e_u}$.

**Proof.** We prove the lemma by induction on $k = \text{height}(u)$. The base case, $k = 0$, corresponds to a leaf $u$. In this case the function always returns true, and indeed the mapping $u \mapsto e_u$ is a trivial match of $Q_u$ in $D_{e_u}$. The ending $Q_u$-configuration equals in this case the starting $Q_u$-configuration. The latter induces the trivial match, by assumption.

Assume that the lemma holds for all nodes of height at most $k$. Let $u$ be a node of height $k + 1$. Since the function returns true, then also the calls to NextMatchUnderParent($v$, $e_u$), for every child $v$ of $u$, return true. Consider one such child $v$.

Since NextMatchUnderParent($v$, $e_u$) returns true, the function must have found an element $e_v$ that satisfies the following: (1) the structural relationship between $e_v$ and $e_u$ matches $\text{axis}(v)$; and (2) the function NextMatchUnderSelf($v$, $e_v$) returns true. By the induction hypothesis, the latter implies that there is a match $\phi_v$ of $Q_v$ in $D_{e_v}$ and that the ending $Q_v$-configuration of the call to NextMatchUnderSelf($v$, $e_v$) induces this match.

We can now define a match $\phi_u$ of $Q_u$ in $D_{e_u}$ as follows: (1) $\phi_u(u) = e_u$; (2) for every child $v$ of $u$ and for every node $w \in Q_v$, $\phi_u(w) = \phi_v(w)$. As each node in $Q$ has its own separate cursor, then the ending $Q_u$-configuration of NextMatchUnderSelf($u$, $e_u$) consists of the ending $Q_v$-configurations of NextMatchUnderParent($v$, $e_u$), for each child $v$ of $u$. The latter induce the matchings $\{\phi_v\}_v$ is a child of $u$ and therefore the ending $Q_u$-configuration induces the matching $\phi_u$. $\square$

**Completeness** The completeness of Filter($Q$, $D$) follows from the following lemma:

**Lemma 2.6** (Completeness). Suppose $Q$ does not contain the (//)(/) pattern and let $C$ be the starting $Q_u$-configuration of NextMatchUnderSelf($u$, $e_u$). If there exists a $Q_u$-configuration $C' \succeq C$ that induces a match of $Q_u$ in $D_{e_u}$, then NextMatchUnderSelf($u$, $e_u$) returns true.

We assume from now on that $Q$ does not contain the (//)(/) pattern. The following propositions are a key to proving the completeness lemma:

**Proposition 2.7.** Let $\phi$ be a match of $Q$ in $D$. Then, for every child-axis node $u$, $\text{depth}(\phi(u)) = \text{depth}(u)$.

The proof is straightforward by induction on $\text{depth}(u)$.

40
**Proposition 2.8.** If during the execution of Filter\((Q, D)\), NextMatchUnderSelf\((u, e_u)\) is called with a child-axis node \(u\), then \(\text{depth}(e_u) = \text{depth}(u)\).

**Proof.** We prove by induction on \(k = \text{depth}(u)\). The base case, \(k = 0\), corresponds to the query root. In this case the function NextMatchUnderSelf\((u, e_u)\) is only called with the document root as its second parameter, and indeed its depth is 0.

Assume that the lemma holds for all nodes of depth at most \(k\). Let \(u\) be a node of depth \(k + 1\). The function NextMatchUnderSelf\((u, e_u)\) can only be called from NextMatchUnderParent\((u, e_v)\), where \(e_v \in T_v\) and \(v = \text{parent}(u)\) in \(Q\), and only if \(e_u\) is a child of \(e_v\). \(u\) is a child-axis node, therefore all of its ancestors are too (by the definition of \(Q\)). Since NextMatchUnderParent\((u, e_v)\) can only be called from NextMatchUnderSelf\((v, e_v)\), then by the induction hypothesis, \(\text{depth}(e_v) = \text{depth}(v) = k\), which proves that \(\text{depth}(e_u) = k + 1\) (because \(e_u\) is a child of \(e_v\)).

\(\square\)

**Proof of Lemma 2.6.** We prove the lemma by induction on \(k = \text{height}(u)\). For \(k = 0\), \(u\) is a leaf. In this case the function always returns true.

Suppose that the lemma holds for all nodes of height at most \(k\). Consider a node \(u\) of height \(k + 1\). NextMatchUnderSelf\((u, e_u)\) returns true only if the calls to NextMatchUnderParent\((v, e_u)\), for each child \(v\) of \(u\), return true. Consider such a child \(v\), then.

Let \(C_v\) be the restriction of \(C\) to \(Q_v\). Note that \(C_v\) is the starting \(Q_v\)-configuration of NextMatchUnderParent\((v, e_u)\), even if \(v\) is not the first child to be processed. This is because a call to NextMatchUnderParent\((v, e_u)\), for any other child \(v'\) of \(u\), cannot change cursors corresponding to nodes in \(Q_v\).

Let \(C'_v\) be the restriction of \(C'\) to \(Q_v\). \(C'_v\) induces a match of \(Q_v\) in \(D_{e'_u}\), where \(e'_u = C'[v]\). Note that \(e'_u \in D_{e_u}\) and its structural relationship with \(e_u\) matches axis\((v)\).

Since \(C'\) dominates \(C\), then also \(C'_v\) dominates \(C_v\). It follows that \(C_v[v]\) precedes (or equals) \(e'_v\) in the stream \(T_v\). When calling NextMatchUnderParent\((v, e_u)\), the function enumerates the elements \(e_v\) on the stream \(T_v\), starting with \(C_v[v]\). We next show that the enumeration has to stop either at \(e'_v\) or before in success.

If the enumeration stops at some \(e_v\) that precedes \(e'_v\), then NextMatchUnderParent\((v, e_u)\) returns true. So suppose the enumeration has not stopped at any of these nodes. We would like to show it must stop at \(e'_v\).

**Claim 2.9.** Let \(C''_v\) be the \(Q_v\)-configuration when the algorithm starts processing \(e'_v\), i.e., when NextMatchUnderParent\((v, e_u)\) reaches line 3 and the cursor \(T_v\) points to \(e'_v\). Then, \(C''_v \preceq C'_v\).

Before we prove this claim, let us use it to conclude the proof of Lemma 2.6. Since \(C''_v[v] = e'_v\) and \(C''_v\) is dominated by \(C'_v\), which induces a match of \(Q_v\), then by the induction hypothesis, the function NextMatchUnderSelf\((v, e'_v)\) returns true, and so does its calling function NextMatchUnderParent\((v,
We conclude that NextMatchUnderParent(v, eu) returns true for all children v of u, and thus also NextMatchUnderSelf(u, eu) returns true.

**Proof of Claim 2.9.** Suppose, to reach a contradiction, that Cv'' is not dominated by Cv'. This implies that there exists a node b ∈ Q_v s.t. Cv''[b] > Cv'[b]. If there is more than one such node, we choose b to be the node, for which Tb is the first to be advanced beyond Cv'[b]. Let a be the parent of b in Q. Tb must be advanced beyond Cv'[b] during the execution of NextMatchUnderParent(b, e''_a), where e''_a is some node in the stream Ta. Note that at the time Tb is advanced beyond Cv'[b], the cursor Ta points to e''_a. By the choice of b, the position of e''_a in the stream Ta is at most Cv'[a], i.e., e''_a.End ≤ Cv'[a].Begin.

There are two possible positions for e''_a in the document: (1) e''_a.End < Cv'[a].Begin, or (2) e''_a is an ancestor of (or equals) Cv'[a]. We prove that both lead to a contradiction.

Consider the first option, i.e., e''_a.End < Cv'[a].Begin. Cv'[b] is nested within Cv'[a] since C' induces a match. Therefore, e''_a.End < Cv'[b].Begin, which means that the condition of the while loop in NextMatchUnderParent(b, e''_a) is not satisfied, and the function could not advance Tb beyond Cv'[b], in contradiction to our assumption.

Consider then the second option, i.e., e''_a is an ancestor of (or equals) Cv'[a]. There are three subcases here. (i) a is a child-axis node and e''_a ≠ Cv'[a], (ii) a is a descendant-axis node, or (iii) e''_a = Cv'[a]. In case (i), e''_a is an ancestor of Cv'[a] and therefore depth(e''_a) < depth(Cv'[a]). Since C' induces a match, then according to Proposition 2.7, depth(Cv'[a]) = depth(a), and thus depth(e''_a) < depth(a). Therefore, based on Proposition 2.8, there is no call to NextMatchUnderSelf(a,e''_a), which means there is no call to NextMatchUnderParent(b,e''_a), in contradiction to the assumption.

To deal with cases (ii) and (iii), we first show that in both of them the relationship between Cv'[b] and e''_a matches axis(b). In case (ii), a is a descendant-axis node. Hence, also b must be a descendant-axis node (Q does not have the (l/l)(l) pattern), and since C' induces a match, then Cv'[a] is an ancestor of Cv'[b]. In addition, recall that e''_a is an ancestor of (or equals) Cv'[a]. Therefore, Cv'[b] is a descendant of e''_a, and thus the relationship between Cv'[b] and e''_a matches axis(b). In case (iii), e''_a = Cv'[a], and thus the relationship between Cv'[b] and e''_a matches axis(b), because C' induces a match.

Now consider the Q_b-configuration when NextMatchUnderParent(b, e''_a) starts processing Cv'[b], i.e., when it reaches line 3 and the cursor Tb points to Cv'[b]. The relationship condition in line 4 is satisfied, therefore the function calls NextMatchUnderSelf(b,Cv'[b]). Since we assumed b is the node in Q_v whose cursor is the first to move beyond Cv'[b], then the current Q_b-configuration is dominated by C_v (the restriction of C' to Q_b). By the induction hypothesis, NextMatchUnderSelf(b,Cv'[b]) returns true. NextMatchUnderParent(b,e''_a) would also return true, without advancing Tb, in contradiction to our assumption.

□
2.6.2 The full-fledged evaluation algorithm

In this section we extend the filtering algorithm we presented in Section 2.6.1, which evaluates queries that do not contain the \((//)(/)\) pattern, into a full-fledged evaluation algorithm. The algorithm, presented in Algorithm 2.2, makes use of the NextMatchUnderSelf procedure from the filtering algorithm. The basic procedure of the algorithm is \(\text{Eval}(Q, t, D)\), which gets as input a query tree \(Q\), its output node \(t\), and a document \(D\), and works by iteratively looking for a match of \(Q\) in \(D\). For each match found, it: (i) outputs the document element \(e_t\) that \(t\) is mapped to by this match, and (ii) advances the cursor beyond \(e_t\).

\textbf{Algorithm 2.2} FFE algorithm for queries without \((//)(/)\).

\begin{verbatim}
1: procedure \text{Eval}(Q, t, D)
2: while (NextMatchUnderSelf(root(Q), root(D))) do
3:   output \(T_t.\text{ReadElement}()\)
4:   \(T_t.\text{Advance}()\)
5: end while
6: end procedure
\end{verbatim}

2.6.2.1 Complexity analysis

\textbf{Space complexity:}

\textbf{Proposition 2.9.} Let \(Q\) be a basic twig query that does not contain the \((//)(/)\) pattern. Then, the space complexity of the algorithm \text{Eval} is \(\tilde{O}(1)\).

\textbf{Proof.} The function \text{Eval} is iteratively calling \text{NextMatchUnderSelf}, which uses constant space (see Proposition 2.6). Therefore, the space complexity of \text{Eval} is \(\tilde{O}(1)\). \qed

\textbf{Time and I/O complexity:} The time and I/O complexity of the algorithm are both linear in the length of the input streams. As we noted in Section 2.6.1.2, the I/O complexity may be easily decreased.

2.6.2.2 Correctness analysis

In order to prove the correctness of the algorithm \text{Eval}, we need to show it is \textit{sound} (every element it outputs indeed matches \(t\)) and \textit{complete} (every element that matches \(t\) node is output).

\textbf{Soundness} To prove soundness, let \(e_t\) be an element that \text{Eval} outputs. \(e_t\) must have been the element pointed by the cursor \(T_t\) after the function \text{NextMatchUnderSelf} returned true. By Lemma 2.5, the ending configuration of \text{NextMatchUnderSelf} (if it returns true) induces a match \(\phi\) of \(Q\) in \(D\). Therefore, \(e_t = \phi(t)\) indeed matches \(t\).

\textbf{Completeness} Let \(e_{t_1}, e_{t_2}, \ldots, e_{t_k}\) be the elements that match \(t\), in document order. We prove that \text{Eval} outputs them in this order.
Proof sketch. We show that the $i$-th call to NextMatchUnderSelf in line 2 of Eval advances the cursor configuration to the “minimum” match $\phi$, for which $\phi(t) \geq e_t$. Begin. Here, the “minimum” is w.r.t. the partial order induced by the domination relation, and the existence of the minimum is guaranteed by the fact $Q$ does not have the $(//)(//)$ pattern. Since we always move to the minimum match, we are guaranteed not to miss a match of $t$ with one of the $e_t$'s.

We now examine the execution of Eval($Q,t,D$), which consists of a sequence of calls to NextMatchUnderSelf. Let $C^s_i$ and $C^e_i$ denote the starting and ending configuration of the $i$-th call, respectively. Note that $\forall v \in Q$, if $v \neq t$ then $C^s_{i+1}[v] = C^e_i[v]$, and for $t$: $C^s_{i+1}[t] = C^e_i[t] + 1$.

Since every match $m$ induces a configuration, we sometimes abuse notation and think of $m$ as the induced configuration. Let $M_i$ denote the set of all matches $m$ of $Q$ in $D$, for which $m[t] \geq e_t$. Recall that domination induces a partial order over the space of configurations [matches]. Therefore, not every set of configurations necessarily has a minimum (i.e., a configuration that is dominated by all other configurations in the set). The following lemma describes a scenario, where such a minimum always exists. The lemma will be used at the core of the completeness arguments below:

Lemma 2.7. Suppose $Q$ is a basic twig query, which does not contain the $(//)(//)$ pattern. Then $\forall 1 \leq i \leq k$, $M_i$ has a minimum, denoted $m_i$.

Proof. In order to show $M_i$ has a minimum, it suffices to prove that $M_i$ has only one match that does not dominate any other match in $M_i$. Suppose, to reach a contradiction, $M_i$ has two such matches, $m_1$ and $m_2$. We use $m_1$ and $m_2$ to define a new match $m_3$. For each $v \in Q$, $m_3[v] = \min\{m_1[v], m_2[v]\}$. Since $m_1, m_2 \in M_i$, i.e., $m_1[t], m_2[t] \geq e_t$, then also $m_3 \in M_i$. We would like to show that $m_3$ induces a match.

For each $v \in Q$, let $m^v_3$ denote the restriction of $m_3$ to $Q_v$. We prove by induction on height($v$), that for every such $v$, $m^v_3$ induces a match. The base case, height($v$) = 0, corresponds to leaves. In this case, $m_3^v$ consists of the single element $m_3[v]$, which by definition belongs to $T_v$, and thus matches $v$.

Assume, then, that for every node $w$ of height $k$, $m^w_3$ induces a match of $Q_w$. Let $v$ be a node of height $k + 1$. By the induction hypothesis, for every child $w$ of $v$, $m^w_3$ induces a match of $Q_w$. In order for $m^v_3$ to induce a match, we need to make sure that the structural relationship between $m_3[w]$ and $m_3[v]$ matches axis($w$), for every child $w$ of $v$.

Fix one such child $w$. If $m_3[w] = m_1[w]$ and $m_3[v] = m_1[v]$, then since $m_1$ induces a match, the structural relationship between $m_3[w]$ and $m_3[v]$ matches axis($w$). The same thing happens if $m_3[w] = m_2[w]$ and $m_3[v] = m_2[v]$, due to the fact $m_2$ induces a match. Assume, then, that $m_3[w] = m_1[w] < m_2[w]$ and $m_3[v] = m_2[v] < m_1[v]$ (the opposite case is handled analogously). We therefore have the following situation: $m_2[v]$ precedes $m_1[v]$ in pre-order traversal. However, the child or descendant $m_2[w]$ of $m_2[v]$ succeeds the child or descendant $m_1[w]$ of $m_1[v]$. This can happen only if $m_1[v]$ is a
descendant of $m_2[v]$. We now split to two cases, based on the axis of $w$.

- **Case 1:** $\text{axis}(w) = /$. In this case $m_1[w]$ is a descendant of $m_1[v]$ (since $m_1$ induces a match), which in turn is a descendant of $m_2[v]$ (by the above). Therefore, $m_3[w]$ is a descendant of $m_3[v]$, as desired.

- **Case 2:** $\text{axis}(w) = //$. In this case we have an element and its descendant, namely $m_2[v]$ and $m_1[v]$, both of which match the same query node $v$. This can happen only if $v$ or one of its ancestors has a descendant axis. However, since $w$ has a child axis and is a child of $v$, neither $v$ nor its ancestors are allowed to have a descendant axis (recall that in our query no $(//)$ is followed by a $(/)$). Therefore, this case simply cannot happen.

We conclude that $m_3 \in M_i$ induces a match. By definition, both $m_1$ and $m_2$ dominate $m_3$, in contradiction to our assumption that neither of them dominates any other configuration in $M_i$. The lemma follows.

**Lemma 2.8.** $m_i[t] = e_{t_i}$.

**Proof.** Recall that for each $m \in M_i$, $m_i[t] \geq e_{t_i}$. In addition, $e_{t_i} \in \text{FFE}_Q(D)$, therefore $\exists m \in M_i$, s.t. $m[t] = e_{t_i}$. Since $m_i$ is the minimum, then $m_i[t] = e_{t_i}$. ☐

**Lemma 2.9.** Let $1 \leq i \leq k$. If $C_i[t] > e_{t_{i-1}}$ and $M_i \succeq C_i^\circ$, then $C_i^\circ = m_i$.

**Proof.** Let us examine the starting configuration of the $i$-th call to NextMatchUnderSelf, i.e., $C_i^\circ$, and the set $S$ of matches dominating it. By the lemma, $M_i \succeq C_i^\circ$. In addition, $C_i^\circ[t] > e_{t_{i-1}}$, therefore for any match $m$ dominating $C_i^\circ$, $m \in M_i$. Thus $S = M_i$. The next property of NextMatchUnderSelf shows that its ending configuration is exactly $m_i$. ☐

**Claim 2.10.** Suppose that we call NextMatchUnderSelf($u, e_u$) with a starting $Q_u$-configuration $C$ with $C[u] = e_u$. Let $M_C$ denote the set of $Q_u$-configurations that: (1) dominate $C$; (2) induce a match of $Q_u$ in $D_{e_u}$. If $M_C$ has a minimum, then NextMatchUnderSelf($u, e_u$) returns true and its ending $Q_u$-configuration is the minimum of $M_C$.

**Proof.** By the completeness property of NextMatchUnderSelf (Lemma 2.6), we know that if $M_C$ is not empty, then the function returns true. By Lemma 2.5, the ending configuration induces a match. We only need to prove that this match $m$ is the minimum of $M_C$. Suppose, to reach a contradiction, that there is a different match $m' \in M_C$, which is the minimum. Now assume the following setting: we "end" all the streams right after the positions of $m'$, and call NextMatchUnderSelf with the same starting configuration $C$. In this case $M_C = \{m'\}$, and by the correctness of NextMatchUnderSelf, the ending configuration is $m'$. Since the algorithm reads the streams sequentially, then until reaching
configuration $m'$, it can not differentiate between the two settings, and therefore in our original setting the ending configuration would also be $m'$ and not $m$.

Lemma 2.10. For each $i \in [k]$: (\text{*}) $C_s^e[t] > e_{t_i-1}$ and (\text{**}) $M_i \succeq C_s^e$.

Proof. We prove by induction on $i$. The base case corresponds to the first call to NextMatchUnderSelf, when the starting configuration is the beginning of all streams. In this case both (\text{*}) and (\text{**}) trivially hold. Assume the lemma holds for every $i \leq j$. By the induction hypothesis, $C_s^e[t] > e_{t_j-1}$ and $M_j \succeq C_s^e$. By Lemma 2.9, $C_s^e[j] = m_j$. Note that before the $(j+1)$-th call only $T_t$ is advanced, and $m_j[t] = e_{t_j}$ (Lemma 2.8), therefore $C_s^e[j+1[t] > e_{t_j}$ (\text{*}). $m_j \leq M_{j+1}$, since $m_j$ is the minimum of $M_j$ and $M_{j+1} \subseteq M_j$. Note that $m_j$ is identical to $C_s^e[j+1$ except for the position of $T_t$, which was advanced only by one element. It follows that $C_s^e[j+1 \succeq M_{j+1}$ (\text{**}).

Corollary 2.2. For each $i \in [k]$: $C_s^e = m_i$.

The above corollary and Lemma 2.8 prove that for each $i \in [k]$: $C_s^e[t] = e_{t_i}$, which means that the $i$-th call to NextMatchUnderSelf outputs $e_{t_i}$. Thus the ffe algorithm outputs all $e_{t_i} \in \text{FFE}_{Q}(D)$, and this concludes the proof of completeness.

2.6.3 Pattern matching upper bound

In this section we present an upper bound for computing all the matches of basic twig queries.

We observe that existing algorithms (TwigStack [14] and TwigStackList [57]) use in some cases much more space than indicated by the DOE lower bound. When the query contains only descendant axes, both algorithms keep only elements that are guaranteed to be in at least one match. However, when child axis nodes are involved, the algorithms may keep many redundant intermediate results. To demonstrate this sub-optimality, consider the query $Q = /a[/b$ and $/c]$, and the document $D$ depicted in Figure 2.8. There is no match for $Q$ in $D$ (i.e., the DOE is 0), but both TwigStack and TwigStackList keep the $n$ paths $(a_1, b_1), \ldots, (a_1, b_n)$ in memory until they reach their second phase, in which they merge path solutions.

