UNIVERSITY OF CALIFORNIA, SAN DIEGO

Context-sensitive Ranking

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Liang Chen

Committee in charge:

Professor Yannis Papakonstantinou, Chair
Professor Alin Deutsch
Professor James D. Hollan
Professor Chen Li
Professor Victor Vianu

2012

The dissertation of Liang Chen is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2012

DEDICATION

To my parents, Guoguang and Zhengbi.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Yannis Papakonstantinou, for his guidance. Yannis is an exceptional professor who is always looking for challenging and interesting problems. He has been constantly encouraging me to conduct innovative research, rather than to follow others' steps. His research style and taste heavily influence me and remind me what kinds of works I should do in the future. I would also like to thank his insightful comments all through my graduate studies. He always has a clear high level picture, and guides me to formal and elegant solutions.

I would like to thank Phil Bernstein. Phil was my mentor for two summer internships. The work done with him is one of the most elegant works I've ever done. The experience significantly changes my view of research. Phil is extremely humble, even though he is a renowned researcher and has a busy schedule. He reviewed every document I produced and carefully provided detailed comments. It was such a delightful experience working with him.

I would like to thank my other collaborators. I had a lot of discussions with Nathan Bales and Alin Deutsch on a large body of this thesis through last two years. These discussions were very inspiring. I worked with Yu Xu as a summer intern when I was still young in database research. His careful guidance helped me mature in research. I would also like to thank my committee members Victor Vianu, Chen Li, and James Hollan. Their insightful comments help this thesis a lot.

I would like to thank every member of the database lab. It's been a great pleasure to attend inspiring seminars, enjoy delicious potlucks and talk technology and academia gossips.

Finally, I would like to thank my parents in Beijing. They paid close attention on every step in my graduate study and always encouraged me when things didn't go smoothly. I couldn't go so far without their support.

Parts of Chapter 2 were published in ACM SIGMOD International Conference on Management of Data 2011, entitled "Context-sensitive Ranking for Document Retrieval". This is joint work with Yannis Papakonstantinou. The

other parts are currently being prepared for submission for publication. This is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the two papers.

Chapter 3 was published in IEEE International Conference on Data Engineering (ICDE) 2010, entitled "Supporting Top-K Keyword Search in XML Databases", and in ACM SIGMOD International Conference on Management of Data 2011, entitled "Context-sensitive Ranking for Document Retrieval". Both papers were joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the two papers.

Chapter 4 and 5 are currently being prepared for submission for publication, which is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the paper.

Chapter 6 was published in ACM SIGMOD International Conference on Management of Data, 2011, entitled "Context-sensitive Ranking for Document Retrieval". This is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the paper.

Chapter 7 and 8 are currently being prepared for submission for publication, which is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the paper.

Parts of Chapter 9 were published in ACM SIGMOD International Conference on Management of Data, 2011, entitled "Context-sensitive Ranking for Document Retrieval". This is joint work with Yannis Papakonstantinou. The other parts are currently being prepared for submission for publication. This is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the two papers.

VITA

2004	B. S. in Electrical Engineering, Tsinghua University, Beijing, China
2004-2006	M. S. in Electronic Engineering, Tsinghua University, Beijing, China
2012	Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS

Liang Jeff Chen, Philip A. Bernstein, etc, "Mapping XML to a Wide Sparse Table" (extended version), *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2012.

Liang Jeff Chen, Philip A. Bernstein, etc, "Mapping XML to a Wide Sparse Table", *International Conference on Data Engineering (ICDE)*, 2012.

Liang Jeff Chen, Yannis Papakonstantinou, "Context-sensitive Ranking for Document Retrieval", *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 757—768, 2012.

Liang Jeff Chen, Yannis Papakonstantinou, "Supporting Top-K Keyword Search in XML Databases", *International Conference on Data Engineering (ICDE)*, pages 689—700, 2010.