![Figure 2.8: A document demonstrating the sub-optimality of TwigStack and TwigStackList on the query /a[/b and /c].](image)
We now describe a minor modification to the known TwigStack algorithm. The new version uses $O(DOE)$ space in the worst-case, for queries which do not consist of the $(//)(/) pattern. Finding matching upper bounds for queries that contain the $(//)(/) pattern remains an open problem.

We suggest the following modification. Note that each child axis node in this fragment can match only elements of the same depth (see Proposition 2.7). Therefore, we can filter its stream to read only such candidate elements. Done that, we can run twigstack as if all nodes have a descendant-axis. Now consider the example document and query presented in Figure 2.8. The suggested modification will result in automatically ignoring the element $c$, since its depth is not 2 (as in the query), and therefore twigStack will not consider any of the $b_i$ elements as candidates for a match, and therefore will not store them.

2.A Filtering lower bound

In Section 2.4.2, we proved Theorem 2.1 for the special case $Q = //a/b$. We now prove the more general result:

**Theorem 2.1 (restated)** Let $a, b$ be any two labels, and let $Q$ be any basic twig query that contains the path segment $//a/b$. Furthermore, assume $a \neq b$ and that $a, b$ do not appear elsewhere in $Q$. Then, for every algorithm for $\text{FILTER}_Q$ and for every $d \geq 1$, there exists a document of depth at most $d - 1 + \text{depth}(Q)$, on which the algorithm uses at least $d - O(|Q| \log(|Q| \cdot d))$ bits of space.

**Proof.** We first characterize the structure of $Q$: a schematic illustration of $Q$ is presented in Figure 2.9.

The spinal path of $Q$ is the path from the root of $Q (f_0)$ to $b$. Every $T_i$, for $i = 0, \ldots, k$ [or $i = a, b$], represents all the subtrees rooted at the children of $f_i$ [or $a, b$]. $T_i$ is essentially a forest, and may be empty. Note that the subtrees included in $T_i$ can occur on either side of the spinal path.

Let $n = d$. We prove the theorem by showing an MDS reduction from the reverse-set-disjointness problem ($\text{RDISJ}_n$) to $\text{FILTER}_Q$. The MDS reduction is based on the following functions:

- $r_{in}^1$ and $r_{in}^2$ construct the index streams $T_a$ and $T_b$, respectively, of an XML document $D(x, y)$. The document structure, which is presented in Figure 2.10, is as follows. All edges denote parent-child relationships. $f_0, \ldots, f_k$ and $T_0, \ldots, T_k$ are exact copies of the corresponding elements in $Q$. $T_{a,i}$ and $T_{b,i}$ are an exact copy of $T_a$ and $T_b$, respectively, for $i = 1, \ldots, n$. The only difference between documents of different $(x, y)$ is the labels of the nodes $s_1, s_2, \ldots, s_n$ and $t_1, t_2, \ldots, t_n$. When $x_i = 1$, the corresponding node $s_i$ is labeled ’a’, and otherwise it is labeled ’c’. When $y_i = 1$, the corresponding node $t_i$ is labeled ’b’, and otherwise it is labeled ’d’. ’c’ and ’d’ are any labels that do not appear in $Q$. 

47
Figure 2.9: A schematic illustration of a basic twig query that contains a unique “//a/b”.

- \( r^1_e, \ldots, r^l_e \) construct the index streams of all the other labels in \( Q \) (except for \( T_a \) and \( T_b \)), of the document \( D(x, y) \).

- \( r_{out}(q) = q \) (note that the output of both \( \text{RDISJ}_n \) and \( \text{FILTER}_Q \) is one bit). Therefore \( \text{DSS}(r_{out}) = 0 \).

Claim 2.11. \( \text{DSS}(r^1_{in}) = \text{DSS}(r^2_{in}) = O(\log(|Q| \cdot n)) \).

Proof. We describe algorithms \( A \) and \( B \) for \( r^1_{in} \) and \( r^2_{in} \), resp. In order to output the next tuple in \( T_a \) [\( T_b \)], \( A \) [\( B \)] advances the stream \( x[y] \) to the next set bit. The tuple created is a simple function of the position of this bit. Specifically, if the position is \( i \), then the index tuple of node \( s_i \) [\( t_i \)] is added to \( T_a \) [\( T_b \)]. The values of the tuple are bounded by the document size, i.e., \( 2|Q| \cdot n \).

It is easy to check that the index streams constructed are well-formed, i.e., sorted by the “Begin” attribute, and that they represent \( T_a \) and \( T_b \) of the document \( D(x, y) \). The space needed for \( A \) [\( B \)] is \( O(\log(|Q| \cdot n)) \) bits for keeping the current position in \( x[y] \) and for computing the index tuple.

Claim 2.12. \( \forall i, 1 \leq i \leq l : \text{DSS}(r^i_{in}) = O(\log(|Q| \cdot n)) \)

Proof. Note that the document \( D(x, y) \) has a fixed structure independent of \( x \) and \( y \). In addition, the position of all the labels, except for \( a \) and \( b \), is also fixed. It follows that the index stream of any label (\( \neq a, b \)) can be generated on-the-fly, based on the position of the required tuple. The values of the tuples
are bounded by the document size, i.e., $2|Q| \cdot n$. Therefore, the space needed to maintain this "virtual" position and for generating the index tuples is $O(\log(|Q| \cdot n))$ bits.

**Lemma 2.11.** $RDISJ_n(x, y) = FILTER_Q(D(x, y))$.

**Proof.** First we assume that $RDISJ_n(x, y) = 1$. By definition, there exists some index $1 \leq i \leq n$, such that both $x_i$ and $y_i^R$ (i.e., $y_{n+1-i}$) are 1. This means that in $D(x, y)$ the label of $s_i$ is 'a' and the label of $t_{n+1-i}$ is 'b'. Since $t_{n+1-i}$ is a child node of $s_i$, since $s_i$ is a descendant of $f_k$, and since all other query nodes can be matched, the value of $FILTER_Q(D(x, y))$ is 1. The proof of the opposite direction is similar. Assume that $RDISJ_n(x, y) = 0$. Then for every index $1 \leq i \leq n$, at most one of $x_i$ and $y_i^R$ (i.e., $y_{n+1-i}$) is 1. Therefore, there is no $s_i$ node whose label is 'a' that has a $t_{n+1-i}$ child whose label is 'b'. Recall that the labels $a$ and $b$ do not appear elsewhere in $Q$, and therefore can appear in the document.
only in nodes $s_*$ and $t_*$. It follows that $\text{FILTER}_Q(D(x,y)) = 0$.

Since $(r^1_m, r^2_m, r^1_c, \ldots, r^l_c)$ construct the index streams of $D(x,y)$, it follows that:

$$r_{out}(\text{FILTER}_Q(r^1_m(x), r^2_m(y), r^1_c(\epsilon), \ldots, r^l_c(\epsilon))) = \text{FILTER}_Q(D(x,y)) = \text{RDISJ}_n(x,y)$$

Therefore, by Corollary 2.1,

$$\text{MDSS}(	ext{FILTER}_Q) \geq \text{MDSS}(	ext{RDISJ}_n) - \text{DSS}(r^1_m) - \text{DSS}(r^2_m) - \sum_{i=1}^{l} \text{DSS}(r^i_c) - \text{DSS}(r_{out})$$

$$\geq n - O(|Q| \log(|Q| \cdot n)) = d - O(|Q| \log(|Q| \cdot d))$$
Chapter 3

Maximal Join of Ranked Lists

3.1 Introduction

Complex search tasks require combining the answers of several basic search queries, by joining the ranked lists [16]. Example 3.1 is used throughout this chapter to demonstrate our framework.

Example 3.1. Consider a tourist Alice who arrives at a city during a festival and wants to find a festive event that is related to classical music, a good Italian restaurant and an underground station that contains an elevator such that all three will be located within a short walking distance. Alice can pose the following three keyword search queries. (1) A search “festival, classical music” over a repository of messages that are associated to geographic locations\(^1\). (2) A Search “Italian restaurant” over an online restaurant guide of the city. (3) A search “underground station, elevator” over a service that provides information about public transportation. Assuming that there are several festive events dispersed over the city, there can be different combinations of an event, a restaurant and an underground station. A combined search should return relevant triples, ordered according to the combined scores of the items. However, integrating the answers to these independent queries can be a hard task for a human user, because in each answer the items are sorted merely according to their relevance with respect to a single query and independently from their proximity to items in the answers to the other queries.

Answering complex search tasks, such as those in the example above, is by joining the ranked lists that are the answers to the basic search queries. The result of a join is a set of combinations, where each combination is a set of related items—comprising a single item from each list. Combinations are ranked according to the ranking scores of the items they comprise.

When integrating information from different data sources, the algorithms should be able to cope with missing data, and should avoid discarding information from one source simply because it does not

\(^1\)For instance, the geo-location API of Twitter provides the ability to attach geographic metadata to tweets. See http://twitter.com.
have a matching information in another source. For instance, in Example 3.1, a festive event in the result of the first query may be relevant to the user even if there is no nearby Italian restaurant according to the result of the second query.

In *incomplete combinations*, nulls are being used to fill in for the missing values. Permitting incomplete combinations can prevent the absence from the result of relevant, yet incomplete, information. However, in order to provide to the user as much relevant information as possible, the result comprises only combinations that are *maximal*, in the sense that none of these combinations is a proper subset of any other combination. Requiring maximality makes the semantics of null values clear—a null in such case means that there is no appropriate item in the search results to complete the combination. Moreover, maximality prevents having in the result several subsets of the same combination, and hence, prevents duplications that are caused by such subsets.

Computing the join of ranked lists, when the result contains only complete combinations, is a well-studied subject. In many papers, the main goal was to minimize the number of items being read and processed. In that aspect, previous papers tried to achieve *instance optimality* (see [29]). Ilyas et al. [43] proposed a method that is based on using pipelining and binary join operators, for computing equijoin of ranked lists. This approach has been further investigated by Schnaitter and Polyzotis [65], for the case where the result merely comprises complete combinations. In this study, we show how their algorithm can be modified to return maximal rather than complete combinations. We refer to the modified algorithm by the name HNL. We also consider in this thesis a basic nested-loop algorithm, VNL, which is more efficient than HNL for the case where associations between list items are infrequent.

One of the difficulties when computing the maximal answers in a join of ranked lists is that when the result contains an incomplete answer, the lists for which there is a null item in some combination should be read completely, i.e., till reaching their end. We show this in this thesis. This causes instance optimality to be a problematic measure—some lists must be read completely by any algorithm and it becomes impossible to discern between algorithms by the number of items read from these lists. So, up to a constant, all the algorithms read the same number of items (assuming the lists for which there are nulls in the result are not significantly shorter than the other lists). Accordingly, instance optimality for computing a $\theta$-approximate answer (see Section 3.5 and [29]) is a more appropriate measure when computing maximal answers.

In this work we introduce two novel join algorithms that return maximal answers. One algorithm, namely TMax, is less efficient than the other algorithms, however, it reads less items from the lists in comparison to the other algorithms, it is instance optimal, and it can provide a $\theta$-approximate answer, to any $\theta \geq 1$. A second novel algorithm is a hybrid between VNL and HNL and it is more efficient than both VNL and HNL in almost all cases.
3.1.1 Chapter Outline

In Section 3.2 we formally define the problem of maximal join of ranked lists. We present our join algorithms and the optimizations that were used for an efficient implementation of these algorithms in Section 3.3. Analytic comparison is given in Section 3.4. Instance optimality and \( \theta \)-approximation are discussed in Section 3.5, and the hybrid algorithm is presented in Section 3.6. The results of our experimental evaluation are given in Section 3.7, and Section 3.8 surveys related work.

3.2 Framework

In this section, we present our framework, and we formally define the problem of computing top-k maximal answers in a join of ranked lists.

Information Sources and Search Queries. An information source is a set of items of some type, e.g., a repository of documents, geographic objects in a geographic information system, people records in a database, etc.

A basic search query is posed over an information source to find specific items. When a search query \( q \) is posed over an information source \( S \), the answer \( q(S) \) is a sequence \( o_1, \ldots, o_{k_q} \) of items retrieved from \( S \), sorted by their relevance score. The relevance score of an item \( o \), denoted by \( \text{score}_q(o) \), specifies to what extent the item is related to the query. In keyword search, the relevance score of an item can be determined by taking into account several factors, such as the number of keywords that appear in the attributes of the item, the “importance” of these keywords and the “importance” of the attributes in which they appear. Ordinary ranking methods, such as TF-IDF, Okapi BM25 [51, 63] and others [64], can be applied to determine relevancy. Additional constraints can be used to specify exact conditions on specific attributes.

Formally, \( q(S) \) satisfies: \( \text{score}_q(o_{j_1}) \geq \text{score}_q(o_{j_2}) \), for every \( 1 \leq j_1 < j_2 \leq k_q \).

Example 3.2. Consider the scenario presented in Example 3.1 where the three queries being posed are \( q_1 \), \( q_2 \) and \( q_3 \). The answers to \( q_1 \), \( q_2 \) and \( q_3 \) are ranked lists of festive events, restaurants and underground stations, respectively.

Combined Search Query. A combined search query comprises several basic search queries and the information sources over which they are posed. We denote by \( Q = ((q_1, S_1), (q_2, S_2), \ldots, (q_t, S_t)) \) a combined query that consists of the search queries \( q_1, q_2, \ldots, q_t \) over the sources \( S_1, S_2, \ldots, S_t \), respectively, and of a join condition. When the sources are clear from the context, we denote the query as \( Q = (q_1, q_2, \ldots, q_t) \). We refer to \( q_1, \ldots, q_t \) as the subqueries of \( Q \).

A preliminary step when evaluating a combined search query \( Q \) is to evaluate all its subqueries. The result of this step is the pre-answer of \( Q \). The pre-answer of a combined query \( Q = ((q_1, S_1), \cdots, (q_t, S_t)) \)
is \( \text{PreAns}(Q) = (q_1(S_1), \ldots, q_t(S_t)) \), where each \( q_j(S_j) \) (for \( 1 \leq j \leq t \)) is the answer to \( q_j \) over \( S_j \). Since each answer is a list of items, we will refer to \( q_j(S_j) \) as the \( j \)-th list of \( \text{PreAns}(Q) \) and we will denote it by \( L_j \). We denote the \( i \)-th item in list \( L_j \) by \( o_j^i \).

The join condition is a boolean function that defines whether items are associated w.r.t. the combined query. For instance, in Example 3.1, two items are associated if the distance between them is a walking distance (say, 1000 meters). We consider a pair of such items (two items that are associated) as an association and we refer to the items as join consistent. The predicate \( \text{JoinConsistent}(o_i, o_j) \) is true when \( o_i \) and \( o_j \) are associated, and false otherwise.

Given a combined query \( Q \), items of different lists of the pre-answer are joined to form combinations. The join operation is based on the join condition, and only associated items may be joined.

**Complete Combinations.** A complete combination is a \( t \)-tuple of items \( C = (o_1^1, o_2^2, \ldots, o_t^t) \) ∈ \( L_1 \times L_2 \cdots \times L_t \) such that each item is an element of the corresponding list of the pre-answer, and every pair of items in \( C \) satisfies the join condition. We denote by \( \text{Join}(Q) \) the set of all the combinations that are produced by joining the answers to the subqueries of \( Q \), i.e., the join of the lists of the pre-answer of \( Q \).

The score of a combination \( C \), denoted \( \text{score}(C) \), is the result of a monotonic function over the item scores. In particular, it can be any linear combination achieved by giving different weights to the different relevance scores.

**Maximal Combinations.** We represent missing items using nulls. A null item is denoted by the \( \bot \) symbol. A partial combination is a \( t \)-tuple of items, \( C = (o_1^1, o_2^2, \ldots, o_t^t) \) ∈ \( (L_1 \cup \{\bot\}) \times (L_2 \cup \{\bot\}) \cdots \times (L_t \cup \{\bot\}) \), where each item is either an element of the corresponding list of the pre-answer or a null item, and every pair of non-null items in \( C \) satisfies the join condition. When computing the score of a partial combination, a null item is considered as an item whose score is zero.

A partial combination \( C_1 = (o_{i_1}, o_{i_2}, \ldots, o_{i_t}) \) subsumes a partial combination \( C_2 = (o_{j_1}, o_{j_2}, \ldots, o_{j_t}) \), denoted \( C_1 \succ C_2 \), if for every \( 1 \leq k \leq t \), either \( o_{j_k} = \bot \) or \( o_{j_k} = o_{i_k} \). That is, every non-null item in \( C_2 \) is equal to the corresponding item in \( C_1 \). For example, \( (o_1, o_2, \bot, o_4) \) subsumes \( (\bot, o_2, \bot, o_4) \). However, neither \( (o_1, o_2, \bot, o_4) \) nor \( (o_1, o_2, o_4, o_5) \) subsume \( (o_1, o_2, o_3, \bot) \) because in both cases, the non-null object \( o_3 \) does not have an equal object in the corresponding position, assuming \( o_3 \neq o_4 \).

A combination is maximal if it is not a proper subset of any other combination. Formally, a combination \( C \) is maximal if there is no combination \( C' = (L_1 \cup \{\bot\}) \times (L_2 \cup \{\bot\}) \cdots \times (L_t \cup \{\bot\}) \) such that \( C' \neq C \) and \( C' \succ C \). That is, there is no combination that subsumes \( C \), other than \( C \) itself.

We denote by \( \text{MaxJoin}(Q) \) the set of all the maximal combinations that are produced by joining the answers to the subqueries of \( Q \). The top-\( k \) answers of \( Q \) are the \( k \) combinations with the highest scores in \( \text{Join}(Q) \). The top-\( k \) maximal answers of \( Q \) are the \( k \) combinations with the highest scores in \( \text{MaxJoin}(Q) \).
Example 3.3. Consider a combined search query comprising the three queries \(q_1, q_2, q_3\) of Example 3.2. Suppose that \(e\) is an event in the answer returned by \(q_1\), \(r\) is a restaurant in the answer returned by \(q_2\) and \(s\) is a station returned by \(q_3\). Then a triplet \((e, r, s)\) is a combination if every pair \((e, r), (r, s)\) and \((e, s)\) satisfies the join condition of being located within a walking distance. The triplet \((e', \perp, s')\) is a partial combination if \(e'\) is an event near the station \(s'\) and it is a maximal combination if there is no restaurant in the answer to \(q_2\) that satisfies the join condition with respect to both \(e'\) and \(s'\).

The main problem studied in this chapter is how to efficiently compute the top-k maximal answers of a given query \(Q\).

### 3.3 Algorithms

In this section we present three algorithms for computing the top-k maximal answers, to a given combined search query \(Q\) (the fourth algorithm is presented in Section 3.6). The first two algorithms are adaptations of existing algorithms for computing the top-k complete answers. Throughout this section, we assume that the input consists of the lists \(L_1, \ldots, L_t\) of \(PreAns(Q)\). In two algorithms, the initial step is to modify these lists by adding a null value with score zero at the end of each list. We denote these extended lists by \(L^+_1, \ldots, L^+_t\), i.e., \(L^+_j = L_j \cup \{\perp\}\), for \(j = 1 \ldots t\). The join condition is such that a null value, in any list, is join consistent with any item and with any null of any other list.

The algorithms we present manage a heap of size (at most) \(k\) that stores combinations. We denote by \(\text{MinScore}(H)\) a function that returns the minimal score of a combination in \(H\) if the heap is full, or zero in the case that \(H\) currently contains less than \(k\) combinations. That is,

\[
\text{MinScore}(H) = \begin{cases} 
\min\{\text{score}(C) \mid C \in H\}, & \text{if } |H| = k \\
0, & \text{otherwise.}
\end{cases}
\]

Finding the top-k maximal combinations is more intricate than answering common top-k queries (queries that return merely complete answers). There are two main subtle cases that should be handled correctly. One case is when a new combination \(C\) is found, but the heap already contains a different combination that subsumes \(C\). The second case is when a new combination \(C'\) is added to the heap and the heap already contains one or more combinations that are subsumed by \(C'\). The subsumed combinations should be discarded. However, before removing them from the top-k heap, they occupy positions that should have been free, and without properly handling this, it may lead to a failure in inserting into the heap combinations that belong to the top-k answer. The following example illustrates this.

Example 3.4. Consider the following three lists, of two items each. List \(L_1\) contains the items \(a_1\) and
Figure 3.1: Example of three lists of two items each. Connections between items denote associations.

\[ a_2, \text{with scores 10 and 2, respectively. List } L_2 \text{ contains the items } b_1 \text{ and } b_2, \text{with scores 10 and 3, respectively. List } L_3 \text{ contains the items } c_1 \text{ and } c_2, \text{with scores 10 and 4, respectively. The items } a_1 \text{ and } b_1 \text{ are join consistent. The items } a_2, b_2 \text{ and } c_2 \text{ are pairwise join consistent. Every other pair of items is not join consistent. The lists and the associations between the items are presented in Figure 3.1. Let the score of a combination be the sum of scores of the items it comprises. The top-2 answer of the query is } \{(a_1, b_1, \bot), (\bot, \bot, c_1)\}. \text{ Note that the answer comprises only partial combinations, although there exists a complete combination, } (a_2, b_2, c_2), \text{ because the score of } (a_2, b_2, c_2) \text{ is lower than the scores of the top-2 partial combinations.}

To illustrate the first case, suppose an algorithm initially inserts the combinations \((a_2, b_2, c_2)\) and \((a_1, b_1, \bot)\) into the heap, and then tries to insert \((a_1, \bot, \bot)\) into the heap. Although the score of \((a_1, \bot, \bot)\) is higher than the score of \((a_2, b_2, c_2)\), it should not be added to the heap, because it is subsumed by \((a_1, b_1, \bot)\).

To illustrate the second case, consider an algorithm that initially inserts \((a_1, \bot, \bot)\) and \((\bot, b_1, \bot)\) into the heap. Later, when \((\bot, \bot, c_1)\) is discovered, it is not added to the top-2 heap, because its score is not higher than \(\text{MinScore}(H)\). However, when \((a_1, b_1, \bot)\) is added to the heap, both \((a_1, \bot, \bot)\) and \((\bot, b_1, \bot)\) are subsumed by it and therefore should be removed from the heap. The top-2 heap now contains only one combination, and \((\bot, \bot, c_1)\) should not have been deemed irrelevant.

Example 3.4 demonstrates the importance of the order by which combinations are inserted into the heap. A naive top-k algorithm that iterates over combinations in an arbitrary order, and maintains the top-k combinations seen so far, is correct for computing top-k complete combinations, but may be erroneous for computing maximal answers.

Due to the considerations explained above, we changed the procedure of adding a combination to the top-k heap as follows. We denote by \(H.add(C)\) the two-step addition operation that (1) adds a combination \(C\) to \(H\), if there is no other combination in \(H\) that subsumes \(C\); and (2) if after the addition there are \(k + 1\) elements in \(H\), removes from \(H\) the combination with the lowest score.

We next present the first two algorithms, which are based on existing algorithms for computing the
Algorithm 3.1 Vertical Nested Loops

Input: $L_1, \ldots, L_t; k$
Output: Top-k maximal combinations

1: add an item $\bot$ with score 0 to the end of each list, creating $L_1^+, \ldots, L_t^+$
2: let $H$ be a heap of size $k$
3: $H := \emptyset$
4: for $a_1$ in $L_1^+$ do
5:   if $\text{score}(a_1, o_2^1, \ldots, o_t^1) < \text{MinScore}(H)$ then
6:     break
7:   end if
8: end for
9: for $a_i$ in $L_i^+$ do
10:   let $s = \text{score}(a_1, \ldots, a_{i-1}, o_i^{i+1}, \ldots, o_t^1)$
11:   if $s < \text{MinScore}(H)$ then
12:     break
13:   end if
14:   if $\exists 1 \leq j \leq i - 1$ such that $\neg \text{JoinConsistent}(a_j, a_i)$ then
15:     continue
16:   end if
17:   for $a_t$ in $L_t^+$ do
18:     if $\text{JoinConsistent}(a_t, a_j)$, for $1 \leq j \leq t - 1$ then
19:       newComb := $(a_1, \ldots, a_t)$
20:       $H$.add(newComb)
21:     end if
22:   end for
23: end for
24: end for
25: return $H$

3.3.1 The Vertical Nested Loop Algorithm

Algorithm Vertical Nested-Loop (VNL) is an optimized version of a nested loop algorithm that iterates over the lists in a vertical fashion (see Algorithm 3.1). Essentially, VNL iterates through the lists by nested loops and maintains the top combinations in a heap $H$. Whenever it finds a combination $C$, it inserts $C$ to $H$ by applying $H$.add($C$), (i.e., $C$ is being added to $H$ only if there is no combination in $H$ that subsumes $C$, and the score of $C$ is higher than $\text{MinScore}(H)$). In order to increase the efficiency, VNL breaks the loops in cases where continuing the iteration would not be able to produce relevant combinations, i.e., combinations that will be inserted into the top-k heap. Consequently, VNL checks less combinations than a naive nested loop algorithm.

VNL reduces the number of combinations being examined by applying two termination rules on the nested loops.

Rule 1: Suppose the loops through $L_1, \ldots, L_{i-1}$ have reached the items $a_1, \ldots, a_{i-1}$, respectively,
and the loop through $L_i$ has reached $a_i$. If $\text{score}(a_1, \ldots, a_i, o_i^{i+1}, \ldots, o_i^t) < \text{MinScore}(H)$, then the iteration on $L_i$ should be terminated. This is because without advancing the iterations on $L_1, \ldots, L_{i-1}$, only combinations with a score lower than $\text{MinScore}(H)$ will be produced.

**Rule 2:** A tuple of items (i.e., a combination whose construction has not been completed yet), in which some pair of items does not satisfy the join condition, cannot be extended to be a combination. Hence, the algorithm avoids iterating the inner loops and continues to the next iteration of the current loop.

The termination conditions appear in each loop. To see that, consider the $i$-th loop in Algorithm 3.1 (lines 9-16). The conditions for Rule 1 and Rule 2 appear in Line 11 and Line 14, respectively.

**Optimization:** The following optimization technique is being employed to increase the efficiency of VNL. The order of the lists is modified so that short lists or lists whose items have a few associations with items of other lists will appear before the other lists, i.e., will be in the outer loop, and thus, Rule 2 will be applied as early as possible.

In VNL, a partial combination may be added to the heap $H$ prior to the insertion of some complete combination. Yet, in each step of the algorithm all the combinations in $H$ are maximal combinations. This is because a combination $C$ cannot be added to $H$ if $H$ already contains a combination that subsumes $C$ (see the description of the procedure $H.add(C)$), and due to the following lemma.

**Lemma 3.1.** If $C'$ and $C$ are two combinations such that $C' \succ C$ and such that $H.add(C)$ is called during VNL, then $H.add(C')$ is called during VNL and the call $H.add(C')$ precedes the call $H.add(C)$.

**Proof.** Since $C' \succ C$ there exists a list $k$ such that the $k$-th item of $C$ is null while in $C'$ the $k$-th item is not null. That it, if $C' = (o_1^{1}, \ldots, o_k^{2}, \ldots, o_k^{k}, \ldots, o_i^t)$, then $C = (o_1^{1}, \ldots, o_k^{k-1}, \bot, o_{k+1}^{k+1}, \ldots, o_i^t)$. Recall that nulls exist at the end of each list. Thus, this null is reached at the last iteration of the $k$-th loop. Since $H.add(C)$ is called during VNL, the loops of lists $L_1, \ldots, L_{k-1}$ reach the items $o_i^{1}, \ldots, o_{k-1}^{k-1}$, and the $k$-th loop reaches the null item. Within this iteration of the $k$-th loop, the items of $C$ from the following lists are reached and $H.add(C)$ is called. It follows that there exists a previous iteration of the $k$-th loop that reaches the item $o_k^{k}$. It also follows, since $\text{score}(C') > \text{score}(C)$, that within this iteration on $o_k^{k}$, the inner loops will continue and reach the items $o_{k+1}^{k+1}, \ldots, o_i^t$, and then $H.add(C')$ is called. This call preceded the call $H.add(C)$ due to the order of the loops and the sequential progress in the iterations over the lists.

From Lemma 3.1 follows the soundness of VNL. All the combinations that are returned are maximal. To prove correctness, we also need to prove that they are the top-$k$ maximal combinations.

Although VNL applies optimizations that reduce the number of tuples being examined, in the nested loops, all the relevant tuples are being constructed and considered, as stated in the following lemma.
Lemma 3.2. Given a combined search query $Q$ and an integer $k$, if $C$ is among the maximal top-$k$ combinations of $Q$ then VNL calls $H.add(C)$.

Proof. Combination $C$ is join consistent, so any pair of items in it satisfies the join condition, and when the loops reach the items of $C$, $H.add(C)$ is called. Thus, we need to show that the iterations reach the items of $C$.

VNL applies two termination rules. In Rule 1, an iteration is stopped when the leftmost items of a potential combination are too small in the sense that extending the combination could not exceed the minimal score of the top-$k$ heap. However, Rule 1 is not being applied for $C$. To see that, suppose the leftmost items of $C$ are $a_1, \ldots, a_i$. Then $\text{score}(a_1, \ldots, a_i, o_1^{i+1}, \ldots, o_t^i) \geq \text{MinScore}(H)$ because otherwise, $\text{score}(C) < \text{score}(a_1, \ldots, a_i, o_1^{i+1}, \ldots, o_t^i)$ and $\text{score}(a_1, \ldots, a_i, o_1^{i+1}, \ldots, o_t^i) \leq \text{MinScore}(H)$, so $\text{score}(C) < \text{MinScore}(H)$. This means that there are $k$ combinations in $H$ whose score is greater than the score of $C$, in contradiction to $C$ being among the top-$k$ combinations.

In Rule 2, the iteration stops when there are two items that do not satisfy the join condition. However, since $C$ is a combination, every pair of items in $C$ satisfies the join condition, and hence, the iterations continue.

From Lemma 3.2 follows the completeness of VNL. When $H.add(C)$ is called, the combination $C$ is inserted into $H$, because (1) there cannot be in $H$ a combination that subsumes $C$ (otherwise, $C$ would not be maximal); and (2) $H$ cannot include $k$ combinations whose score is higher than the score of $C$ (otherwise, $C$ would not be among the top-$k$ maximal combinations of $Q$).

From Lemma 3.2 and Lemma 3.1 follows the next proposition.

Proposition 3.1. Given a combined search query $Q$ and an integer $k$, VNL returns the top-$k$ maximal answers of $Q$, when $|\text{MaxJoin}(Q)| \geq k$, and $\text{MaxJoin}(Q)$, otherwise.

3.3.2 The Horizontal Nested Loop Algorithm

The Horizontal Nested-Loop Algorithm, HNL for short, is based on the following existing algorithm template.

3.3.2.1 The Pull Bound Rank Join Pattern

Computing the join of ranked lists is a well-studied subject. Ilyas et al. proposed a method that is based on using pipelining and binary join operators, for computing equijoin of ranked lists [43]. In their proposed algorithms the lists are being scanned and the items are inserted into hash tables where the hash keys are the join attributes. This makes the search for pairs of join-consistent items efficient. However, in cases where the join is not an equijoin, as in Example 3.1, such methods cannot be used.
Algorithm 3.2 Horizontal Nested Loops

Input: $L_1, \ldots, L_t; k$

Output: Top-k maximal combinations

Data Structures: input buffers $T_1, \ldots, T_t$; minimum heap $H$

1: add an item $\bot$ with score 0 to the end of each list, creating $L_1^+, \ldots, L_t^+$
2: $H := \emptyset$ where $H$ is a minimum heap of size $k$
3: $j := 0$
4: $b := \infty$
5: while $b > \text{MinScore}(H)$ do
6: \hspace{1em} if $j < t$ then $j := j + 1$ else $j := 1$
7: \hspace{1em} $a_j := \text{NextItemInList}(j)$
8: \hspace{1em} $R := \text{Join}(T_1, \ldots, T_{j-1}, \{a_j\}, T_{j+1}, \ldots, T_t)$
9: \hspace{1em} for $C$ in $R$ do
10: \hspace{2em} $H.\text{add}(C)$
11: \hspace{1em} end for
12: \hspace{1em} $\text{Add} \ a_j \ \text{to} \ T_j$
13: \hspace{1em} $b := \text{updateBound}(a_j)$
14: end while
15: return $H$

A general pattern—Pull Bound Rank Join (PBRJ)—for a join of ranked lists, was suggested and investigated by Schnaitter and Polyzotis [65]. They have studied the use of bounds for achieving instance-optimal algorithms.

PBRJ operates iteratively, as follows. On each iteration, a new item $o^j_i$ is read from a list $L_i$, according to the pulling strategy. The item is stored in an input buffer, and new combinations are generated by joining $o^j_i$ to items that have already been read from the other lists. The generated combinations are pushed into a heap $H$. After being processed, each item is given to a bounding scheme $B$, such that $B$ returns an upper bound $b$ on the scores of unseen combinations. Let $C_{\text{min}}$ be the lowest ranked combination in $H$. When the score of $C_{\text{min}}$ is not smaller than the upper bound $b$, $H$ is returned as the top-k answer, because the upper bound $t$ indicates that the scores of all the combinations to be generated by processing additional items will be lower than the score of $C_{\text{min}}$.

3.3.2.2 HNL

The HNL algorithm (Algorithm 3.2), is based on the PBRJ template with some modifications. We process the items horizontally, and use the same bounds suggested in [43], but instead of applying it over $L_1, \ldots, L_t$, we apply it over the lists $L_1^+, \ldots, L_t^+$. Given lists $L_1, \ldots, L_t$ and a value $k$, the algorithm iterates over the extended lists horizontally, and in each iteration adds the discovered combinations to a minimum heap $H$ of size $k$.

Initially, the algorithm visits the first item of each list, and in the $i$-th iteration it visits the $i$-th item of each list. When visiting an item $o^j_i$ of list $j$, the algorithm joins the singleton $\{o^j_i\}$ with all the items of the lists $L_1^+, \ldots, L_{j-1}^+, L_{j+1}^+, \ldots, L_t^+$, that were visited before $o^j_i$. That is, the algorithm computes the join of the lists $T_{i,1}, \ldots, T_{i,j-1}, \{o^j_i\}, T_{i,j+1}, \ldots, T_{i,t}$, where $T_{i,j}$ are the top $i$ items of $L_j^+$, and...
it adds the produced combinations to \( H \).

For computing the join in each iteration, the algorithm employs the vertical nested loop algorithm (VNL) that was presented in Section 3.3.1. As an optimization, it takes the singleton list as the outer most list in the nested loop.

In addition to the bounding scheme used in Lines 5 and 13, which applies early termination, we suggest the following optimization, which does not appear in Algorithm 3.2. When for some visited item \( o_i^j \) it holds that \( \text{score}(o_1^1, \ldots, o_i^{j-1}, o_i^j, o_i^{j+1}, \ldots, o_i^1) \leq \text{MinScore}(H) \), the algorithm stops visiting items of list \( j \). That is, let \( \tau_i^j \) be the tuple that contains \( o_i^j \) and the first item \( o_k^j \) of list \( L_k^+ \), for all \( k = 1 \ldots j-1, j+1, \ldots t \). If the score of \( \tau_i^j \) is lower than the score of the lowest combination in the heap \( H \), then there is no additional possible combination that can be added to \( H \), among the combinations that contain an item lower than \( o_i^j \) in \( L_j^+ \). Thus, the algorithm stops visiting items below \( o_i^j \) in \( L_j^+ \). When in all the lists the algorithm either reaches this stopping condition or reaches the end of the list, the computation terminates.

Note that in HNL, all the complete combinations are constructed prior to the construction of partial combinations.

The correctness of the algorithm emanates from the following properties of the algorithm.

**Lemma 3.3.** Given two combinations \( C' \) and \( C \), such that \( C' \succ C \), and such that \( H.\text{add}(C) \) is called during \( \text{HNL} \), then: (i) \( H.\text{add}(C') \) is also called during \( \text{HNL} \); (ii) the call \( H.\text{add}(C') \) precedes the call \( H.\text{add}(C) \); and (iii) when \( H.\text{add}(C) \) is called, \( C \) will not be added to \( H \).

**Proof.** Since \( C' \succ C \) there exists a list \( L_n \) such that the \( n \)-th item in \( C \) is null while in \( C' \) it is not. In case there are multiple such lists, let \( L_n \) be the rightmost list (assuming the lists are ordered from left to right). Therefore, if \( C' = (o_1^{t_1}, o_2^{t_2}, \ldots, o_n^{t_n}, \ldots, o_t^1) \) such that \( o_n^t \neq \bot \), then \( C = (o_1^{t_1}, \ldots, o_{n-1}^{t_{n-1}}, \bot, o_{n+1}^{t_{n+1}}, \ldots, o_t^1) \) such that \( o_{n+1}^{t_{n+1}}, \ldots, o_t^1 \neq \bot \). In addition, let \( L_l \) be the list of the rightmost null item in \( C \). Recall that nulls are located at the end of each list, therefore these nulls are reached only when \( \text{HNL} \) processes the last row. To prove the lemma, we need to analyze two cases.

In the first case, \( n = l \), i.e., the rightmost null item in \( C \) is on the \( l \)-th list, and there is no other null in both \( C \) and \( C' \) to the right. Given that \( H.\text{add}(C) \) is called during \( \text{HNL} \), then \( C \) is generated when \( \text{HNL} \) processes the null item of \( L_l \), and then \( H.\text{add}(C) \) is called. Therefore there exists an earlier iteration in which \( \text{HNL} \) processes the item \( o_n^{\tau_{l_t}} \). It also follows that there exists an iteration, in which \( \text{HNL} \) generates \( C' \) and calls \( H.\text{add}(C') \). This iteration preceded the call to \( H.\text{add}(C) \) because the lowest item in \( C' \) is either located higher on the lists than the nulls row, or to the left of \( L_l \), and is therefore processed before the null item of \( L_l \).

In the second case, \( n \neq l \), i.e., the \( l \)-th item in both \( C \) and \( C' \) is null. Given that \( H.\text{add}(C) \) is called during \( \text{HNL} \), then \( C \) is generated when \( \text{HNL} \) processes the null item of \( L_l \). Since this is the lowest and rightmost item in both \( C \) and \( C' \), then both are generated on this same iteration. The order in which
they are generated now depends on the VNL algorithm. Since the order of the loops in VNL is from left to right, and it processes the items of each list sequentially, then it reaches \( o_{in}^j \) before reaching the null item of \( L_n \), and also generates \( C' \) and calls \( H.add(C') \) before it generates \( C \).

Finally, when \( H.add(C) \) is called, the heap \( H \) already contains a combination that subsumes \( C \). It is either \( C' \) or another combination that subsumes both \( C \) and \( C' \), and was added before \( C' \). Either way, \( C \) will not be added to \( H \), because \( H.add() \) adds a combination to \( H \) only if \( H \) does not contain a combination that subsumes it.

It follows from Lemma 3.3 that the heap always contains only maximal combinations. The following lemma shows that all maximal top-\( k \) combinations are returned. The two lemmas prove the correctness of HNL.

**Lemma 3.4.** Let \( C \) be a maximal combination among the top-\( k \) combinations of the joined ranked lists. Then, \( C \) appears in the top-\( k \) heap at the end of the run of HNL.

**Proof.** The lemma follows from the claim that HNL calls \( H.add(C) \). This is because when calling \( H.add(C) \), combination \( C \) is among the top-\( k \) combinations and \( H \) contains \( k \) combinations, so \( C \) will be inserted into \( H \). Any combination whose score is less than the score of \( C \) would not cause the removal of \( C \) from the heap, so \( C \) will remain in the heap.

To show that HNL calls \( H.add(C) \) we need to show that the algorithm does not terminate before reaching \( C \). Let \( o^j \) be the \( j \)-th item of \( C \) and let \( \bar{o}^j \) be either \( o^j \) or an item that appears in list \( L_j \) before \( o^j \). Then, when processing item \( \bar{o}^j \), the bound is \( b = score(o_1^1, \ldots, o_{j-1}^1, \bar{o}^j, o_{j+1}^1, \ldots, o_t^1) \). Since \( o^j \) succeeds \( \bar{o}^j \), \( score(o^j) \leq score(\bar{o}^j) \). Thus, \( b = score(o_1^1, \ldots, o_{j-1}^1, \bar{o}^j, o_{j+1}^1, \ldots, o_t^1) \geq score(o_1^1, \ldots, o_{j-1}^1, o^j, o_{j+1}^1, \ldots, o_t^1) \). Also, \( score(o_1^1, \ldots, o_{j-1}^1, o^j, o_{j+1}^1, \ldots, o_t^1) \geq score(C) \) because \( o_t^1 \) is the item with the highest score in list \( L_i \), for \( 1 \leq i \leq t \).

In addition, prior to adding \( C \) to \( H \), \( score(C) > MinScore(H) \), because \( C \) is among the top-\( k \) scores. From all these follows that \( b > MinScore(H) \) prior to the insertion of \( C \) to the heap, so HNL does not terminate prior to calling \( H.add(C) \).

Lemma 3.3 and Lemma 3.4 show the soundness and the completeness of HNL, as stated by the following proposition.

**Proposition 3.2.** Given a combined search query \( Q \) and an integer \( k \), HNL returns the top-\( k \) maximal answers of \( Q \), when \( |MaxJoin(Q)| \geq k \), and MaxJoin(\( Q \)), otherwise.

### 3.3.3 The TMax Algorithm

Algorithms VNL and HNL are modifications of algorithms for computing complete answers. In this section we present Algorithm TMax that in each step generates the maximal combinations with respect
### Algorithm 3.3 TMax.

**Input:** \( L_1, \ldots, L_t; k \)

**Output:** Top-\( k \) maximal combinations

1. let \( H \) be a top-\( k \) heap
2. \( H := \emptyset \)
3. \( TMC := \emptyset \)
4. for \( r = 1 \) to \( m \) do \( \triangleright \) iterates on rows
   5. for \( l = 1 \) to \( t \) do \( \triangleright \) iterates on lists
   6. \( \text{UpdateTMC}(TMC, o^r_l, l, H) \)
5. end for
6. end for
7. return \( H \)

1. **function** \( \text{UpdateTMC}(TMC, o, l, H) \)
2. \( C := \emptyset \)
3. for each \( P \) in \( TMC \)
4. if \( P \) contains an item \( o_l \) from list \( L_l \) then
5. \( N := P \setminus \{ o_l \} \)
6. else
7. \( N := P \)
8. end if
9. add \( o \) to \( N \)
10. for each \( o' \) such that \( \text{JoinConsistent}(o, o') = \text{false} \)
11. remove \( o' \) from \( N \)
12. end for
13. add \( N \) to \( C \)
14. if \( P \subseteq N \) then
15. remove \( P \) from \( TMC \)
16. end if
17. remove from \( C \) duplicates and subsets
18. for each \( P \) in \( C \)
19. if \( \text{score}(P) > \text{MinScore}(H) \) then
20. \( H.\text{removeSubsetsAndAdd}(P) \)
21. end if
22. \( TMC := TMC \cup C \)
23. end function

The algorithm **Managing Temporarily-Maximal Combinations (TMax)**, iterates over the lists horizontally and builds *temporarily-maximal* combinations. That is, in each step, it creates combinations that are maximal with respect to the seen items, but are not necessarily maximal w.r.t. all the items of the given lists. It continues till maximality w.r.t. all the items is guaranteed. As in the previous algorithms, the top-\( k \) combinations are managed using the heap \( H \) (see Algorithm 3.3).