Yu Xu, Xin Zhou, Pekka Kostamaa, Liang Chen, Handling Data Skew in Parallel Joins in Shared-Nothing Systems, *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1043—1052, 2008

Liang Chen, Shaozhi Ye, Xing Li, Template Detection for Large Scale Search Engines, *ACM Symposium on Applied Computing (SAC)*, pages 1094—1098, 2006

ABSTRACT OF THE DISSERTATION

Context-sensitive Ranking

by

Liang Chen

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Yannis Papakonstantinou, Chair

We are witnessing a growing number of applications that involve both structured data and unstructured data. A simple example is academic citations: while the citation's content is unstructured text, the citation is associated with structured data such as author list, categories and publication time. To query such hybrid data, a natural approach is to combine structured queries with keyword search. Two fundamental problems arise for this unique marriage: (1) How to evaluate hybrid queries efficiently? (2) How to model relevance ranking? The second problem is especially difficult, because all the foundations of relevance ranking in information retrieval are built on unstructured text and no structures are considered.

We present context-sensitive ranking, a ranking framework that integrates structured queries and relevance ranking. The key insight is that structured queries provide expressive search contexts. The ranking model collects keyword statistics in the contexts and feeds them into conventional ranking formulas to compute ranking scores. The query evaluation challenge is the computation of keyword statistics at runtime, which involves expensive online aggregations.

At the core of our solution to overcome the efficiency issue is an innovative reduction from computing keyword statistics to answering aggregation queries. Many statistics, such as document frequency, require aggregations over the data space returned by the structured query. This is analogous to analytical queries in OLAP applications, which involve a large number of aggregations. We leverage and extend the materialized view research in OLAP to deliver algorithms and data structures that evaluate context-sensitive ranking efficiently.

Chapter 1

Introduction

We are witnessing a growing number of applications that involve both structured data and unstructured data. A simple example is academic citations: while the citation's content is unstructured text, the citation is associated with structured data such as author list, categories and publication time. To query such hybrid data, a natural approach is to combine structured queries and keyword search, as these two querying paradigms have been used in database and information retrieval (IR) for decades.

Consider academic citations shown in Table 1.1[1]. Each citation is associated with structured data such as publication year, author list and category, as well as unstructured text. Structured queries are extended with keyword search to query both structured data and unstructured text. For example, the SQL query Q_1 searches citations that belong to the category of digestive system and contain two keywords "`leukemia, pancreas`". The predicate Category = `'digestive system'` is a conventional structured predicate over the structured data, and the query construct `CONTAINS` performs keyword search over text.

[1]The Citation table is not in a norm form. The types of the Author and Category columns are lists rather than atomic values.

$\mathbf{Q_1}$:

```
SELECT   *
FROM     Citation
WHERE    Category = 'digestive system' AND
         Content CONTAINS ''leukemia, pancreas''
```

Table 1.1: Academic citations of biomedical science

CID	Authors	Year	Category	Content
86	Smith, Chen	2006	digestive system, transplant	We present in the paper studies on complications following pancreas transplant ...
200	Mason,Allen	2004	leukemia, digestive system	organ failure in patients with acute leukemia ...
...	...	...	...	.....

Two important problems arise for such hybrid query languages: first, how to evaluate the query efficiently? Query processing of structured queries and keyword search have different techniques, e.g., B-tree indexes for structured data and inverted indexes for text. How to integrate these techniques seamlessly is a critical problem in order to achieve high query efficiency. Second, how to evaluate result ranking? Ranking is an attractive feature of IR systems. While a user of conventional databases needs to examine all returned results, a user of an IR system usually can get what he/she is looking for in top ranked results, due to sophisticated ranking formulas that evaluate the relevance between the query and results. On the integration of structured data and unstructured text, it is a natural desire to introduce relevance ranking in query semantics, and to improve search effectiveness and user experience.

In this dissertation, we study result ranking for the integration of structured queries and keyword search. At the first glance, a straightforward ranking scheme is to decouple the structured data and unstructured text and to process

structured queries and keyword search separately. Specifically, keyword search is evaluated on unstructured text, yielding a ranked list of results; structured queries over the structured data are further processed to eliminate unsatisfied results in the ranked list. In the above example query, the keyword query ``pancreas, leukemia'' is first evaluated over the Content column, returning a list of tuples ranked by IR relevance scores; the structured query Category = 'digestive system' is then evaluated over the Category column, removing tuples that do not satisfy the predicate.

Such a straightforward ranking scheme, however, always violates IR ranking heuristics. IR heuristics are intuitions used to design IR ranking functions. They reflect people's common understandings on how textual content is ranked. When a heuristic is violated, the ranking function may not perform well empirically since it is not entirely consistent with our intuitive preferences. To design ranking schemes for the integration of structured queries and keyword search, we expect that these heuristics are satisfied too. In the following, we first describe an intuitive example that illustrates the value of structured queries in regulating IR heuristics and ranking. Then we formally discuss IR heuristics, and analyze why the naive decoupling of structured queries and keyword search will result in violations.

1.1 Motivating Use Case

PubMed [4] is a database that contains 18 million biomedical citations. All the citations include title, abstract, and authors' information. Citations are often linked to full-text articles. Additionally, every citation is annotated with one or more MeSH (Medical Subject Headings) terms from a controlled vocabulary, which specifies a variety of concepts in biomedical science, e.g., "anatomy", "diseases" and "diagnosis". MeSH terms in the vocabulary are organized in a hierarchy, as shown in Figure 1.1. A MeSH term may appear in several places in the hierarchy tree.

The vocabulary and the hierarchy of MeSH terms represent an ontol-

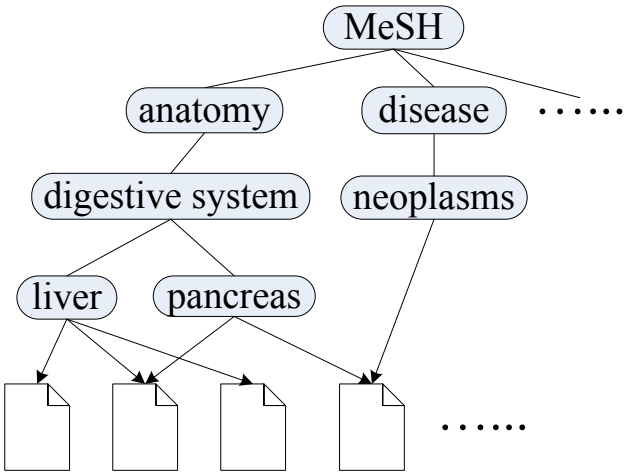


Figure 1.1: MeSH terms and the hierarchy

ogy of biomedical science and structurally organize citations. Each MeSH term represents a biomedical concept and indexes a list of related citations. A combination of MeSH terms forms a structured query that specifies a context spanning the corresponding concepts. For example, "neoplasms" and "digestive system" represent two concepts under "diseases" and "anatomy" respectively. The combination of the two terms identify a set of citations, which form a search context for researchers and doctors concentrating on gastrointestinal (GI) cancer.

Such MeSH term combinations have a special value for information retrieval in PubMed. Keyword distributions and statistics often vary dramatically from a specialized context to another. For example, research on cancer and research on digestive system have very different terminology. Using keyword statistics from a specialized context in ranking functions will deliver a specialized ranking order for the documents in that context. For example, the classical TF-IDF model uses *document frequency* (DF) as term weights to boost the ranking of documents that contain query terms that are rare in the collection. The rationale is that rare query terms are more discriminative, and therefore are more important in identifying relevant documents than frequent query terms. In the above example, a query term that is frequent for the citations on neoplasms may be rare for the citations on digestive system. The ranking order of two documents may be reversed when users are interested in different contexts.