The main difference between TMax and VNL is that TMax visits items by scanning rows rather than columns. The main difference between TMax and HNL is that TMax generates partial temporary combinations instead of initially generating the complete combinations.

In TMax, each item is being read once, and a suitable data structure, named \( TMC \), is being used for...
storing partial combinations of visited items, for cases where these partial combinations will be extended by items that have not been visited thus far.

The rational behind reading the items horizontally, in TMax, is similar to that of HNL. In the lists the items are sorted by their scores, and it is beneficial to read and join items with high scores as soon as possible. In vNL, for comparison, loops go through all the lists, so inner loops may visit items with low scores already in early stages.

In order not to read items several times, TMax maintains a data structure of temporarily-maximal combinations (referred to as TMC). The algorithm iterates over the rows of the lists, i.e., it reads all the first items of the lists, then the second items, and so on. It processes the items, and incorporates them into TMC.

TMax iterates on the rows (Line 4 of Algorithm 3.3), and for each row iterates on the items of the row (Line 5). When processing an item, TMax tries extending with it all the partial combinations that were already found, i.e., adding the new item to those combinations that are in the TMC (Line 6). When adding an item o to a partial combination P for creating a new partial combination, it is essential to remove from P the item which belongs to the same list as o, if such item exists in P (Line 4 and Line 5 of the function UpdateTMC). It is also necessary to remove from P items that are not join consistent with o, so that the created tuple will be a partial combination (Line 10 and Line 11 of UpdateTMC).

In order not to increase the size of TMC more than required, we only keep in it temporarily maximal combinations (maximal w.r.t. all the combinations considered until now). Thus, at the end of each iteration, before adding to TMC the set C of new partial combinations created for an item o, combinations that are subsumed by other combinations, and duplications, are being discarded (Line 18).

Along with adding the set C to TMC, the algorithm tries to add every maximal combination P in C to the top-k heap (Lines 19-22). The insertion succeeds if the score of P is high enough. In this algorithm, as opposed to the previous ones, the heap cannot contain combinations that subsume P but it may contain a combination that is subsumed by P. The reason to this is that partial combinations are always discovered before complete combinations, i.e., they can only be extended by adding items to it during execution. Therefore, when adding P to H, the subsumed combination is removed from H.

**Lemma 3.5.** During the TMax algorithm, TMC always contains all the maximal combinations composed of items seen thus far, and H contains the top-k combinations of TMC.

**Proof.** We prove the lemma by induction on i—the number of items processed by TMax. The base case corresponds to the beginning of the algorithm, before any item has been read. In this case both TMC and H are empty, and thus satisfy the required conditions.

Suppose the lemma holds for all the first i items read by TMax. Let item o be the i + 1-st item read by TMax. First, note that before calling the function UpdateTMC with o, TMC contains all the maximal combinations composed of the first i items. Second, before adding to TMC the new set C, C contains all
the maximal combinations among the combinations that contain the item \( o \). This is because all possible extensions of \( TMC \) by \( o \) are added to \( C \), and because Line 17 discards duplications and subsets within \( C \).

Third, for any pair of combinations—\( C_m \) that is an element of \( TMC \) prior to the addition, and \( C_n \) that is an element of \( C \)—one cannot be subsumed by the other. On one hand, \( C_n \) is not subsumed by \( C_m \) because \( C_n \) contains \( o \), whereas \( C_m \) does not. On the other hand, \( C_m \) is not subsumed by \( C_n \) because otherwise it would have been removed earlier (see Lines 14 and 15). This is because if \( C_m \) is subsumed by \( C_n \) then \( C_m \) is subsumed by \( C_n - \{ o \} \), however, according to the induction hypothesis, \( C_n - \{ o \} \) is already in \( TMC \) and subsumes \( C_m \), in contradiction to the maximality of \( C_m \) as an element of \( TMC \).

It follows from these three observations, that after adding \( C \) to \( TMC \), \( TMC \) still contains only temporarily maximal combinations.

After reading and processing \( o \), \( TMC \) contains all the maximal combinations produced by the first \( i + 1 \)-st items. All the combinations that do not contain \( o \) are produced by the first \( i \)-th items and according to the induction hypothesis are in \( TMC \). All the combinations that contain \( o \) are produced in the \( i + 1 \)-st step and are added to \( TMC \). At the end, \( TMC \) contains all the maximal combinations of the join of the given lists, and thus, \( H \) contains the top-k maximal answers.

From Lemma 3.5 follows the correctness of TMax.

**Proposition 3.3.** Given a combined search query \( Q \) and an integer \( k \), Algorithm TMax returns the top-k maximal answers of \( Q \), when \( |\text{MaxJoin}(Q)| \geq k \), and \( \text{MaxJoin}(Q) \), otherwise.

**Optimizations:** We included in the implementation of TMax two simple, yet effective, optimizations. These optimizations have a significant effect on the efficiency of the algorithm.

The first optimization is by reducing the size of \( TMC \) whenever possible. Recall that for every item the algorithm iterates over all the elements of \( TMC \). Thus, decreasing the size of \( TMC \) has an immediate effect on the running time. We decrease the size of \( TMC \) by examining, after each row, all the partial combinations, and computing for each partial combination an upper bound on its score. If for a partial combination, the upper bound is below the minimal score of the top-k heap, we remove it from \( TMC \). More precisely, consider a partial combination \( P \) that is in \( TMC \) at the end of the iteration on the \( r \)-th row, i.e., after processing the items \( \alpha_{i_r}^1, \ldots, \alpha_{i_r}^l \). Suppose that the lists for which no item is included in \( P \) are \( L_{i_1}, \ldots, L_{i_n} \). Then, we consider the combination \( P^+ \) that is generated by adding \( \alpha_{i_r}^1, \ldots, \alpha_{i_r}^s \) to \( P \). Obviously, if \( P \) is extended in following iterations, it is by items whose scores are lower than the scores of \( \alpha_{i_r}^1, \ldots, \alpha_{i_r}^s \), and hence, \( \text{score}(P^+) \) is an upper bound on the score of \( P \). If \( \text{score}(P^+) < \text{MinScore}(H) \), then we remove \( P \) from \( TMC \). If at some point \( TMC \) becomes empty because of the removal of combinations, TMax terminates.

The second optimization stops the processing of input lists when reaching items whose score is too low to participate in the result. Similarly to the previous optimization, it is also performed at the end of
every iteration on a row.

Suppose the algorithm completed processing the \( r \)-th row. We apply the optimization as follows. Let us denote by \( o_j^r \) the last item of list \( L_j \) that was processed by the algorithm, for \( 1 \leq j \leq t \), i.e., \( o_j^r \) is the lowest item of list \( L_j \) that was read. Given a partial combination \( P \) in \( TMC \), and an item \( o_j^r \) of list \( L_j \), the upper-bound completion of \( P \) with respect to \( o_j^r \), denoted \( \overline{P_j} \), is the result of replacing each null values of \( P \) with the corresponding item among \( o_1^r, o_2^r, \ldots, o_t^r \) and replacing the item in position \( j \), whether it is null or not, with \( o_j^r \). Let \( B_j = \{ \overline{P_j} \mid P \in TMC \land \text{score}(\overline{P_j}) > \text{MinScore}(H) \} \) be the set of upper-bound completions of partial combinations in \( TMC \), with respect to \( o_j^r \), whose score exceeds the minimum score of the heap \( H \). Then, if \( B_j \) is empty, the iteration over list \( L_j \) is stopped.

Correctness of the optimization follows from the following observation. Item \( o_j^r \) and any item that succeeds \( o_j^r \) may only appear in combinations whose score is not greater than \( \text{MinScore}(H) \). To see that, consider an item \( o_s^r \) in \( L_j \) where \( s \geq r \), i.e., \( o_s^r \) either succeeds \( o_j^r \) or is equal to \( o_j^r \). Let \( C' \) be a combination that contains \( o_s^r \). We denote by \( \text{before-r}(C') \) the items of \( C' \) that appear before the \( r \)-th position in the lists. These items are pairwise join consistent because they are part of \( C' \). Hence, according to Lemma 3.5, in \( TMC \) there is a partial combination \( P \) that contains the items of \( \text{before-r}(C') \). Each item \( o_i \) of \( C' \) that appears in list \( L_i \) on row \( r \) or on a row that succeeds \( r \) has a score lower than the score of \( o_i^r \). Thus, based on the monotonicity of the scoring function, \( \text{score}(C') \leq \text{score}(\overline{P_j}) \leq \text{MinScore}(H) \).

### 3.4 Analysis and comparison

We now analyze the complexity of the algorithms. A standard worst-case analysis is insufficient for comparing the algorithms for a join of ranked lists because in the worst case, the algorithms must read the entire input. However, there are scenarios where the algorithms can terminate before reading the entire input and we want to compare the algorithms on such cases. Also note that merely measuring the amount of items being read from the lists is insufficient because the algorithms also differ in the amount of work being done after reading the input.

Due to the above reasons, we analyze and compare the algorithms for different integration scenarios that are not necessarily worst-case scenarios. In the analysis, we denote by \( t \) the number of lists and by \( m \) the number of items in each list. The parameter \( k \) is the number of combinations in the result.

Computing the top-\( k \) maximal join of lists is hard, in the sense that the decision problem whether for a given input (lists and \( k \)) there are at least \( k \) complete combinations in the result is NP-hard. This follows from the hardness of the computation of a natural join. (The join-emptiness problem, of deciding if the natural join of given relations is empty, is NP-hard [19].)

**Theorem 3.1.** Given a combined search query \( Q \) and an integer \( k \), it is NP-hard to decide whether
MaxJoin($Q$) contains a complete combination.

**Proof.** (sketch) The proof is by a reduction from the join-emptiness problem. Given a set of $t$ relations for which it is desired to decide whether their natural join is empty, we construct a list from each relation by considering the tuples as items. We associate to the items scores such that the sum of scores of any $t$ items is larger than the sum of scores of any $t-1$ items. We then compute the top-1 maximal combinations and check if the result contains a complete combination. If so, the join of the given relations is not empty. Otherwise, the join is empty.

In the worst case, the complexity of the problem is exponential, i.e. $O(m^t)$. However, the average-case complexity may be much lower. Thus, we analyze three different scenarios and compare the complexity of the three algorithms in these particular cases.

### 3.4.1 No Associations

The first case we consider is when any two items, of different lists, are not join consistent. Obviously, in such case, every combination contains at most one non-null item. The question is how well the algorithms deal with such case.

**VNL.** If VNL should compute only complete answers, it performs in this case only two nested loops, because every two items from the first two lists fail the association test. The time complexity is, thus, $O(m^2)$. When computing the maximal answers, we have $m$ items of the most outer list that for each one of them the loop should continue in all the other $t-1$ lists till reaching the null values (reading $m$ items in each list). This requires $m(t-1)m$ iterations. For the null item of the first list, the $m$ items of the second list will only be associated with nulls of the following lists. This requires $m(t-2)m$ iterations. This continues for the null items in all the lists, and the total number of iterations required is $O(\sum_{i=1}^{t-1}(m(t-i)m)) = O(m^2\frac{(t-1)(t-2)}{2}) = O(m^2t^2)$. Adding items to the heap has $O(kt)$ time complexity, because of the need to check whether the added combination has a subsuming combination already in the heap. So, the total complexity is $O(m^2t^3k)$.

**HNL.** If HNL should compute only complete answers, it iterates over the $tm$ items and for each one checks whether there are join consistent items in one of the other lists. This requires iterating over $m$ items, and thus, the total number of iterations is $O(tm^2)$. When computing the maximal combinations, the algorithm iterates over the lists, without adding anything to the heap, till reaching the null values. Then, it applies for each null value the VNL algorithm for $t$ lists of size $m$. It does that for $t$ null values. Thus, the time complexity is $t$ times the complexity of VNL, i.e., $O(m^2t^4k)$.

**TMax** The TMax Algorithm reads all the $tm$ items. Since there are no associations, TMC contains only singletons. Each item that has been read is checked against every tuple that is in the TMC. Checking an
item against a tuple has $O(t)$ time complexity and adding a tuple to the heap has $O(kt)$ time complexity. Note that for every new item that has been read, only one new tuple is created and added to the heap, so there is no need to remove duplicates or subsets from the set $C$ (Line 17). Thus, the time complexity is $O((mt)((mt)t + kt)) = O(m^2t^3)$ (assuming $k < mt$).

### 3.4.2 Everything is Associated

We consider the case where every two items are associated. That is, every pair of items are join consistent and can be combined.

Obviously, only the first $k$ items of each list should be read because some (or all) of these items will construct the top-$k$ combinations.

**VNL** Each loop will have at most $k$ iterations, because the potential score of any combination with the $(k + 1)$-st item of any list is lower than the top-$k$ combinations. In every iteration, we check if the current tuple is a combination (this has $O(t^2)$ time complexity), and in some cases add it to the top-$k$ heap (addition to the heap has $O(\log k)$ time complexity, because there is no need to check for subsumption when all the combinations in the heap are complete). Therefore, the time complexity is $O(k^t(t^2 + \log k))$.

**HNL** The algorithm will need to reach the $k$-th item of each list. For each item among the $kt$ visited items we will apply VNL with complexity $O(k^{t-1}(t^2 + \log k))$. Thus, the overall complexity is of $O(tk^t(t^2 + \log k))$.

**TMax** At most $k$ items of each list are read, so there are $O(kt)$ iterations. For each item that the algorithm reads, it scans TMC, and for every combination in TMC, it checks which of the items of this combination are associated with the processed item (in $O(t)$ running time). If a new combination is constructed, it is added to the top-$k$ heap (in $O(\log k)$ running time because all the combinations are complete in this case, thus there is no need to check for subsumption). The size of TMC is $O(k^t)$, because it contains only complete combinations and reads no more than $k$ items from each list (except for the beginning, when the first row is read). Therefore, the time complexity is $O(ktk^t\cdot(t + \log k)) = O(k^{t+1}t(t + \log k))$.

### 3.4.3 One-to-one Associations

We now assume that each item participates in exactly one complete combination and is not associated with any other item. (Note that this case has some similarity to the problem of optimal aggregation in middleware [30].) In this case, there are $m$ complete combinations because each item can participate in at most one complete combination.
Table 3.1: Summary of the complexity analysis

<table>
<thead>
<tr>
<th></th>
<th>No associations</th>
<th>Everything is associated</th>
<th>One-to-one associations</th>
</tr>
</thead>
<tbody>
<tr>
<td>VNL</td>
<td>$O(m^2t^3k)$</td>
<td>$O(k^t(t^2 + \log k))$</td>
<td>$O(m^2t^2 + m \log k)$</td>
</tr>
<tr>
<td>HNL</td>
<td>$O(m^2t^4k)$</td>
<td>$O(tk^t(t^2 + \log k))$</td>
<td>$O(m^2t^2 + m \log k)$</td>
</tr>
<tr>
<td>TMax</td>
<td>$O(m^2t^3)$</td>
<td>$O(tk^{t+1}(t + \log k))$</td>
<td>$O(m^2t^2 + m \log k)$</td>
</tr>
</tbody>
</table>

* Analysis parameters: $m$ is the list length, $t$ is the number of lists, and $k$ is the number of combinations in the output.

**VNL.** First we consider the complexity of computing only the complete combinations (without adding null values to the lists). For every item of the first list (the list of the most outer loop), the algorithm iterates through the second list until it finds an item to which the first item is associated (it does not apply the inner loops because of Rule 2 that was presented for this algorithm). When finding an associated item, it starts the inner loop through the third list, and so on. Therefore, for every item of the first list there are $O(m(t-1))$ iterations of inner loops. In every such iteration there are $O(t^2)$ association checks. There are at most $m$ combinations, each may be added to the top-k heap. Thus, the heap management has $O(m \log k)$ time complexity. Therefore, the time complexity for this case is $O(m^2(t-1)t^2 + m \log k) = O(m^2t^3 + m \log k)$.

Adding null values, in this case, does not increase the number of iterations or the complexity of adding a combination to the heap (since only complete combinations are added). Thus, the complexity is the same.

**HNL.** The algorithm will apply VNL $mt$ times, each time when the most outer loop has only one iteration [item]. So, the complexity is $O(mt(mt^3) + m \log k) = O(m^2t^4 + m3 \log k)$.

**TMax** The TMax algorithm reads all the items by $mt$ iterations. The size of TMC is bounded by $m$, since every item can participate in at most one maximal combination. Recall that for each item being read, the algorithm tries to extend all the partial combinations that are stored in TMC. Each such extension requires testing associations, which has $O(t)$ time complexity. In addition, inserting the $m$ combinations into the top-k heap has a complexity of $O(m \log k)$. Therefore, the time complexity is $O(m^2t^2 + m \log k)$.

### 3.4.4 Conclusions from the Analysis

The analysis shows that VNL is more efficient than HNL in the cases we examined. In particular, there is a factor of $t$ between their complexities. TMax has the worst time complexity when there are many associations between items, but it has the lowest time complexity when there are a few associations or no associations at all. Note that VNL and HNL require $O(t + kt)$ space in addition to the input data, whereas TMax requires memory for storing the data structures (the TMC).
3.5 Instance Optimality and approximation

Typically, top-\(k\) algorithms, including top-\(k\) join algorithms, have been evaluated by their access cost, i.e., the number of items read by the algorithm. Moreover, some algorithms [30, 43, 65] were shown to be instance optimal with respect to their access cost. Instance optimality, defined by Fagin et al. [30], corresponds to optimality in every instance, as opposed to just the worst case or the average case. An algorithm \(B\) is instance optimal over a class of algorithms \(A\) and a class of databases \(D\), if for every algorithm \(A\) in \(A\) and every database \(D\) in \(D\), \(\text{cost}(B, D) = O(\text{cost}(A, D))\), for some cost function \(\text{cost}\). Typically, the cost function is linearly proportional to the number of items being processed (accessed).

When evaluating algorithms for finding the top-\(k\) maximal answers, as opposed to complete answers, instance optimality should not be the goal. In fact, as the following proposition shows, if the top-\(k\) maximal answers contain at least one partial combination, then any sequential-access algorithm, i.e., an algorithm that reads the items in each list sequentially, must read all the items of at least one list. This means that when limiting the set of instances to those that contain a partial combination in their top-\(k\) maximal answer, every sequential-access algorithm is instance optimal\(^2\).

**Proposition 3.4.** Given a combined search query \(Q\) and an integer \(k\), suppose there exists a partial combination \(C\) in the top-\(k\) maximal answers of \(Q\). Let \(L_i\) be one of the lists for which no item is included in \(C\). Then any sequential-access algorithm for top-\(k\) maximal answers must read all the items of list \(L_i\).

**Proof.** To reach a contradiction, we assume there exists an algorithm \(A\) that skips one item from \(L_i\), say item \(o\). We will prove that there exists a different query \(Q'\), on which \(A\) errs. Let \(Q'\) be a query such that the lists of \(\text{PreAns}(Q')\) are similar to those of \(\text{PreAns}(Q)\), except for one item. Instead of item \(o\) in list \(L_i\), which is the item that \(A\) skips when running on \(\text{PreAns}(Q)\), there is an item \(o'\), which is join consistent with all the items of \(C\). This means that the only difference between \(\text{MaxJoin}(Q)\) and \(\text{MaxJoin}(Q')\) is that instead of the partial combination \(C\) in \(\text{MaxJoin}(Q)\), there is combination \(C'\), which comprises \(o'\) and all the items of \(C\). When \(A\) computes the top-\(k\) maximal answers of \(Q'\), it cannot differentiate between the two inputs \(\text{PreAns}(Q)\) and \(\text{PreAns}(Q')\), as the only difference between them is in one position that \(A\) skips, and therefore it reads exactly the same items as when running on \(\text{PreAns}(Q)\). Thus the answer of \(A\) on \(\text{PreAns}(Q')\) is \(\text{MaxJoin}(Q)\) and not \(\text{MaxJoin}(Q')\), which is wrong. \(\square\)

Given the limitations entailed in finding top-\(k\) maximal answers with respect to access cost, an approximate answer may sometimes be sufficient. Based on the definition of Fagin et al. [30], a \(\theta\)-approximation (\(\theta > 1\)) to the top-\(k\) maximal answers is a collection \(S\) of \(k\) (possibly partial) combina-
tions such that for each pair of combinations \( C_{\text{answer}} \in S \) and \( C_{\text{other}} \notin S \), holds

\[
\theta \cdot \text{score}(C_{\text{answer}}) \geq \text{score}(C_{\text{other}})
\]

The most important advantage of the TMax algorithm, is that it can be easily modified to provide, at any time during its computation, its current top-\( k \) approximate answer, along with the degree \( \theta \) of the approximation to the correct answer. The user can decide, based on this information, whether to stop the computation or proceed to a better approximation. Therefore, TMax is able to return an approximate answer without reading all the items, while the other algorithms cannot do so, because they may not find any \( k \) combinations to return (hence \( \theta = \infty \)) without reading at least one null item, and the null is reached only after the entire corresponding list has been read.

We now explain how to compute \( \theta \) when the algorithm is being stopped when reaching items \( o^1, \ldots, o^t \) in lists \( L_1, \ldots, L_t \), respectively. Suppose \( C \) is a combination in \( TMC \). The completion bound of \( C \) is a tuple \( \bar{C} \) that is produced by replacing every null item of \( C \) by an item among \( \{o^1, \ldots, o^t\} \), in correspondence to the position of the null. For instance, a null in the \( i \)-th position will be replaced by \( o^i \). Any combination that will be produced from \( C \) by replacing the null values with items would have a score that is at most \( \text{score}(\bar{C}) \), because as we progress in the lists, item scores decrease. Let \( \mathcal{B} = \{\bar{C} \mid C \in TMC\} \) be the completion bounds of the elements of \( TMC \) and let \( M_b = \max\{\text{score}(C) \mid C \in \mathcal{B}\} \) be the maximal score among the scores of the completion bounds of the combinations in \( TMC \). Then, we define

\[
\theta = \frac{M_b}{\text{MinScore}(H)}
\]

where \( \text{MinScore}(H) \) is the lowest score among the scores of the combinations in the heap \( H \). Note that as the algorithm progresses, the scores of the items \( o^1, \ldots, o^t \) decrease (actually, we replace these items with items that have lower scores), thus \( M_b \) declines while \( \text{MinScore}(H) \) can only grow, and accordingly, \( \theta \) becomes smaller.