Consider the example query Q_1 that searches two keywords ``pancreas, leukemia'', and two citations d_{86} and d_{200} in Table 1.1, both annotated with the MeSH term "digestive system". Assume we rank the two citations by $tf \times idf$. Since both citations match precisely one query term (matched query terms are underlined in Table 1.1), the ranking order of the two citations is only determined by idf. Without the predicate Category = 'digestive system', the frequency of leukemia is higher than pancreas in PubMed. Hence, d_{86} is ranked higher than d_{200}. However, if the query is issued by a GI doctor or researcher, whose focus is on digestive system, the frequency of leukemia is much less than pancreas in the corresponding context, and therefore d_{200} should be ranked higher than d_{86}. Intuitively, pancreas transplant is a common topic among GI researchers. Leukemia in the query is more discriminative in the context. Given that d_{200} is annotated with the MeSH term "digestive system", it is very likely that the organs mentioned in the d_{200} refer to digestive organs, which include pancreas.

1.2 IR Heuristics

The deduction behind the above PubMed example relies on ranking heuristics, specifically TF-IDF intuition. A variety of IR ranking models for text search have been developed over decades. Though these models evolved separately in the history, they all follow a small set of IR heuristics. In particular, Fang et al. [42] formally define a set of basic constraints that any reasonable ranking functions should satisfy. These constraints capture commonly used retrieval heuristics, such as TF-IDF scheme, in a formal way, so as to apply them to conventional IR ranking formulas analytically.

Let d or d_i be a document modeled as a bag of words, Q_k be a keyword query, and w or w_i be a keyword/term. Let $tf(w, d)$ be the number of occurrences of w in document d, and $|d|$ be the length of d (in terms of number of terms). Let $df(w)$ be the number of documents containing w, and $idf(w)$ be the inverse of $df(w)$. The ranking function $f(d, Q_k)$ assigns a ranking score of d with respect to

Q_k to evaluate the relevance between d and Q_k.

The *Term Discrimination Constraint* (TDC), which formalizes the intuition of TF-IDF weighting, is described as follows:

For a two-keyword query $Q_k = \{w_1, w_2\}$ *and two documents* d_1, d_2, *assume* $|d_1| = |d_2|$ *and* $tf(w_1, d_1) + tf(w_2, d_1) = tf(w_1, d_2) + tf(w_2, d_2)$. *If* $idf(w_1) \geqslant idf(w_2)$ *and* $tf(w_1, d_1) \geqslant tf(w_1, d_2)$, *then* $f(d_1, Q_k) \geqslant f(d_2, Q_k)$.

The TDC constraint formalizes the effect of term discrimination and regulates the interaction between TF and IDF. The above constraint mathematically ensures that, given a fixed number of occurrences of query terms, the ranking function favors a document that has more occurrences of discriminative terms (i.e., high IDF terms).

The TDC constraint, however, will be violated if we decouple structured data and unstructured text, and only apply conventional IR ranking functions on text. Consider the example query Q_1. The relevance ranking is determined the Content column. Since both citations match precisely one query term, i.e.,

$$
\begin{aligned}
&w_1 = \texttt{leukemia}, w_2 = \texttt{pancreas} \\
&tf(w_1, d_{86}) = 0, tf(w_2, d_{200}) = 0 \\
&tf(w_1, d_{86}) + tf(w_2, d_{86}) = tf(w_1, d_{200}) + tf(w_2, d_{200}) = 1
\end{aligned}
$$

by the TDC constraint, the order of the two citations is only determined by $idf(w_1)$ and $idf(w_2)$. If we only consider the Content column of the entire table, `leukemia` has a higher DF than `pancreas` in PubMed. Hence, `pancreas` has a higher IDF and d_{200} should be ranked higher than d_{86}. Such a ranking order, however, violates the TDC constraint for the structured query. In Q_1, the structural predicate Category = `'digestive system'` explicitly specifies the search scope to be citations related to digestive system, in which `pancreas` has a much higher DF than `leukemia`. As a result, `leukemia` has a higher IDF and d_{200} should be ranked higher than d_{86}.

TDC is not the only constraint that is violated for the integration of structured queries and keyword search. Length normalization, formalized as *Length Normalization Constraint* (LNC) [42], penalizes long documents' ranking scores.

The intuition is that long documents may contain duplicate information or be less specific on the topic, even though longer content usually means more occurrences of query terms, i.e., high TF. Thus, a long document with a high TF is not necessarily more relevant to the query. To penalize long documents, many IR ranking functions introduce *average document length* (denoted by $avgdl$): for a document d, if $\frac{|d|}{avgdl} > 1$, $tf(w, d)$ is divided by a factor to decrease the d's ranking score; if $\frac{|d|}{avgdl} < 1$, $tf(w, d)$ is multiplied by a factor to promote the d's ranking score. The absolute value of $avgdl$ will determine how much each document is penalized or promoted, and therefore change the ranking order.

Consider the following query that searches citations for clinical trials:

Q_2 :

```
SELECT   *
FROM     Citation
WHERE    Type = 'clinical trial' AND
         Content CONTAINS ''leukemia side effects''
```

The query searches citations classified to "clinical trials". Most clinical trial reports tend to be shorter than research papers, because the former usually only report clinical findings, while the latter include extensive materials, including background, ideas, experimental results and discussions. When a user's structured query specifies the focus to be clinical reports, these reports should not be particularly promoted or penalized. However, if we decouple the structured data and only apply IR ranking functions on the Content column, we will blindly mix citations of all types and clinical reports will be unfairly promoted. As a consequence, the ranking order of the matched citations will be skewed.

1.3 Expressive Context

The fundamental cause of various violations of IR heuristics is the misinterpretation of keyword statistics. Keyword statistics are basic ingredients of ranking heuristics. Conventionally, keyword statistics are collected over documents to characterize which document is more relevant to the keyword

query. On the integration of structured queries and keyword search, a structured query explicitly specifies a search scope, i.e., a subset of the document collection. When we decouple the structured data from unstructured text and only apply IR models on text, keyword statistics collected from the entire collection no longer characterize real ranking heuristics for the corresponding subset. Therefore, ranking schemes based on decoupling always violate some IR heuristics in the presence of structured queries.

In this dissertation, we present *context-sensitive ranking*, a ranking framework that integrates structured queries and keyword search and applies IR heuristics holistically. The key insight is that structured queries provide expressive search contexts. The basic semantics of a structured query is to return a subspace of data. The subspace defines a search context and depicts unique keyword statistics. These context-sensitive statistics are fed into conventional IR ranking functions to compute the results' ranking scores. Note that we do not change mathematical formulas of ranking functions, but only input statistics. Since conventional IR functions combine statistics in a way such that IR heuristics are satisfied[2], our result ranking too satisfies IR heuristics of the specified structured query.

Consider the previous query Q_1 that searches citations related to "digestive system". If we only collect DF (document frequency) over citations that are annotated by MeSH term "digestive system", rather than the entire collection, the parameters `idf(pancreas)` and `idf(leukemia)` will discriminate the two terms appropriately, and the TDC heuristic will be satisfied. Similarly, for the query Q_2 that only searches citations of "clinical trials", if the average document length `avgdl` is computed purely based on clinical reports, returned results will be normalized appropriately, and the LNC heuristic will be satisfied.

[2]Surprisingly, none of the conventional IR ranking functions satisfy all heuristics unconditionally, as proved in [42]. Since our ranking relies on mathematical formulas of existing ranking functions, they cannot satisfy all heuristics unconditionally either. However, the ranking framework guarantees that they do not violate more heuristics than existing IR ranking functions.