We need to show that the desired approximation is achieved for the defined \( \theta \). There are two cases to consider. First, for every combination \( C_{\text{other}} \) that was inserted into \( TMC \) and does not appear in the heap \( H \), \( \text{score}(C_{\text{answer}}) \geq \text{MinScore}(H) \geq \text{score}(C_{\text{other}}) \), because if \( \text{MinScore}(H) < \text{score}(C_{\text{other}}) \) had been true, \( C_{\text{other}} \) would have been in \( H \). Second, for a combination \( C_{\text{other}} \) that has not been inserted
into $TMC$, $score(C_{other}) \leq M_b$, thus,

$$\theta \cdot score(C_{answer}) = \frac{M_b}{\text{MinScore}(H)} \cdot score(C_{answer})$$

(3.1)

$$\geq \frac{score(C_{other})}{\text{MinScore}(H)} \cdot score(C_{answer})$$

(3.2)

$$= score(C_{other}) \cdot \frac{score(C_{answer})}{\text{MinScore}(H)}$$

(3.3)

$$\geq score(C_{other})$$

(3.4)

where the move from (3.3) to (3.4) follows from the fact that for every combination $C_{answer}$ in $H$, $score(C_{answer}) \geq \text{MinScore}(H)$, i.e., $\frac{score(C_{answer})}{\text{MinScore}(H)} \geq 1$.

To show optimality of $TMax$ we consider the class $A$ of all the algorithms that compute the $\theta$-approximation to the top-$k$ answer by reading the lists $L_1, \ldots, L_t$ sequentially. $TMax$ is instance optimal if for every given lists $L_1, \ldots, L_t$ and every algorithm $A \in A$ holds $cost(TMax) = O(cost(A))$, when Algorithm $A$ and $TMax$ are evaluated over the given lists [29].

When given an approximation parameter $\theta$, the approximated version of $TMax$ reads items till $\frac{M_b}{\text{MinScore}(H)} \leq \theta$. The following proposition states that the approximated version of $TMax$ is instance optimal.

**Proposition 3.5.** Given $\theta$, let $A$ be the class of all algorithms that compute correctly the $\theta$-approximation to the top-$k$ maximal answers over given ranked lists, by reading the lists sequentially. Then, $TMax$ is instance optimal over $A$ and the class of all sets of ranked lists $L_1, \ldots, L_t$.

**Proof.** To simplify the proof, we assume that $TMax$ does not apply any optimization for early termination of the iterations through the lists. Obviously, if $TMax$ is instance optimal over $A$ without optimizations, it is also is instance optimal over $A$ with optimizations because optimizations only reduce the number of processed items.

Suppose for some given lists $L_1, \ldots, L_t$, $TMax$ reaches the $r$-th row in some list and then terminates. Note that it is possible $TMax$ stopped processing items from various lists in the $(r - 1)$-st row. Consider an algorithm $A \in A$. We need to show that $cost(TMax) = O(cost(A))$. There are two cases to examine.

**Case 1.** Suppose there is a list $L_j$ such that $A$ reads all the items that $TMax$ reads from this list. In this case, $A$ reads at least $r - 1$ items and $TMax$ reads at most $t \cdot r$ items (because there are $t$ lists and it reads no more than $r$ rows). So, $cost(TMax) \leq t \cdot r = t \cdot (r - 1) + t \leq t \cdot cost(A) + t$. Thus, $cost(TMax) = O(cost(A))$.

**Case 2.** Suppose $A$ reads the list sequentially but does not reach the $(r - 1)$-st row of any list. We show that this leads to a contradiction.
Let the lowest (last) items TMax reads be \( o_1, \ldots, o_t \). These items are either on the \( r \)-th row or on the \((r - 1)\)-st row. Thus, \( A \) does not read them, and the \( \theta \)-approximation to the top-k answer that \( A \) computes does not include them.

Before TMax starts reading \( o_1, \ldots, o_t \), there is a partial combination \( P \in TMC \) such that for the completion \( \bar{P} \), holds \( \frac{\text{score}(\bar{P})}{\text{MinScore}(H)} > \theta \). Otherwise, TMax would have terminated at this point.

Let \( o'_1, \ldots, o'_t \) be the items that precede \( o_1, \ldots, o_t \) in the lists, respectively. That is, if \( o_i \) is the \( r \)-th \([(r - 1)\)-st\)] item of \( L_i \) then \( o'_i \) is the \((r - 1)\)-st \([(r - 2)\)-nd\)] item of \( L_i \), for \( 1 \leq i \leq t \).

We define \( t \) new items \( o'_1, \ldots, o'_t \), and we change the lists \( L_1, \ldots, L_t \) by replacing the items \( o_1, \ldots, o_t \) with the new items \( o'_1, \ldots, o'_t \), respectively. The new items \( o'_1, \ldots, o'_t \) satisfy the following two conditions.

1. Every item among \( o'_1, \ldots, o'_t \) is join consistent with any item of the other lists.

2. \( \text{score}(o'_j) = \text{score}(o'_j) - \varepsilon \), for all \( 1 \leq j \leq t \), where

\[
\varepsilon = \frac{\text{score}(\bar{P}) - \theta \text{MinScore}(H)}{2t}
\]

Note that from \( \frac{\text{score}(\bar{P})}{\text{MinScore}(H)} > \theta \) follows \( \varepsilon > 0 \). We denote by \( L'_1, \ldots, L'_t \) the modified lists.

Now, let \( \bar{P}' \) be the completion of \( P \) with items of \( o'_1, \ldots, o'_t \). This completion is join consistent because of the way we defined \( o'_1, \ldots, o'_t \). The following holds.

\[
\text{score}(\bar{P}') \geq \text{score}(\bar{P}) - \sum_{j=1}^{t} (\text{score}(o'_j) - \text{score}(o'_j)) \quad (3.5)
\]

\[
= \text{score}(\bar{P}) - t \cdot \varepsilon \quad (3.6)
\]

\[
= \text{score}(\bar{P}) - t \cdot \frac{\text{score}(\bar{P}) - \theta \text{MinScore}(H)}{2t} \quad (3.7)
\]

\[
= \text{score}(\bar{P}) - \frac{\text{score}(\bar{P}) - \theta \text{MinScore}(H)}{2} \quad (3.8)
\]

\[
= \frac{\text{score}(\bar{P}) + \theta \text{MinScore}(H)}{2} \quad (3.9)
\]

\[
> \frac{\theta \text{MinScore}(H) + \theta \text{MinScore}(H)}{2} \quad (3.10)
\]

\[
= \theta \text{MinScore}(H) \quad (3.11)
\]

Line 3.5 and Line 3.6 follow from the fact that the completion \( \bar{P}' \) is different from the completion of \( P \) in at most \( t \) items and the difference for each pair of items is \( \varepsilon \). Line 3.10 follows from \( \frac{\text{score}(\bar{P})}{\text{MinScore}(H)} > \theta \).

Since the items that \( A \) reads from \( L_1, \ldots, L_t \) remain unchanged in \( L'_1, \ldots, L'_t \), the answer computed by \( A \) over \( L'_1, \ldots, L'_t \) is equal to the answer over \( L_1, \ldots, L_t \). Let \( C_i \) be the combination with the lowest score in the answer computed by \( A \). Then \( \text{score}(C_i) \leq \text{MinScore}(H) \), because \( \text{MinScore}(H) \) is the
The lowest score of a combination among the top-k maximal answers computed by TMax after it processed a superset of the items A read. (Increasing the number of processed items can only increase the lowest score.)

The combination $\bar{P}^*$ is not in the answer computed by A, because A does not reach the $(r-1)$-st and the $r$-th rows. However, equations 3.5 – 3.11, above, show that $\text{score}(\bar{P}^*) > \theta \text{MinScore}(H)$. Combining this with $\text{MinScore}(H) \geq \theta \text{score}(C_l)$ provides $\text{score}(\bar{P}^*) > \theta \text{score}(C_l)$. This contradicts the fact that A computes a $\theta$-approximation to the answer.

Since Case 2 above leads to a contradiction, only Case 1 holds. This shows that TMax is instance optimal over the class A of algorithms that compute $\theta$-approximation to the top-k maximal answers.

\[ \square \]

### 3.6 The Hybrid Approach (VHNL)

Consider the first two algorithms VNL and HNL that we have discussed and analyzed so far. VNL is more efficient than HNL when the percentage of associations is low, i.e., when there are relatively few join consistent pairs. The analysis shows that the complexity of HNL is higher than the complexity of VNL at a factor of $t$, where $t$ is the number of lists. HNL is more efficient than VNL when the percentage of associations is high. The reason to this is that VNL may read, in such case, many more items than HNL in its inner loops.

The hybrid approach tries to combine the benefits of both algorithms to produce an algorithm that is at least as efficient as HNL in the case where there are many associations between items, and at least as efficient as VNL when there are a few associations between items.

Algorithm VHNL is a hybrid between VNL and HNL, and works as follows. It computes a value $h$, iterates over the $h$ first elements of each list using VNL and then continues by iterating over the lists horizontally using HNL, starting from the $(h + 1)$-st row. Note that when $h = 0$, the hybrid algorithm VHNL immediately applies HNL, while for $h \geq m$, where $m$ is the number of items in the longest list, VHNL calls only to VNL, without switching to HNL.

Our goal is to choose $h$ such that VHNL will outperform both VNL and HNL. We do that as follows. Given the lists $L_1, \ldots, L_t$, we denote by $P_{ij}$ the probability that a pair $(o_i, o_j)$, of uniformly selected items $o_i \in L_i$ and $o_j \in L_j$, are associated. That is, if $A_{ij}$ is the set of associated pairs of items in $L_i \times L_j$, then

$$P_{ij} = \frac{|A_{ij}|}{|L_i \times L_j|}.$$  

The probability that some randomly chosen tuple among the tuples of $L_1 \times L_2 \times \cdots \times L_t$ is a combination is

$$P_C = \Pi_{1 \leq i < j \leq t}(P_{ij}).$$
Algorithm 3.4 VHNL.

Input: \( L_1, \ldots, L_t, k \)  
Output: Top-k maximal combinations

1: \( H := \emptyset \)  \hspace{1cm} \triangleright \text{a heap of size } k 
2: Compute \( h \) according to Equation 3.12 
3: For \( i = 1, \ldots, t \) let \( V_i \) be the top \( h \) items in \( L_i \) 
4: Compute \( \text{VNL}(V_1, \ldots, V_t, k, H) \)  
5: Compute \( \text{HNL}(L_1, \ldots, L_t, k, H) \) starting from row \( h + 1 \) 
6: return \( H \)

Figure 3.2: Effect of association degree \((m = 1000, t = 4, k = 100)\): running time.

Thus, \( h \) items from each list are expected to produce \( h^t \cdot P_C \) combinations. Since \( k \) combinations fill the top-k heap, we take \( k = h^t \cdot P_C \); and we compute \( h \) as follows:

\[
h = \sqrt[3]{\frac{k}{P_C}} \tag{3.12}
\]

Algorithm 3.4 presents VHNL. It computes \( h \) using Equation 3.12, and applies \( \text{VNL} \) and \( \text{HNL} \) accordingly. The \( P_{ij} \)'s probabilities for the computation of \( h \) are either known a priori or being estimated using sampling.

**Proposition 3.6.** Given a combined search query \( Q \) and an integer \( k \), VHNL returns the top-\( k \) maximal answers of \( Q \), when \(|\text{MaxJoin}(Q)| \geq k\), and \( \text{MaxJoin}(Q) \), otherwise.
3.7 Experimental Evaluation

We conducted a variety of experiments to evaluate the performance of the presented algorithms. In what follows, we first describe the experimental methodology and then present the results of the evaluation.

3.7.1 Methodology

The experiments were conducted on an Intel Core i5 2.53GHz machine with 4GB RAM, running the Windows 7 operating system. The algorithms were implemented in Java. In order to be able to vary parameters on demand we generated and utilized synthetic data sets. We also tested and demonstrated our algorithms on real-world data. We next describe the data sets and the evaluation metrics we use.

**Synthetic data.** Our synthetic data sets were generated as follows. The following parameters are given as input: the number of lists, their lengths, the range of scores of each list and their distribution (uniform or normal), and the association degree, i.e., the probability that two items are joined. Based on these parameters we generated a synthetic pre-answer to a virtual arbitrary combined search query. Specifically, we generate synthetic items, with scores, for each list, and association lookup tables. The score is chosen randomly in the specified range and from the specified distribution, and the value of the join condition (association) between every two items is set to be true / false according to the specified probability (the association degree parameter).

The parameters $m$ (i.e., list length), $t$ (i.e., number of lists), $k$ (i.e., number of desired outputs), and the association degree $p$ are specified for each experiment over the synthetic data. The default values are

![Figure 3.3: Effect of association degree ($m = 1000$, $t = 4$, $k = 100$): access cost.](image-url)
$m = 1000$, $t = 4$, $k = 100$, and $p = 2\%$. The score of a combination is the sum of scores of the joined items. Note that ordinary search engines typically return, at most, a few hundreds of items, and hence, $m = 1000$ represents relatively long lists.

**Real data.** The real-world data we used was retrieved from Yahoo! Local\(^3\) using the Local Search web services. We sent four queries to Yahoo! Local, searching for hotels, Italian restaurants, steak houses and Irish pubs, all in Boston. The number of results received for the four queries was: 240 hotels, 240 Italian restaurants, 240 steak houses and 189 Irish pubs. The results of each query were received in the order of their relevance score, and several attributes were provided, such as latitude, longitude and average rating. We used a simple join condition which is based on the distance between every two locations. We join two items if the distance between them is at most $400\, m$. The score of a combination is the sum of scores of the items joined.

**Evaluation metrics.** To evaluate the performance of the algorithms, we measured, in addition to the running times (which are measured in milliseconds), the number of list items scanned ($\#\text{ItemsReads}$), and the number of pairwise association checks ($\#\text{BinJoins}$). The first measure is the access cost of the algorithm, while the second corresponds to the join computational cost.

\(^3\)http://local.yahoo.com/
3.7.2 Results

In the following experiments, the scores in every list are distributed uniformly in the range [0,100].

The first experiment studies the effect of the association degree on the performance of the algorithms. Figure 3.2, Figure 3.3 and Figure 3.4 show the performance of the algorithms over synthetic data (with default parameters), as a function of the association degree between items. When the association degree is low (up to 2%), all algorithms read the entire input, which is expected as there are relatively few combinations, and the top-k ones contain nulls. However, there is a significant difference in the join cost of the algorithms. VNL is the most efficient, both in running time and in join cost, while HNL performs the largest number of join checks. These findings match the analytic evaluation, and are due to the holistic processing VNL does. In addition, TMax, too, is better than HNL in terms of binjoins (number of binary join tests), because when there are very few associations, the maintenance of the maximal-combination data structure saves many redundant checks, but costs a lot in running time. We would like to comment that for low association degrees, even when there are dozens of complete combinations, the top-10 answer (and obviously for larger values of k) contains partial combinations. As the association degree increases, and the top-k results are located higher in the lists, the access cost of all algorithms decreases. However, in this case HNL outperforms VNL. It reads less items than VNL and performs less join checks than both VNL and TMax. VHNL is optimal in all degrees of association. This emphasizes the importance of estimating the depth in the lists over which VNL should be executed.

The second set of experiments evaluates the performance of the algorithms as we vary the desired number of outputs (i.e., k). Figure 3.7 shows the results over real data as a function of k. All algorithms
have about the same running time (a few milliseconds), therefore only access cost and join cost are presented. TMax has the lowest access cost, and VNL the highest. With respect to join cost, HNL outperforms VNL, but TMax is the worst. As expected, all costs increase moderately when \( k \) is increased. VHNL, again, provides better results than VNL and HNL. We performed the same experiment over synthetic data with \( p = 2\% \). Figure 3.6 shows the running times and the join costs as a function of \( k \). In this case, due to the low association degree, all algorithms read the entire input (even for low values of \( k \)), and therefore access costs are not presented. VNL has the lowest running time and join cost. TMax has the highest running time but lower join cost than HNL. The interesting result is w.r.t. VHNL. When \( k \) is at most 60, the value of \( h \) is lower than the lists length, which means that HNL is executed over the low items in the lists. The effect is that VHNL performs better than HNL but worse than VNL. When \( k \) is large enough to make \( h \) exceed the lists length, only VNL is executed and the performance is the highest. This emphasizes the large overhead of HNL compared to VNL when processing the same items. The next experiment demonstrates the effect of \( h \) more clearly. As to the effect of \( k \), it causes a minor increase in the join cost. This is expected due to the high access cost, i.e. since all items are read anyhow, the overhead of maintaining a larger priority queue is very low.
The third experiment studies the effect of $h$ on the performance of VHNL. Figure 3.5 shows the join cost of VHNL as we vary the value of $h$ over synthetic data with $p = 5\%$. As the figure shows, in this setting, HNL performs better than VNL (represented by the two endpoints). However, it turns out that there exists an $h$ such that VHNL outperforms HNL. When running HNL over the same data, we found that it terminates after processing about 400 rows of items. Thus, intuitively, one would expect $h = 400$ to be the optimal point for VHNL. However, the optimal $h$ is much smaller (240). The explanation to this is that the optimal $h$ is the number of rows required to fill the top-k heap and reach a low enough threshold, using VNL. After that, HNL can proceed efficiently. Note that for this data, according to Equation 3.12, the computed value of $h$ is 189, which is near the optimal value.

3.8 Related Work

The problem of how to compute the top-$k$ complete combinations of ranked lists has been studied mainly under the goal of minimizing the number of accesses to (reading of) items from the lists. Two different cases were studied. One case is when all the lists contain the same set of objects, and an object may
receive different scores in different lists. The goal is to find the top-$k$ objects with the highest combined score. The threshold algorithm (TA) has been proposed as an instance-optimal solution for this case [29].

In the other case, each list represents a relation. A given join condition specifies which tuples can be combined. This case was studied in the context of efficient evaluation of top-$k$ queries in relational database systems [43, 44, 71], and in this context it has been shown how to minimize the number of accesses for reading of items from the lists [59, 65]. Answering top-$k$ queries over probabilistic datasets was studied by Ré et al. [62] and by Soliman et al. [71]. An extensive survey by Ilyas et al. [45] provides a comparison of the different top-$k$ algorithms.

A more general context is combined search, which is a new paradigm for responding to complex queries over multiple, diverse yet related, domains [16]. It consists of understanding the underlying domains, building appropriate subqueries to each domain specific source, and combining answers from each source to build a complete answer that is meaningful for the user. Braga et al. [12] looked into the problem of how to join heterogeneous search engines, and they propose several execution strategies. Braga et al. [13] presented an overall framework for mapping a multi-domain query into a query plan, i.e., a scheduling of service invocations. These papers, however, have not considered the problem of maximal join.

The algorithm HNL that we present and discuss in this thesis is a variation of the algorithm of Ilyas et al. [43], which was designed for computing the join of ranked lists, for the case where the result contains only complete combinations. Their method is based on using pipelining and binary join operators, for computing equijoin of ranked lists. This approach has been further investigated by Schnaitter and Polyzotis [65]. In their proposed algorithms the lists are being scanned and the items are inserted into hash tables where the hash keys are the join attributes. This makes the search for pairs of join-consistent items efficient. However, in cases where the join is not an equijoin, such as in Example 3.1 where items are joined based on proximity rather than equality, such methods cannot be used. Thus, we had to slightly change the algorithm and use the general pattern—Pull Bound Rank Join—for a join of ranked lists, that was suggested and investigated by Schnaitter and Polyzotis [65]. They have studied the use of bounds for achieving instance-optimal algorithms, for the computation of complete answers. The algorithm HNL that we present and discuss in this thesis is an adaptation of their general algorithm to computation of maximal answers (and thus, handling incomplete information).

Over the years, different approaches for dealing with incomplete data were proposed. Some researchers suggested dealing with missing values by considering each valid completion of the missing values as a possible world and they investigate the problem of computing answers that are true in all possible worlds, e.g., see [1, 55, 58]. Efficiently managing sets of possible worlds was studied in [4–6]. Some papers considered the problem of dealing with null values that arise during a join that should not discard any input tuple—even when a tuple from one relation cannot be joined with any tuple from
another relation. In that context, the problem of computing partial or maximal answers, rather than complete answers, was studied. Traditionally, the outerjoin operator has been used for joining tuples without loss of information [28, 33]. However, outerjoins are not associative. Thus, in a join of several relations the order of the join may affect the result. Hence, Galiano-Legaria proposed full disjunctions as a clear semantics for maximally joining tuples [34]. Following papers showed how to compute full disjunctions efficiently [25, 27, 53, 61]. The problem being investigated in this thesis is, however, different from the evaluation of full disjunctions in the following two points. First, in this thesis the lists are ranked and only the top-k answers should be returned. Secondly, in a full disjunction, there is a requirement that every set of joined tuples should be from a connected subgraph of the schema. This is not a difficulty in our setting.
Chapter 4

Ranking Joined Search Results

4.1 Introduction

Answering complex search tasks, such as those in Example 1.1 and Example 3.1, is by joining the ranked lists that are the answers to the basic search queries. The result of a join is a set of combinations, where each combination is a set of related items—comprising a single item from each list, e.g., a combination in Example 1.1 is a triplet that comprises a hotel and two restaurants. Combinations are ranked according to the ranking scores of the items they comprise. A combination $C$ is optimal for an item $i$ if $C$ is the most highly ranked combination among the combinations that contain $i$.

One of the downsides of joining query answers is that the result can include many duplicate appearances of items. For instance, a hotel in Example 1.1 that can be joined with five restaurants that serve business lunch and three restaurants that serve an appropriate supper may appear fifteen times in the join result. Consequently, a join result can be too large to be useful—in fact, the number of combinations can be exponential in the size of the search answers, where $t$ ranked lists of size $n$ can produce up to $n^t$ combinations. Moreover, since any join condition can be used, the answer can be any subset of these $n^t$ combinations. In some cases the entire result can be succinctly presented to the user. However, in many cases there is too much information in the entire join result and most of this information has low relevancy, e.g., a combination of a hotel and two restaurants where all three have relatively low scores in the scenario of Example 1.1. Dealing with such cases requires to decrease the amount of information presented to the user.

To decrease the amount of information presented to the user, we need to decrease the size of the result. This is being done by discarding some of the combinations. However, discarding combinations should be done carefully, in order not to remove too many combinations.

The issue of discarding combinations leads to a conflict. On the one hand, for queries to be effective, their results should be such that users will easily find a combination that satisfies them. Hence, having too many combinations in the result reduces the effectiveness of queries. On the other hand, to enhance
user experience, the user should be given several options to choose from. Discarding too many combinations may diminish the diversity of the result and may even reduce the effectiveness of the query by dismissing all the combinations that satisfy the user. The main goal of this work is to show how to select \( k \) combinations of the join, for some given \( k \), to yield effective and diverse query results.

Three common approaches to decrease the size of a join result are (1) returning merely the top-\( k \) combinations, (2) applying the skyline operator, and (3) forcing diversity on the results by not allowing items to appear more than once in the result. All these approaches have drawbacks that prevent them from providing an effective and genuinely diverse result.

Merely taking the top-\( k \) combinations gives preference to highly ranked items, and thus, typically the result has low diversity. The skyline operator is too selective. It discards too many results, and as such, in many cases it dismisses all the combinations that satisfy the user. So, the result of the skyline operator has low effectiveness and low diversity. The third approach of forcing diversity by not allowing repetition of items significantly limits the choice options of users. When each item can appear in only one combination, many combinations will include items with low ranking scores and will be ignored by the user. Hence, many items that would have been considered relevant if they appeared in their optimal combination, are considered irrelevant in the non-optimal combination that contains them.

Note that true diversity requires not only to return many different items, but also to return them as part of satisfying combinations. For instance, in the scenario of Example 1.1, it is desired that the answer will include several hotels to provide a choice to the businessperson, however, for the diversity to be genuine, the hotels should be combined with restaurants that have high ranks. In other words, we only consider a hotel as an option if it appears in a combination that may be selected by the user. Genuine diversity means providing to the user several options to choose from and not just technically including many items in the result.

In this thesis, we investigate an approach of choosing combinations according to the following two goals.

1. **Maximize the coverage.** Items that appear in the results of the basic search queries are considered relevant. The goal is to include in the result as many as possible appearances of distinct relevant items in their optimal combination.

2. **Maximize the optimality ratio.** The goal is to maximize the ratio of the number of appearances of items in their optimal combination to the overall number of appearances of items in the result. In other words, the goal is to minimize the number of cases where an item appears in a non-optimal combination.

Merely maximizing the coverage can be done naively by not discarding any combination. Merely maximizing the optimality ratio can be done naively by discarding all the combinations except for the single
one that is most highly ranked. Typically, not discarding combinations will result in having too many repetitions of items and too many cases where items appear in a non-optimal combination. Returning only a single combination will hide the variety of possible answers and will not offer to users several options to choose from. The main challenge is to provide good results in terms of both coverage and optimality ratio.

Maximizing the coverage and the optimality ratio is expected to provide a result that is effective and genuinely diverse, because many items will be included in the result, in their optimal combination. Consider, for instance, a case where the businessperson of Example 1.1 uses a credit card that gives a discount in three hotels out of the, say twenty, hotels returned by the search. These three hotels will not be known in advance, so it will be impossible to include the discount preference as a search condition, and yet, the businessperson would like to see these three hotels in the result, in their optimal combination. The approach of maximizing the coverage and the optimality ratio increases the chances for that to happen.

To show that maximizing the coverage and the optimality ratio increases the effectiveness and the diversity of the result, we conducted a simulation of a user study as part of our experimental evaluation. In the simulation, we generated virtual users. For each item and user, it was decided whether the item satisfies the user probabilistically, proportionally to the score of the item. Our experiments show that indeed maximizing the coverage and the optimality ratio increases the effectiveness and the diversity of the result.

4.1.1 Chapter Outline

First, in Section 4.2 we present our framework. We formally define coverage and optimality ratio. We present three new semantics that are designed to provide high coverage and high optimality ratio, in a join of search results. Algorithms for evaluation of complex search queries under the new semantics are presented in Section 4.3. The results of our experimental evaluation are given in Section 4.4, and Section 4.5 surveys related work.

4.2 Framework

In this section we present our framework, we provide formal definitions of coverage and optimality ratio and we present existing and novel semantics for complex search queries. Throughout this section, we illustrate our definitions using the following running example.

Example 4.1. Consider a “Find a Researcher” application that receives a research area and returns a list of researchers ranked according to the quality and the number of their publications. Suppose that some researcher, Alice, wants to form a research group that would submit a grant application. She
Figure 4.1: An illustration of relationships between researchers. The shapes represent two researchers who are experts in Search ($S_e$), six researchers who are experts in Harmonic Analysis ($H_e$) and six researchers who are experts in Machine Learning ($M_e$). We represent relationships as proximity, i.e., we assume here that researchers are related if and only if they are in the same cluster.

Figure 4.2: Results of the search queries $q_1$, $q_2$ and $q_3$.

<table>
<thead>
<tr>
<th>Item</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>0.95</td>
</tr>
<tr>
<td>$S_2$</td>
<td>0.93</td>
</tr>
</tbody>
</table>

needs to find three researches—one who is an expert in Search Engines, one who is an expert in Machine Learning and one who is an expert in Harmonic Analysis. To strengthen the application, she wants that every two researches among the three will either have a joint paper or work in the same institute. This is a complex search task that can be answered by posing three independent search queries to the Find a Researcher application and joining the search results according to the specified condition.

Note that the join condition in Example 4.1 is merely an example, so different join conditions can be used in other search tasks.

### 4.2.1 Search Queries and Joined Search Results

The following definitions were already presented in Section 3.2, but are repeated here to enable continuous reading.

In complex search tasks, a combined search query comprises several basic search queries and a join condition. Keyword search queries are examples of basic search queries. The basic search queries are posed over appropriate data sources and the result of each basic search query is a ranked list of items. The result of the combined search query is a subset of the join of the ranked lists, where the join is according to the given join condition and the subset of the join result is defined by the semantics of the
combined query.

The relevance score of an item in the result of a basic search query provides an indication to the importance of the item and to its pertinence to the query, in the context of the search. We denote by $score_q(o)$ the relevance score of an item $o$ that is contained in the answer to a search query $q$.

Since in this chapter we are interested in the semantics of the join of ranked lists, we assume that the input is already given as ranked lists of items, i.e., items with their relevance scores.

**Example 4.2.** Consider the scenario presented in Example 4.1. The search queries are $q_1 = \{\text{search engines}\}$, $q_2 = \{\text{harmonic analysis}\}$ and $q_3 = \{\text{machine learning}\}$. Suppose that these queries return the researchers that the relationships between them are depicted in Figure 4.1. These relationships define the join condition. The search results are ranked lists such as the lists in Figure 4.2.

In a join of ranked lists, we create combinations. Given the ranked lists $L_1, \ldots, L_t$, we can consider each list as the set of items it comprises. A combination is a $t$-tuple $C = (o^1_{i_1}, \ldots, o^t_{i_t}) \in L_1 \times \cdots \times L_t$, i.e., a tuple that consists of exactly one item from each list. The join operation is based on the given join condition. For instance, in Example 1.1, a hotel and two restaurants are joined if the sum of costs does not exceed the budget. In Example 4.1, the three specialists are joined if for each pair among them, there is a joint publication or they both work in the same institute.

When $Q$ is a combined search query with join condition $\theta$, we denote by $\text{Join}(Q)$ the set of all the combinations that are produced by joining the ranked lists that are answers to the subqueries of $Q$, i.e., all the combinations that satisfy $\theta$ among the combinations in the Cartesian product of the lists.

The score of a combination $C$, denoted $score(C)$, is the result of a monotonic function over the item scores. In particular, it can be any linear combination of the item scores, achieved by giving different weights to the different relevance scores. For simplicity, in the rest of the chapter we assume that the score of a combination is the sum of the scores of its items.

**Example 4.3.** Consider a combined search query comprising the three queries $q_1, q_2, q_3$ whose ranked lists are depicted in Figure 4.2. Let the score of a combination be the sum of the scores of its items. We assume that pairs of objects can be joined if they are both in the same cluster in Figure 4.1. Then, $(S_1, H_{11}, M_{11})$ is a combination with score 2.88 (the sum 0.95 + 0.99 + 0.94); In this scenario there are 18 combinations because there are two clusters, and in each cluster there are three specialist in Harmonic Analysis, three experts in Machine Learning and one leading researcher in the area of Search Engines, providing nine combinations per cluster.

### 4.2.2 Coverage and Optimality Ratio

Consider a combined search query $Q$. The answer to $Q$ could have been the entire join of the ranked lists $L_1, \ldots, L_t$. However, since the number of combinations in the join of the lists can be large, we take
a subset $A$ of the join as the answer to $Q$. Next, we formally define the measures that should affect the selection of this subset.

Given an item $o_{ij} \in L_j$, an optimal combination of $o_{ij}$ is a combination with the highest score among the combinations that contain $o_{ij}$. That is, if $C(o_{ij})$ are all the combinations that contain $o_{ij}$ then $C_j$ is an optimal combination of $o_{ij}$ if $C_j$ contains $o_{ij}$ and $\text{score}(C_j) \geq \text{score}(C)$ for every $C \in C(o_{ij})$. For example, $(S_1, H_{11}, M_{11})$ is the optimal combination of $S_1$, $H_{11}$ and $M_{11}$. The combination $(S_1, H_{12}, M_{11})$ is optimal for $H_{12}$ but it is not optimal for $S_1$ and $M_{11}$, because $\text{score}((S_1, H_{11}, M_{11})) = 2.88$ whereas $\text{score}((S_1, H_{12}, M_{11})) = 2.87$.

In the context of a join the relevancy of items should be determined according to the score of their optimal combination. For instance, a hotel in Example 1.1 that has a relatively high score but can only be joined to restaurants with low scores will be considered according to the score of its optimal combination and not just according to its own score. Obviously, we would like to consider each item according to its optimal combination. For instance, given a hotel that appears in a combination with high score, we do not want to consider it as irrelevant just because it appears in another combination whose score is low.

The optimality count of a combination $C$, is the number of items in $C$ for which $C$ is the optimal combination. We denote the optimality count of $C$ by $\text{OptCount}(C)$, e.g., $\text{OptCount}((S_1, H_{11}, M_{11})) = 3$ and $\text{OptCount}((S_1, H_{12}, M_{11})) = 1$. The optimality count of a subset $A \subseteq \text{Join}(Q)$ is the sum of optimality counts over the combinations of $A$, that is $\text{OptCount}(A) = \sum_{C \in A} \text{OptCount}(C)$. Note that two combinations may have the same score, and thus an item may have more than one optimal combination. We denote by $\text{OptCountDistinct}(A)$ a distinct optimality count, i.e., an optimality count where each item is counted at most once.

We now use these definitions to define coverage and optimality ratio for a given subset of combinations $A \subseteq \text{Join}(Q)$.

**Coverage.** We assume that all the items in $L_1, \ldots, L_t$ are relevant to the search (each item is relevant to the appropriate query). Thus, our goal is to provide to the user all these items, yet, as explained before, to correctly evaluate the relevance of an item it should be considered in the context of its optimal combination. Hence, coverage measures the ratio of the number of items that appear in $A$ as part of their optimal combination to the number of all the items in $\text{Join}(Q)$. That is, 

$$\text{Coverage}(A) = \frac{\text{OptCountDistinct}(A)}{\text{ItemCountDistinct}(\text{Join}(Q))}$$

where $\text{ItemCountDistinct}(\text{Join}(Q))$ is the number of distinct items in $\text{Join}(Q)$.

**Optimality Ratio.** In a join of search results, an item may appear in more than one combination. In particular, an item may appear in a non-optimal combination.

It is desired that items will appear only in their optimal combination, for two reasons. First, this
reduces duplications, i.e., cases where the same item appears in several combinations. Reducing duplications increases the effectiveness of the result and helps keeping the size of the result small while maintaining diversity. Second, having items in non-optimal combinations may mislead users. It will cause an erroneous presentation of the item with respect to the search, e.g., in Example 1.1, a user may disregard a hotel that has nearby relevant restaurants due to seeing the hotel in a non-optimal combination with other restaurants whose scores are low.

Thus, optimality ratio is the ratio of the number of optimal appearances of items in $A$ to the overall number of items in $A$. That is,

$$ OptimalityRatio(A) = \frac{OptCount(A)}{ItemCount(A)} $$

where $ItemCount(A)$ is the number of appearances of items in $A$ (with repetitions).

**Example 4.4.** Consider a set $A = \{(S_1, H_{11}, M_{11}), (S_1, H_{12}, M_{11}), (S_2, H_{23}, M_{22})\}$, of three combinations, with respect to the queries in Example 4.2. The coverage of $A$ is $\frac{2}{7}$ because in $A$, four distinct items appear in their optimal combination (the items $S_1, H_{11}, M_{11}, H_{12}$) out of the 14 items that are in the join result. The optimality ratio is $\frac{4}{9}$, because the three items $S_1, H_{11}, M_{11}$ appear in their optimal combination $(S_1, H_{11}, M_{11})$, and the item $H_{12}$ appears in its optimal combination $(S_1, H_{12}, M_{11})$. The other appearances of items in $A$ are not in their optimal combination, and there is a total of 9 appearances of items in combinations of $A$.

**Harmonic Mean.**

The harmonic mean of the coverage and the optimality ratio is:

$$ \frac{2}{\frac{1}{OptimalityRatio} + \frac{1}{Coverage}} $$

**Problem Statement.**

The problem we are trying to solve in this work is as follows. Given lists $L_1, \ldots, L_t$ and an integer $k$, find the $k$ combinations that provide the highest harmonic mean of coverage and optimality ratio.

**4.2.3 Existing Semantics**

We now present three common semantics that were suggested as methods to reduce the size of the result in a join of ranked lists, namely top-$k$, skyline and repeated-top-1. These semantics specify which combinations to choose as the answer to a combined search query, however they were not designed to provide high coverage and high optimality ratio.

**Top $k$.** Choosing the $k$ most-highly ranked combinations is a simple and common approach. However, when $k$ is small with respect to the number of items in $Join(Q)$, the coverage is expected to be low. Even
when $k$ is not small, if the join produces many combinations, there are expected to be many repetitions of items in the result, and hence, the optimality ratio is expected to be low. To see that, consider Example 4.2 and let $k$ be 3. The top-$3$ answer is $\{(S_1, H_{11}, M_{11}), (S_1, H_{12}, M_{11}), (S_1, H_{11}, M_{12})\}$, having a low diversity, in the sense that there are only combinations from one cluster, and a low optimality ratio of $\frac{5}{9}$.

**Skyline.** The **skyline operator** has been proposed as an alternative to top-$k$, where a dominance relationship is being defined for specifying the combinations of the result. Let $C' = (o_1', \ldots, o_t')$ and $C = (o_1, \ldots, o_t)$ be two combinations. We say that $C'$ dominates $C$ and write $C' \succ C$, if for all $1 \leq i \leq t$, $\text{score}(o_i') \geq \text{score}(o_i)$, and exists $1 \leq j \leq t$ such that $\text{score}(o_j') > \text{score}(o_j)$. The skyline operator returns only the combinations that are maximal with respect to dominance, i.e., combinations that have no other combination dominating them. Formally, given a query $Q$, $\text{Skyline}(Q) = \{C \mid C \in \text{Join}(Q) \text{ and } \nexists C' \in \text{Join}(Q) \text{ such that } C' \succ C\}$.

Typically, the skyline operator discards many combinations, and hence, the result has a low coverage. For instance, in Example 4.3, the skyline operator will return a single combination $(S_1, H_{11}, M_{11})$.

**Repeated-Top-1.** The Repeated-Top-1 semantics has been initially suggested as a method for providing diversity in search results (see [15]). In Repeated-Top-1, the answer is computed iteratively over the set of candidate combinations, where in each iteration, the top-1 combination is added to the answer and all the combinations that have a common item with the combination that was inserted to the answer are discarded from the set of candidate combinations. More precisely, initially let $J$ be equal to $\text{Join}(Q)$, and let $A$ be an empty set. In each iteration, the top-1 combination in $J$ is added to $A$, and any combination of $J$ that shares an item with the combination that was added to $A$ is discarded from $J$. When $J$ is empty, $A$ is the answer under the Repeated-Top-1 approach. Note that in common scenarios, Repeated-Top-1 will not provide a high optimality ratio because in order to provide diversity, many combinations that are optimal for some items are discarded and these items eventually appear in non-optimal combinations.

### 4.2.4 New Semantics

Next, we present novel semantics for combined search queries.

**ALL semantics.** Under the ALL semantics, we require that items will appear only in their optimal combination. To that end, we define a combination $C$ as completely optimal if $C$ is the optimal combination for all the items in it. The answer to a combined query under the ALL semantics is the set of all the combinations that are completely optimal. Note that the ALL semantics provides the highest possible coverage under the requirement of providing an optimality ratio equal to 1. We denote the answer to $Q$ under the ALL semantics by $\text{All}(Q)$. In the setting of Example 4.3, $\text{All}(Q) = \{(S_1, H_{11}, M_{11}), (S_2, H_{21}, M_{21})\}$.

**EXISTS Semantics.** Under the EXISTS semantics, every combination in the result is optimal for at least
one of the items that appears in it. The answer to a query consists of all such combinations. While the ALL semantics guarantees an optimality ratio of 1, the EXISTS semantics provides full coverage, i.e., guarantees a coverage that is equal to 1. That is, for any item o that appears in Join(Q), the optimal combination of o appears in the answer to Q. We denote the answer to Q under the EXISTS semantics by Exists(Q).

Consider the setting in Example 4.2. Under the EXISTS semantics, the combination (S_2, H_{21}, M_{23}) will be in the answer because it is the optimal combination of the item M_{23}. Note that item M_{23} appears in the result even though its score is relatively low. The combination (S_1, H_{12}, M_{12}) will not be in the result because it is not the optimal combination of any item.

**Optimality Rank.** The ALL and EXISTS semantics can be combined with top-k, so that for a given k, only the top-k combinations of ALL (or of EXISTS) are returned. All(Q) is frequently small, so it is more important to consider applying top-k to Exists(Q). However, returning the top-k combinations of Exists(Q) when the sort is merely by scores may yield an answer with a low optimality ratio. An alternative approach which is also based on the EXISTS semantic, is to primarily sort the combinations of Exists(Q) according to their optimality count, and then sort combinations with the same optimality count according to their scores. In this approach, in each addition of a combination to the answer, the number of cases where the combination is not optimal for items it comprises is minimal. We refer to this approach by the term OptimalityRank.

**Example 4.5.** Consider Example 4.2. The top-2 combinations under EXISTS, sorted by score, are (S_1, H_{11}, M_{11}) and (S_1, H_{12}, M_{11}). When sorting according to optimality count, the top-2 combinations are (S_1, H_{11}, M_{11}) and (S_2, H_{21}, M_{21}), because these combinations have optimality count 3 while for all the other combinations, the optimality count is smaller than 3.

One of the advantages of OptimalityRank is that if scores of combinations are distinct (i.e., there are no two combinations with exactly the same score) then applying top-k over the result of OptimalityRank provides a maximal harmonic mean of coverage and optimality ratio, compared to any set A ⊆ Join(Q) of k combinations. This is because OptimalityRank provides the highest OptCountDistinct(A) and highest OptCount(A) for sets A ⊆ Join(Q) of size k.

**Proposition 4.1.** Given lists L_1, ..., L_t and k, if combination scores are distinct, selecting the k first combinations returned by OptimalityRank provides a maximal harmonic mean of coverage and optimality ratio.

(When scores of combinations are not unique, this claim may not hold, and yet it shows that OptimalityRank is expected to always provide a high harmonic mean of coverage and optimality ratio, in comparison to all other semantics.)
4.2.5 Relationships Between the Semantics.

It is easy to see from the definitions above that any combination that is provided under the \texttt{ALL} semantics is also provided under the \texttt{EXISTS} semantics.

**Proposition 4.2.** For any combined search query $Q$, $\texttt{All}(Q) \subseteq \texttt{Exists}(Q)$.

Answers under the \texttt{EXISTS} or the \texttt{ALL} semantics are not necessarily contained in the result of the skyline operator.

**Proposition 4.3.** There exists a combined search query $Q$ such that $\texttt{All}(Q) \not\subseteq \texttt{Skyline}(Q)$.

**Proof.** Consider a combined search query where the join of the ranked lists is the set $J = \{(h_1, m_1, s_1), (h_2, m_2, s_2)\}$, such that $\text{score}(h_1) > \text{score}(h_2)$, $\text{score}(m_1) > \text{score}(m_2)$ and $\text{score}(s_1) > \text{score}(s_2)$. Then, $\text{Skyline}(Q) = \{(h_1, m_1, s_1)\}$ whereas $\text{All}(Q) = \{(h_1, m_1, s_1), (h_2, m_2, s_2)\}$. \hfill \qed

From Propositions 4.2 and 4.3 we conclude the following.

**Corollary 4.1.** There exists a search query $Q$ such that $\texttt{Exists}(Q) \not\subseteq \texttt{Skyline}(Q)$.

In Repeated-Top-1 each item can appear at most once. Under the \texttt{EXISTS} semantics, an item can be included in more than one combination. Hence, the answer of Repeated-Top-1 does not always contain the answer of \texttt{EXISTS}.

**Proposition 4.4.** There exists a query $Q$ such that $\texttt{Exists}(Q) \not\subseteq \texttt{Repeated-Top-1}(Q)$.

On the other hand, the answer of Repeated-Top-1 contains the answer under \texttt{ALL}.

**Proposition 4.5.** For any combined search query $Q$, $\texttt{All}(Q) \subseteq \texttt{Repeated-Top-1}(Q)$.

**Proof.** Let $C$ be a combination in $\texttt{All}(Q)$. Suppose that $C$ is not in $\texttt{Repeated-Top-1}(Q)$. Obviously, the items of $C$ satisfy the join condition because $C \in \texttt{All}(Q)$. So, there should be an item $o$ of $C$ that is removed in the procedure of generating the Repeated-Top-1 answer. This means that $o$ belongs to a combination whose score is higher than the score of $C$, in contradiction to having $C \in \texttt{All}(Q)$. \hfill \qed

Since Repeated-Top-1 may include combinations that are not optimal for any of their items, it is not necessarily contained in the answers to \texttt{ALL} or \texttt{EXISTS}.

**Proposition 4.6.** There exists a combined search query $Q$ such that the following hold: $\texttt{Repeated-Top-1}(Q) \not\subseteq \texttt{Exists}(Q)$ and $\texttt{Repeated-Top-1}(Q) \not\subseteq \texttt{All}(Q)$.

The \texttt{OptimalityRank} semantics is merely an ordering of \texttt{EXISTS}, so when avoiding applying top-$k$ over it, its answer is equal, as a set, to the answer under \texttt{EXISTS}, and consequently, all the relationships of \texttt{EXISTS} hold for it.
4.2.6 Imprecise Ranking and $h$-Optimality.

Usually, ranking methods merely provide an approximation of item relevancy. This is because the order they specify may be based on incomplete information or may not reflect correctly the preferences of different users. Thus, when two items have almost the same ranking score, there is no certainty that the order between them defines correctly which item is more relevant than the other. This causes the top-$k$, Skyline and ALL semantics to be somewhat imprecise. However, when the ranking scores do provide a good approximation of item relevancy, the optimality ratio remains a good measure for the effectiveness of duplication removal. More importantly, imprecise ranking is one of the motivations for the requirement of providing high coverage.

Requiring the answer to provide high coverage is one approach to deal with imprecise ranking. A more adjustable approach is based on refining the definition of optimality. We say that $C_j$ is an $h$-optimal combination of an item $o_{ij}$ if $C_j$ contains $o_{ij}$ and $C_j$ is among the top-$h$ combinations that contain $o_{ij}$.

(Note that a combination is optimal iff it is 1-optimal.)

The value of $h$ should be determined according to the accuracy of the ranking methods. Given $h$, we define the semantics $\text{ALL}_h$, $\text{EXISTS}_h$ and $\text{OptimalityRank}_h$ to be the same as in the previous sections except that we use $h$-optimal instead of 1-optimal in the definitions. That is, $\text{ALL}_h$ comprises combinations that are $h$-optimal for all their items, $\text{EXISTS}_h$ comprises combinations that contain an item for which the combination is $h$-optimal, and $\text{OptimalityRank}_h$ sorts combinations according to the number of $h$-optimal items in them.

4.3 Algorithms

Answering a combined search query $Q$ requires two steps: producing the ranked lists that are the answers to the subqueries of $Q$; and selecting the combinations of the answer according to the semantics, with or without computing the join result. In this section, we elaborate on the second step.

Throughout this section, we assume that the input consists of the ranked lists $L_1, \ldots, L_t$, produced by posing the subqueries of a combined query $Q$. For simplicity, we assume that all the lists are of length $m$.

We present three algorithms. Two of them scan and filter the entire join result, i.e., they receive a stream of combinations, sorted or unsorted, and filter it according to the desired semantics. The third algorithm computes the result set directly. The next section describes how the stream of join results is generated.
4.3.1 Producing a pipeline of $\text{Join}(Q)$

The PBRJ template [65], which was described in Section 3.3.2.1 is a general pattern for a join of ranked lists. We adopt it as our basic algorithm template, and use a simple implementation, which processes the items horizontally, and uses the same bounds suggested in [43]. Note that the algorithm has another version, used in [43], that maintains a maximum and not minimum heap, and therefore can generate a stream of combinations sorted by score, and is not limited to a fixed output size $k$. Therefore it can be used not only to find the top-k join result, but rather to generate the entire join result. Its major disadvantage is that the size of the heap is unbounded, and may become extremely high.

In our algorithms we use PBRJ in three fashions: (i) to return the top-k combinations, i.e., the $k$ most highly ranked combinations; (ii) to return the entire $\text{Join}(Q)$, where combinations are sorted by score; and (iii) to return the entire $\text{Join}(Q)$ where combinations are not sorted. Obviously, providing the combinations unsorted can be done more efficiently than when providing the combinations sorted, where every combination is delivered as soon as it is found. Therefore, no heap is required, which means the space complexity is $O(1)$.

Next, we present algorithms for computing the answer to a combined search query $Q$, under the EXISTS and ALL semantics.

4.3.2 The FilterSorted (FS) Algorithm

The FilterSorted algorithm (FS, for short), is a simple method that receives the combinations of $\text{Join}(Q)$ sorted, and filters out the combinations according to the semantics. First, we present the computation of answers under the EXISTS$_{h}$ semantics. The pseudocode appears as Algorithm 4.1.

**Algorithm 4.1** FilterSorted

**Input:** $\text{Join}(Q)$ as a sequence $C_1, \ldots, C_n$, ordered by score $h$.

**Output:** EXISTS$_{h}(Q)$.

**Data structures:** a lookup table for all items $\text{Appearances}[m][t]$ initialized by zeros.

1: $E := \emptyset$
2: for $i = 1$ to $n$ do
3:   count$_i := 0$
4:   let $C_i$ be $(o^1_i, o^2_i, \ldots, o^t_i)$
5:   for $j = 1$ to $t$ do
6:     if $\text{Appearances}[i][j] < h$ then
7:       $\text{Appearances}[i][j] := \text{Appearances}[i][j] + 1$
8:       count$_i := \text{count}_i + 1$
9:     end if
10:   end for
11: if $\text{count}_i > 0$ then
12:   $E := E \cup \{C_i\}$
13: end if
14: end for
15: return $E$
FS maintains an item lookup table, named \textit{Appearances}, for counting the appearances of seen items. Initially the counter for all items is zero, and whenever an item is seen, its counter is incremented. The input to FS is a stream $C_1, \ldots, C_n$ of sorted combinations, provided by the join algorithm. FS filters each combination as it arrives. Given a combination $C_i$, every item in $C_i$ is being searched in \textit{Appearances}. If the appearances counter is less than $h$, then this combination is $h$-optimal for this item, and the appearances counter is incremented. The optimality count of $C_i$, computed in $\text{count}_i$, is the number of items in $C_i$ for which $C_i$ is $h$-optimal. If the optimality count is greater than zero, the combination $C_i$ is added to the answer.

Because the combinations are ordered by their score, the first $h$ appearances of an item are its $h$-optimal combinations. Therefore, the optimality count of each combination can be computed immediately.

Computing the answer under the \textit{ALL} semantics is similar to the computation under the \textit{EXISTS} semantics, except that in Line 12 a combination $C_i$ is added to the answer only if its optimality count is equal to $t$. In other words, we change the condition in Line 11 to be $\text{count}_i = t$ instead of $\text{count}_i > 0$.

The FS algorithm can be pipelined to the join. This means that given a join algorithm that outputs a sorted stream of combinations, FS can process each combination on-the-fly and determine immediately whether it belongs to the result. More importantly, under both the \textit{EXISTS} and the \textit{ALL} semantics, combinations can be added to the result before the end of the join computation. Thus, this algorithms is non-blocking.

### 4.3.3 The FilterUnSorted (FUS) Algorithm

The FS algorithm requires the combinations to be provided in the order of their score. However, the initial step of generating the join stream is much more efficient when the combinations may be output without any specific order. In addition, many combinations may be discarded as soon as they are processed, based on previously seen combinations, even without seeing all higher ranked combinations. The \textit{FilterUnSorted} Algorithm (FUS, for short) receives as input a stream of combinations without any specific order (produced on-the-fly by the join algorithm), and returns only the combinations of the answer under the required semantics (\textit{ALL} or \textit{EXISTS}). The pseudocode is presented as Algorithm 4.2.

Let $C_1, \ldots, C_n$ be a sequence of combinations, provided by the join algorithm. The algorithm iterates over the sequence of combinations and maintains for each item a heap of its top-$h$ combinations (among the combinations that contain this item). Essentially, at the end of this phase, the result set under $\text{EXISTS}_h$ is the union of the combinations in all the heaps (removing duplicates).

The second phase of the algorithm computes the optimality count of relevant combinations. In this phase, the algorithm goes over the combinations that were extracted in the first phase, and it maintains a frequency counter for each combination. The counter of each combination specifies the optimality
Algorithm 4.2 FilterUnSorted

Input: $Join(Q)$ as a sequence $C_1, \ldots, C_n$, not ordered. $h$.
Output: $EXISTS_h(Q)$.

Data structure 1: a lookup table for all items $Toph[i][j]$, each entry initialized by an empty minimum heap of size $h$.

Data structure 2: a hash map $H$, that maps each combination to its frequency.

1: for $i = 1$ to $n$ do
2: let $C_i$ be $(o_{i1}^1, o_{i2}^2, \ldots, o_{it}^t)$
3: $s := score(C_i)$
4: for $j = 1$ to $t$ do
5: if $Toph[i][j].size < h$ or $Toph[i][j].minScore < s$ then
6: Add $C_i$ to $Toph[i][j]$.
7: if $Toph[i][j].size > h$ then
8: remove from $Toph[i][j]$ the combination with the smallest score
9: end if
10: end if
11: end for
12: end for
13: $H := \emptyset$
14: for $l = 1$ to $t$ do
15: for pos = 1 to $m$ do
16: for all $C$ in $Toph[l][pos]$ do
17: freq := 1
18: if $C \in H$ then
19: freq := freq + $H(C)$
20: end if
21: $H(C) := freq$
22: end for
23: end for
24: end for
25: return $H$

4.3.4 The Top-$h$-Per-Item (TopHI) Algorithm

The Top-$h$-Per-Item algorithm (TopHI, for short) evaluates the query $Q$ while skipping the step of entirely computing $Join(Q)$. Instead of computing the join, it iterates over the items of the lists $L_1, \ldots, L_t$ and for each item, it directly computes its $h$-optimal combination, as follows.

Consider the item $o_{ij}^l$ from list $L_j$. TopHI replaces the list $L_j$ by a list containing merely the item $o_{ij}^l$. Then, it applies a top-$k$ join algorithm (in our experiments we used $PBRJ$) with $k = h$, over the lists
$L_1, \ldots, L_{j-1}, \{o^j_{i_j}\}, L_{j+1}, \ldots, L_t$. The result is the $h$-optimal combinations of $o^j_{i_j}$.

TopHI iterates over all the lists, computes for each item its $h$-optimal combinations and adds the combinations to a suitable data structure. We denote this data structure by $H$.

Similarly to the creation of the $\text{Toph}[\cdot][\cdot]$ lookup table in $\text{FUS}$, the same combination can be inserted into $H$ several times. To reduce space, $H$ keeps each combination merely once, and maintains a counter for each combination, specifying how many times the combination has been inserted into $H$. After iterating over all the items, the counter of each combination provides the optimality count of that combination.

Under the exists semantics, the answer comprises all the combinations of $H$. The pseudocode appears as Algorithm 4.3. Under the $\text{ALL}$ semantics, only combinations whose optimality count is equal to $t$ are returned. Thus, in Algorithm 4.3, before returning $H$ (Line 16), it discards from $H$ combinations with optimality count smaller than $t$.

**Algorithm 4.3 Top-$h$-Per-Item (TopHI)**

**Input:** Lists $L_1, \ldots, L_t$ of $\text{PreAns}(Q)$. $h$.

**Output:** $\text{EXISTS}_h(Q)$.

**Data structures:** Hash buffer $H$ initialized to be empty.

1: \( E := \emptyset \)
2: for $i = 1$ to $t$ do
3: \quad for each item $o^j_{i_j}$ in $L_j$ do
4: \quad \quad $\mathcal{L} := (L_1, \ldots, L_{j-1}, \{o^j_{i_j}\}, L_{j+1}, \ldots, L_t)$
5: \quad $A := \text{PBRJ}(\mathcal{L}, h)$
6: \quad for all $C \in A$ do
7: \quad \quad if $C \in H$ then
8: \quad \quad \quad $H(C) := H(C) + 1$
9: \quad \quad else
10: \quad \quad \quad add $C$ to $H$
11: \quad \quad \quad $H(C) := 1$
12: \quad \quad end if
13: \quad end for
14: end for
15: end for
16: return $H$

TopHI integrates the join operation with the selection of the combinations. That is, while using pipelining to process the result of the join operation by $\text{FS}$ and $\text{FUS}$, TopHI combines the operations and performs the filtering within the join, to optimize the computation.

Note that in all the algorithms, the optimality count is computed for all the combinations of the answer. Thus, ordering the combinations by their optimality count does not require additional computations other than the sort.
4.4 Experimental Evaluation

We conducted a variety of experiments to evaluate the effectiveness of the various semantics and the scalability of the presented algorithms. In what follows, we first describe the experimental methodology and then present the results of the evaluation.

4.4.1 Methodology

The experiments were conducted on an Intel Core i5 2.53GHz machine with 4GB RAM, running the Windows7 operating system. The algorithms were implemented in Java. In order to be able to vary parameters on demand we generated and utilized synthetic data sets. We also tested our techniques on real data sets. We next describe the data sets and the evaluation metrics we use.

Synthetic Data. Our synthetic data sets were generated as follows. The following parameters are given as input: (1) the number of lists, (2) their lengths, (3) the range of scores of each list and their distribution (uniform or normal), and (4) the association degree, i.e., the probability that two items are joined. Based on these parameters we generate a synthetic set of lists to simulate the answer to a virtual arbitrary combined search query. Specifically, we generate synthetic items for each list, each item with a score. The scores are chosen randomly in the specified range and according to the specified distribution. We create association lookup tables to simulate the join condition. The value of the join condition (association) for every two items is set to be true or false according to the specified probability (the association degree parameter).

Real Data. The real-world data we used was retrieved from Yahoo! Local using the Local Search web services. We sent four queries to Yahoo! Local, searching for hotels, Italian restaurants, steak houses and Irish pubs, all in Boston. The number of results received for the four queries was: 240 hotels, 240 Italian restaurants, 240 steak houses and 189 Irish pubs. The results of each query were received in the order of their relevance score, and several attributes were provided, such as latitude, longitude and average rating. We used a simple join condition which is based on the distance between every two locations. We join two items if the distance between them is at most 400m.

The parameters $m$ (i.e., list length), $t$ (i.e., number of lists), $k$ (i.e., number of combinations in the output), and the association degree $p$ are specified in each experiment. The default values in the experiments over the synthetic data are $m = 1000$ and $t = 4$. The score of a combination, in both the synthetic and real data sets, is the sum of scores of the joined items. Unless noted otherwise, the default value of $h$ (the optimality tolerance) is 1.

\(^{1}\text{http://local.yahoo.com/}\)
Evaluation metrics. We measure the performance of the algorithms in terms of running times. The times measured include the initial phase of producing the join pipeline, for the FS and FUS algorithms. We measure the quality of the results in terms of coverage and optimality ratio. In addition, we use the harmonic mean of the coverage and the optimality ratio (harmonic mean, for short) as an overall quality measure.

4.4.2 Experiments and their results

We conducted three sets of experiments. The first set measures the quality of the result under the various semantics. The second evaluates the efficiency and scalability of the presented algorithms. Finally, the third is a simulation of a user study, which attempts to estimate user satisfaction.

4.4.2.1 Quality measurment

Comparing the (full) result sets. In order to evaluate the semantics presented in Section 4.2, namely EXISTS, ALL and OptimalityRank, we conducted the following experiments. We computed results under EXISTS, ALL, Skyline, RepeatedTop1 and the full Join, over the search results of Yahoo! Local Search, and we compared them in terms of size, coverage, optimality ratio and harmonic mean. The experimental results are presented in Table 4.1. The join size is obviously too large to be useful. It has full coverage but very low optimality ratio, because each item appears in many combinations. Therefore, its harmonic mean is very low. By definition, EXISTS also provides full coverage and a better optimality ratio than the full join, but its size is still relatively high. ALL and Skyline provide high optimality ratio but very low coverage, and their result is very small.

Since in many cases the coverage increases when the size of the result increases, comparing semantics with different result sizes is problematic, because it may give advantage to semantics that yield larger answers. To overcome this, we conducted a test which compares results of the same size, by applying top-\(k\) on the results.

Comparing sets of the same size (by applying top-\(k\)). Given the same result sets mentioned above, we chose an output size \(k\), and computed the same measures over the top-\(k\) combinations of each set. Note that the top-\(k\) results of all the sets except for OptimalityRank are determined simply by the combination scores, while in OptimalityRank (which is equal to EXISTS but sorted differently), combinations are sorted first by optimality count and only then by score. Figure 4.3 presents the coverage, optimality ratio and their harmonic mean, as a function of \(k\). ALL is not presented, because its size is very small and it is contained in OptimalityRank. As expected, as \(k\) increases, the coverage increases while the optimality ratio decreases. The results demonstrate that the new OptimalityRank semantics provides the best coverage for any \(k\), and almost the best optimality ratio. Only Skyline has higher
### Table 4.1: Parameters of result sets over Yahoo! Local data

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Result size (#combinations)</th>
<th>Optimality Ratio (%)</th>
<th>Coverage (%)</th>
<th>Harmonic Mean (Coverage, Optimality Ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Join</td>
<td>83196</td>
<td>0.14</td>
<td>100</td>
<td>0.3</td>
</tr>
<tr>
<td>EXISTS</td>
<td>426</td>
<td>27.3</td>
<td>100</td>
<td>42.9</td>
</tr>
<tr>
<td>ALL</td>
<td>9</td>
<td>100</td>
<td>7.7</td>
<td>14.4</td>
</tr>
<tr>
<td>Skyline</td>
<td>7</td>
<td>64</td>
<td>3.9</td>
<td>7.3</td>
</tr>
<tr>
<td>RepeatedTop1</td>
<td>45</td>
<td>24.4</td>
<td>9.5</td>
<td>13.6</td>
</tr>
</tbody>
</table>

Optimality ratio for $k > 25$, but note that, since the Skyline contains only 7 combinations, it is not influenced by $k$, and its coverage remains very small. Therefore, as demonstrated in Figure 4.3c, the harmonic mean of Skyline is very low and OptimalityRank outperforms all the other semantics.

**Comparing result sets as we vary $h$ (fixed $k$).** In order to evaluate the effect of $h$ on the quality of the results under the new semantics, we computed the result sets of EXISTS$_h$ and ALL$_h$ for various values of $h$ over the Yahoo! Local search results. Figure 4.4a presents the size of ALL$_h$ as a function of $h$. As expected, the size of ALL$_h$ increases linearly with $h$. The same holds for EXISTS$_h$, which is omitted because its size is much larger.

To compare the above result sets in terms of coverage and optimality ratio, again we applied top-$k$ on each result set. In this experiment we used $k = 50$. Figures 4.4b, 4.4c and 4.4d present the coverage, optimality ratio and harmonic mean, respectively, of the top-50 combinations in the various result sets, as a function of $h$. OptimalityRank$_h$ provides the highest coverage for every $h$, in comparison to the other sets. When $h$ increases, the coverage decreases because each item has more optimal combinations, and less new items are included in top-50. Note that the lines of ALL$_h$ and OptimalityRank$_h$ unite from some value $h$, in all three measures. This happens for values of $h$ such that $|\text{ALL}_h| \geq 50$, because then the top-50 of both ALL$_h$ and OptimalityRank$_h$ are equal. For lower values of $h$, ALL$_h$ has higher optimality ratio, because of its small size, while OptimalityRank$_h$ fills all the positions in the top-50. Nevertheless, the harmonic mean of OptimalityRank$_h$ is highest for every $h$. To conclude, the effect of increasing $h$ is a high increase in optimality ratio and a low decrease in coverage. The overall effect on the harmonic mean is that it increases until some point, and then, when the coverage gets low enough, it starts decreasing. Therefore, the optimal $h$ for the harmonic mean of OptimalityRank$_h$ is higher than 1.

**Conclusions.** OptimalityRank$_h$ enables adjusting the output size as desired. The top-$k$ combinations in it are selected greedily with respect to optimality, which is its main advantage over the other semantics. It provides the highest harmonic mean of coverage and optimality ratio compared to the other semantics, for any value of $k$ and $h$. To get the optimal measures, one should use $h > 1$, such that the size of ALL$_h$ is almost but no more than $k$. 

---

2Skyline and RepeatedTop1 are not influenced by $h$ and are only presented for comparison.
4.4.2.2 Computation running times

To test the efficiency of answering search queries under the proposed semantics, we compare the performances of the three algorithms under the \textit{EXISTS} and \textit{ALL} semantics. Unless specified otherwise, the default setting we used is of four lists, the scores in each list are distributed uniformly in the range [0,100], and we used $h = 1$.

\textbf{The effect of association degree.} Figure 4.5 shows the performances of the three algorithms for computing \textit{ALL} as a function of the association degree between items.\footnote{The association degree is the probability that two items are joined.} The experiment was conducted on lists of length 100. As the association degree increases, the running time of both \textit{FS} and \textit{FUS} increases exponentially, while the running time of \textit{TopHI} does not change. This is due to the exponential increase in the join size, which affects both \textit{FS} and \textit{FUS}. Note that \textit{FS} crashed for $p \geq 80\%$ on \textit{Out Of Memory}. 

---

Figure 4.3: The coverage, optimality ratio and the harmonic mean of top-k results under various semantics over Yahoo! Local data, as a function of $k$

(a) Coverage

(b) Optimality ratio

(c) Harmonic mean
Figure 4.4: Measures of the top-50 results under various semantics over Yahoo! Local data, as a function of $h$

**error.** Although FS filters the combinations on the fly, the procedure that produces them sorted has extremely high memory requirements. FUS, on the other hand, has almost no space overhead, but is still very slow.

In order to examine the scalability of TopHI, we conducted the same experiment on lists of length 1000. The results are presented in Figure 4.6, in two separate graphs, due to the different scales. Figure 4.6b demonstrates again the exponential effect of $p$ on FUS\(^4\). TopHI, on the other hand, is highly efficient, and terminates in less than a second (see Figure 4.6a). Its performance is even better for higher association degrees, as the top-1 combination per item can be found in ‘higher’ positions of the lists.

The effect of lists length. Figures 4.7a and 4.7b present the running times of the algorithms, when computing EXISTS, as a function of the lists length ($m$). The experiment was conducted over four lists with an association degree of 10%. It can be seen that the running times of both FS and FUS increase exponentially in the length of the lists, while the running time of TopHI is linear in the length of the lists.

\(^4\)The running times for higher values of $p$ are very long and therefore omitted.
lists. As noted earlier, although the join size increases exponentially, the difficulty of finding the top-1 combination remains about the same, for a fixed association degree, and therefore the effect of the length on TopHI is linear in the size of the input.

**Effect of $h$.** Figure 4.8 illustrates the performance of TopHI for computing $ALL_h$ as a function of $h$. The experiment was conducted over four lists of length 1000 with an association degree of 30%. Only TopHI was evaluated, and as expected, it shows a linear increase in its running time as $h$ increases.

### 4.4.2.3 Simulating user satisfaction

Designing a user study, in which users join search results and evaluate answers is problematic, due to the exponential size of the join result. Even when considering short lists of, say, 10 items, there may
be hundreds or thousands of valid combinations, which makes it almost impossible to be visualized or evaluated by a human user. Therefore, we simulated a user study that estimates user satisfaction from various result sets. Our simulation is based on the following assumptions:

1. An item satisfies the user with probability that is proportional to its score, assuming normalized scores in $[0, 1]$.

2. A combination satisfies the user iff the user is satisfied with all of the items it comprises.

3. A set of combinations satisfies the user iff the user is satisfied with at least one combination in the set.

**The data "evaluated" by the users.** We used the Yahoo! Local search results, where only the first 20 items from each list were considered. Note that the join size for these $4 \times 20$ items is 3757. We generated decreasing scores in the range $[0, 1]$ for the items in each list, such that the scores decay
exponentially. We varied the exponent from 1 to 4 (in four independent experiments), to evaluate the effect of the decay on user satisfaction. Given the lists of scored items, we computed the top-20 results under OptimalityRank, RepeatedTop1, Skyline, and the full Join.

**Simulating user satisfaction.** We simulated 10,000 users as follows. For each user we generated his satisfaction with every item with probability equal to the item score. Then, for each of the four result sets, i.e., the top-20 of OptimalityRank, RepeatedTop1, Skyline, and the full Join, we computed two measures of satisfaction:

- **IsSatisfied(user):** Is the user satisfied with the result set. The value is 0 or 1.
- **#SatisfyingHotels(user):** How many distinct hotels appear in satisfying combinations in the result set.

The first measure — the user satisfaction measure — relates to whether the result set contains at least one highly ranked combination. The second — the diversity measure — relates to the diversity of hotels in “good” combinations. Note that the diversity measure is subjective per user, as it considers only satisfying combinations. Having done that for all users, we computed, for each of the two measures, the average value.

**Results.** Figure 4.9 demonstrates the average measures over 10,000 users, for every result set, as we increase the exponential decay of the scores. OptimalityRank is more effective than the other semantics in satisfying users and in providing diversity. Note that both measures decrease as the exponent increases, because items have lower probability of satisfying the user. Skyline has the lowest satisfaction and diversity measures, as it contains only few combinations. RepeatedTop1 provides a larger result set, but it is also ineffective, as it discards many highly ranked (satisfying) combinations in favor of objective diversity. The top-\(k\) over the join is closest to OptimalityRank, but still less effective. Its measures are high because it favors high scores over diversity, which also increases the subjective diversity, as it considers only satisfying combinations. Note that the difference between the top-\(k\) join and OptimalityRank is highest on the satisfaction measure when the exponent is high, while it is highest on the diversity measure when the exponent is low. The explanation to this is that when the exponent is low, more items satisfy the user, and OptimalityRank provides much more diversity than top-\(k\). On the other hand, when the exponent is high, only the first items in each list may be satisfying. This enables very little diversity for any semantics, while it gives an advantage to methods that favor highly scored combinations. Note that even in this case, OptimalityRank is higher than top-\(k\) join, as it provides a diversity of items, which increases the probability of having a satisfying combination. Note that even a highly scored item may fail to satisfy the user, and therefore diversity in essential.

**Score-independent satisfaction.** User satisfaction is not fully correlated with the ranking of search results, especially when considering the 10-20 most highly ranked items. User satisfaction with good
enough items may depend on subjective preferences. For example, amongst the top-20 hotels provided by Yahoo! Local search, it is reasonable to assume that only few of them will satisfy the user and not necessarily those with the highest rank. In order to simulate this, we conducted another experiment, very similar to the previous one, but with the following modification: only for the list of hotels, we selected for each user 3 random hotels that satisfy him. All other hotels are deemed non satisfying. For the items of the other lists, the satisfaction status is generated as before, proportional to the score. As before, we simulated 10,000 users, and computed the average measures. The results are shown in Figure 4.10. Note that in this experiment, semantics that provide diversity have higher probability to return one of the satisfying hotels in its result, and therefore are expected to have higher satisfaction measure. As expected, \textit{OptimalityRank} is again more effective than the other semantics, while top-$k$ join performs worse than in the previous experiment.

![Figure 4.9: User satisfaction measures (item satisfaction assigned by score)](image)

![Figure 4.10: User satisfaction measures (random satisfying hotels)](image)
4.5 Related Work

Early studies on ranking and on top-k queries considered selection over a single table of items with several scored attributes. The top-k selection algorithms of Fagin et al. [30] have been the most influential. They read the input in sorted order and use bounds on unseen scores to terminate. The more general problem of top-k join has been studied extensively [31, 43, 46, 59, 65]. These various papers all follow the strategy of Fagin et al. [30] of reading the input lists in decreasing order of base scores. Our framework of joining search results is closely related to the rank join problem. However, the latter aims at finding the top-k combinations while reading as few input items as possible, and ignores the problem of item repetitions. As demonstrated in previous sections, the top-k join results have low coverage and low optimality ratio.

The problem of choosing a subset of the join result may have several alternative solutions [69]. One of them is by using the skyline operator [11]. This operation filters out a set of combinations which are considered most interesting, as they are not dominated by any other combination (see definition in Section 4.2). Many algorithms have been proposed for efficient computation of the skyline in different settings, such as multi-relations and distributed systems (e.g., [50, 74]). The skyline semantics is completely different from ours, and its result set usually has poor coverage and may also have low optimality ratio.

Many variations of the skyline operator have been proposed, most of which were designed to return a subset of the skyline, in case the data has high dimensionality, and the skyline result is too large to be useful [17,18,56]. In the problem of joining search results the dimensionality is relatively small, and the full skyline usually has poor coverage, thus, these methods will decrease the already very low coverage of the skyline and they are expected to provide very poor results under our measures.

Yiu and Mamoulis [77] propose the top-k dominating query, which returns the k objects that dominate the highest number of data objects. It combines the advantages of top-k and skyline queries, in that the user can control the output size, and the result is based on preference. Specifically, this query returns the top-k objects, where the score of an object p is equal to the number of objects dominated by p. Although the output size may be tuned, many important combinations may be discarded because they dominate only few or no other combinations, and instead, the combinations returned may have many overlapping items. Therefore, the result in our case is expected to have low coverage and low optimality ratio.

Other studies relax the dominance definition, to allow a distance of $\epsilon$ between objects. Jin et al. [49] propose the "thick skyline", which recommends not only skyline objects but also their nearby neighbors within $\epsilon$-distance. Xia et al. [76] propose a more general definition of dominance, which considers weights on the dimensions and a distance $\epsilon \in [-1, 1]$, such that the result increases gradually from an empty set, through the full skyline, and to the whole input, as $\epsilon$ decreases from -1, through 0, and to 1,
respectively. Applying this method to our case is problematic, since we do not know what the differences in scores are and whether the scores and the differences among them are uniformly distributed. It is common to assume that in search results there is an exponential decay of the scores, and in such case this approach cannot be applied.

Another possible solution to the problem of how to choose a small set of combinations is to use diversifying methods over the entire join result. Search result diversification can be characterized as a bi-criteria optimization problem, in which one seeks to maximize the relevance of the results to the query aspects, while minimizing redundancy with respect to these aspects. The early work of Carbonell and Goldstein [15] described the tradeoff between relevance and novelty (a measure of diversity). They use a similarity function to measure the similarity between documents, and a parameter controls the degree of the tradeoff between the relevance function and this similarity function. Gollapudi and Sharma [36] conducted an axiomatic study of diversification systems, which enable characterizing objectives independent of the algorithms used and the specific relevance and similarity functions used. These general methods may be applied in our case, by choosing appropriate similarity functions, e.g., the number of overlapping items between two combinations. While this method may provide good coverage and low redundancy, it is not suitable for maximizing optimality ratio. A demonstration for that is given in the experimental evaluation of the RepeatedTop-1 semantics, which is essentially a diversification method which requires zero similarity between combinations. Subsequent studies propose very different diversification methods, using specific similarity functions between documents, conditional relevance distributions defined over documents [20], taxonomy of information [3], and more [24, 79, 80].

A different definition for diversity was proposed by Vee et al. [75] for semi-structured data, for the special case where there exists a priority ordering of the attributes of the queried items. The authors propose building a dewey-like index based on the hierarchy of attributes, and present efficient algorithms for searching in it while providing diversity. Yet, the requirement to prioritize the attributes (the subqueries in our case) may be inappropriate in many cases, as we consider the different lists (i.e., subqueries) equally important. In addition, as in classic skyline queries, every tuple of attributes is distinct from other tuples, and repetitions are meaningless.
Chapter 5

Conclusions

In this thesis, we addressed three problems related to combined search. The first problem is how to evaluate twig queries efficiently over indexed XML documents. The second is how to efficiently compute the top-k maximal join of basic search results. Finally, the third problem is which subset of the join result should be returned to the user as the most effective answer to a complex query.

**Data extraction from XML repositories.** We initiated a systematic study of memory lower bounds for evaluating twig queries over indexed documents. We provide an analytical explanation for the difficulty in handling queries with child-axis nodes, and also point out the overhead incurred by algorithms that work in the pattern matching mode. We present a new algorithm that avoids this overhead, and achieves dramatic improvements in space for certain types of queries. We introduce a new model of communication complexity, the TMC model, through which we can prove space lower bounds for multiple data streams algorithms.

**Maximal join of ranked lists.** We have studied the problem of how to compute the top-\(k\) maximal rather than complete combinations, in a join of ranked lists. Such computation is required in integration of search results over heterogeneous sources, where relevant items in one of the lists may not have associated items in the other lists. Allowing incomplete answers in the result reduces the risk of a situation where relevant information is discarded by the join and hence, absent from the result.

We show that when computing maximal answers and the result contains null values, any algorithm must entirely read the lists for which there is a null in some top-\(k\) maximal combination. When lists must be read completely, optimization techniques that reduce the number of items being read are limited. They can increase the efficiency of the algorithms but they only do that for some of the lists and in the worst case (nulls in all places), the entire input must be read by any algorithm. In such cases, instance optimality does not distinguish between different algorithms. Therefore, with respect to instance optimality, it is more appropriate to compare algorithms that provide a \(\theta\)-approximate answer.
We study and compare four algorithms for computing maximal answers in a join of ranked lists. Two algorithms (HNL and VNL) are modifications of known algorithms (for complete answers). The other two algorithms (TMax and VHNL) are novel. We investigated the efficiency of the algorithms—analytically and experimentally. Algorithm TMax is designed to read as few items as possible from the lists and to allow $\theta$-approximate answers by stopping the computation before reaching the end of the lists. Algorithm VNL is efficient for lists in which the percentage of associations is low. Algorithm VHNL combines VNL with HNL and it is, in practice, more efficient than all the other algorithms.

**Ranking joined search results.** The need to limit the size of the result, when joining ranked lists of item, has long been recognized, and several methods were suggested for this problem, among them, the top-k operator, the skyline operators and diversity methods.

However, these methods are not designed to be effective when filtering the join of search results. That is, these methods do not choose the combinations of the result while aiming at maximizing the chance of including in the result at least one combination that satisfies the user. In addition, their result is not genuinely diverse, in the sense that the result does not include sufficient items in their optimal combination, for presenting to the user actual options to choose from. In this study we introduce two measures, namely coverage and optimality ratio, and we show that maximizing the harmonic mean of these measures increases the effectiveness and the actual diversity of the result.

We use the proposed measures for comparing the top-k, skyline and diversity methods and for estimating their quality. Evidently, skyline usually provides a relatively high optimality ratio but fails in providing a good coverage. The diversity method Repeated-Top-1 provides neither good coverage nor good optimality ratio, and Top-k is frequently weak on both measures. We introduced novel semantics—EXISTS that provides maximal coverage, ALL that provides maximal optimality ratio, and we showed how to rank the results of EXISTS to achieve a good answer in terms of both measures (Optimality Rank). We also showed how these semantics can be refined to deal with imprecise ranking, by replacing optimality by $h$-optimality. Algorithms for query evaluation under our semantics were presented and tested over both synthetic and real-world data. Our experimental evaluation shows that the answer to a combined search query can be computed efficiently. However, this cannot be done by merely filtering the answer to a state-of-the-art top-k join algorithm or filtering an ordinary join algorithm. Thus, an algorithm that performs the filtering within the join, namely TopHI, is presented and shown to be much more efficient than the algorithms that separate the join operation from the filtering operation.
References


114


In this work, we define an operation that fills the gaps of items with nulls. We define a combined item as maximally complete if it cannot be expanded by replacing a null item with a real item. We propose two new algorithms to compute the maximally combined best combined k, and in addition, we study two familiar combination algorithms that we adapted to the computation of maximally combined combinations. The two algorithms are much more efficient than the other three algorithms, as we can see both in our analysis and in empirical experiments.

In the process of combination, the same item may appear in many combinations, and therefore, the result of the combination usually contains a very large number of combinations. The solution is to choose a subset of combinations and return only that to the user. In this part of the work, we deal with how to choose the subset to be returned.

We propose two measures for evaluating the efficiency of a subset:

- Coverage: The coverage is the number of items from the lists that appear in the optimal combination.
- Optimal ratio: The optimal ratio is the maximum score of the combination that contains this item.

The challenge of selecting the subset of combinations lies in the fact that there is a conflict between the two measures, and increasing one decreases the other.

We propose new semantics for choosing the subset to be returned to the user, which aim to provide a high coverage and a high optimal ratio, and present algorithms for the efficient computation of complex queries under our semantics.

We study the efficiency of the combination result under existing semantics and under our new semantics using the two measures. Finally, we present the results of experiments conducted on information received from Yahoo! Local, which confirm the efficiency of our semantics and the ability to compute complex queries efficiently.
XML

XPath queries are usually represented by a small tree structure (twig) with tags on the tree nodes.

Calculating such a query over XML documents, which are also modeled as trees, requires finding matches of the query tree in the document trees.

Similar to other information repositories, XML repositories can also build an index of information to speed up searches.

In XML documents, each tag has a separate list, where each occurrence of the tag is stored along with its position in the tree and depth.

Over the years, many algorithms have been proposed for calculating XPath queries over XML document indexes.

Effort was invested in supporting XPath queries more extensively, and optimizing the calculation in terms of running time, memory, and input/output.

Nevertheless, it can be identified as two main trends among existing algorithms.

Firstly, most of them are very efficient when the query contains only direct parent–child relationships, whereas very inefficient when an indirect parent–child relationship exists.

The second trend is related to the type of calculation: many algorithms return all the matches of every query tree in the document, even sometimes just the output of the query or even just the flag indicating whether at least one match exists.

Two questions arise in this situation.

Firstly, is the high memory cost of queries with indirect parent–child relationships necessary, or can it be reduced?

Secondly, does the type of calculation have an effect on memory cost, i.e., is finding matches of the entire tree indeed costlier in memory than finding matches of the output node or filtering.

In order to answer these questions, we studied the memory complexity of calculating XPath queries over document indexes in depth.

We found that this complexity depends on three factors: (1) whether the query contains only a single path or is a tree, (2) the types of relationships between the nodes in the query, and (3) the type of calculation.

We prove lower bounds through a new communication complexity model and upper bounds that are valid in most cases.

The combination of complex queries that integrate information from heterogeneous sources are calculated by combining the results of several basic queries separately.

In this step, each basic query returns a list of items sorted according to the query relevance, and the next step is to combine (not mix) these lists and return the k best combinations.

The problem of combination and finding the best combinations has been studied extensively, but only complete combinations are taken into account.

However, in the combination operation, since a combination must include an item from each list, it is possible that many items with a high score cannot be combined into any complete combination, so they do not appear in the output, even though they may be in partial combinations that are better than other complete combinations.

Therefore, the algorithm returns a combination of k elements with the highest possible memory cost.
Applications such as search engines allow searching within their databases, whether it's a search across the whole internet or within a separate information system. The resolution to the search query is a ranked list of information items relevant to the query.

Today, most search applications allow submitting a single query to a specific database, and each query is independent of the others. However, at times there is a need to search for complex information that requires integrating the results of multiple queries.

Here's an example of a complex search: an entrepreneur plans to travel to a certain city, and is interested in finding a hotel that meets certain criteria, and nearby restaurants where he can have business lunch and dinner.

Additionally, his budget is limited to a certain amount, so the hotel cost for the night, including a business lunch and an evening meal, must fall within his budget.

The entrepreneur can submit three separate search queries to the appropriate websites (for example, www.tripadvisor.com):

- One query for searching for a hotel.
- A second query for searching for a restaurant that offers business lunch.
- A third query for searching for a restaurant for dinner based on taste.

However, an integration of the results from the different lists according to distance and budget is not simple for a human, it's much easier when taking into account the rating of each result and giving preference to the best integrated results.

Working on complex queries, such as in the example, involves several steps:

Firstly, for each basic query, one must extract relevant information from the appropriate database and rate the information items.

Secondly, one must integrate the ranked results from the different lists, that is, perform a "join" operation according to the conditions specified in the general query.

The result of the integration is a set of combinations of information items, where each combination includes one information item from each list.

Finally, one must rate the combinations and select which to present to the user.

This work deals with several aspects of computing complex queries. In the first part, we deal with the output stage of the information and analyze how to efficiently extract information from XML documents. This part does not deal with rating the results.

From here onwards, we assume that we already have ranked lists of results for the basic queries and focus on the integration between them.

In the continuation, we deal with two problems of the integration process and the output choice. The first problem is that in the process of integration, many items may not appear in any combination, despite being relevant and rated highly for the user.

To solve this problem, in the second part of the work, we define the operation of "maximal combination", which allows incomplete combinations, but only maximal ones, that is, combinations that cannot be extended anymore.

The last part of the work deals with the question of which subset of combinations to return to the user so as to maximize the probability of providing him.

The work was conducted by Technion Computer Science Department Ph.D. Thesis PHD-2013-05 - 2013.
The research was conducted with the supervision of Professor Irkon Kanha from the Faculty of Computer Science at Technion, and Professor Meir Ziv Bar-Yosef from the Faculty of Electrical Engineering at Technion and the Engineering Development Center of Google in Haifa.

I wish to thank and express my gratitude to many people, and I am happy and moved to acknowledge to them their gratitude.

First and foremost, the gratitude and great admiration reserved for both my supervisors, Irkon Kanha and Meir Ziv Bar-Yosef. This was a great privilege to work with such beautiful, inspiring and fundamental women. I benefited from your wisdom and guidance in a great extent during all these years, and there are no words to express my gratitude to you, on your patience, the help that you provided for me for a long time.

I wish to add to my gratitude, from the deepest of my heart, for your support and patience throughout the years, and for your patience and patience in the months when we lived together (sometimes actively) and the struggle we had to settle the mental issues of us.

My deepest gratitude to my family, the traditional family, the warmest and the most stimulating one.

I wish to say thank you to my parents, Shoshana and Aron, and to my dear Tal, for your irreplaceable support, your encouragement, and your love, without limit.

I wish to say thank you to my three eldest children, who heard many lectures when they were pregnant, and who had to spend many Saturday evenings alone, thank you for the love and the love.

I wish to thank my dear and wonderful sister, Ronit, for your support and encouragement, the many talks and the wise advice.

Finally, I wish to thank all the dear family, my instructors, who believed in me, for your faith, for the love, and for the love.

I wish to thank Technion for the financial support in my PhD studies.

Technion - Computer Science Department - Ph.D. Thesis PHD-2013-05 - 2013
שאילות חיפוש מורכבות במאגרי מידעי הטרוגנים

הנובר על מחקר

לש מליי חלקי של הדרישות לקבשל התואר

דוקטור לפילוסופיה

מירית שלם

miriya@cs.technion.ac.il

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

אדר תשע"י

פברואר 2013

חיפה
שאילחות חיפוש מורכבות במאגרי מیدע הטרווגיים

מירה שלם