1.4 Challenges & Solutions

While improving ranking effectiveness and ensuring the consistency between ranking and IR heuristics, the new ranking framework radically changes query processing and poses new challenges on query efficiency. In conventional keyword search, the context is always fixed; it is the entire document collection. Statistics required by ranking functions are all precomputed at indexing time. For context-sensitive ranking, however, contexts are specified by users at query time and can be arbitrary subsets of the document collection. Therefore, collection-specific statistics (such as document frequency) have to be computed at query time.

Computing statistics at runtime is often expensive. First, in conventional query processing, a canonical optimization is to evaluate the most selective sub-expression first (e.g., the least frequent keyword), so that the number of intermediate results is dramatically reduced. A query optimizer may interleave keywords and structured predicates in a query plan to achieve optimality. This strategy, however, is not valid in our ranking framework. Since ranking now relies on a context specified by a structured query, the structured query must be evaluated completely, even if some of the keywords are highly selective. This restriction can lead to efficiency issues, when the structured query is not selective. Second, computing keyword statistics not only requires the evaluation of the structured query, but also aggregations. As we will show later, some statistics in ranking formulas demand aggregations of the documents in the context, which can be very expensive when the context is not small.

In this dissertation, we present an innovative reduction from computing statistics to answering aggregation queries and use view-based ideas to overcome efficiency challenges. At the core of the challenges is the computation of context-specific statistics, some of which (e.g., document frequency) demand aggregations over all documents satisfying the structured query. This is analogous to analytical queries in RDBMSs, which often need to aggregate parameters over a large number of tuples satisfying a structured query. An important technique used in RDBMSs to improve the performance of analytical queries is material-

ized views. A materialized view is a database object that stores the results of a query. When an online query matches a precomputed query and can use the materialized results, query evaluation does not need to start from scratch; query performance can be improved significantly.

Following the idea of OLAP techniques, we develop view-based algorithms and data structures to evaluate context-sensitive ranking efficiently. We reduce context-sensitive statistics to aggregation queries and present two forms of views to answer queries. We particularly concentrate on the problem of view selection, whose goal is to guarantee the performance of worst-case queries. We conduct a series of studies and experiments, and show that the proposed techniques are effective in guaranteeing the system's overall performance.

1.5 Organization

In this dissertation, we present context-sensitive ranking, a ranking framework for the integration of structured data and unstructured text. Our main contributions are a novel ranking framework, an innovative problem reduction, and a series of view-based algorithms that improve the efficiency of context-sensitive ranking.

In Chapter 2, we formally define data model for the integration of structured data and unstructured text, query semantics, and ranking model. While data model and query semantics share resemblances with data models in the database literature, our ranking model departs from existing ranking schemes in that it defines keyword statistics based on the structured query, leaving ranking formulas unchanged. The main merit of the ranking model is that we do not need to re-design ranking formulas based on various heuristics derived from application semantics.

In Chapter 3, we discuss evaluation of ranked query semantics, with an emphasis on the influence of context-sensitive ranking. We present a straightforward evaluation strategy based on existing query processing techniques. We analyze the complexity of query evaluation, and analytically demonstrate the

bottleneck of the straightforward evaluation.

While context-sensitive ranking provides an elegant integration of structured queries and text ranking, it raises a new problem on query efficiency. Chapter 4 provides an overview of our view-bases solution. We present an innovative reduction from computing statistics to answering aggregation queries. Using views to answer queries to improve efficiency is an old problem in database research. Instead of studying generic queries and views, in our problem setting, we concentrate on two forms of views. We present technical challenges, namely view selection, and a high-level problem formalization.

In Chapter 5, we discuss the first form of views: views as statistics caching. This form of views cache pre-computed collection-specific statistics directly. Only when a query matches the view exactly, the cached statistics are retrieved. The challenge is how to choose a number of views for caching.

In Chapter 6, we introduce another form of views: views as intermediate results. The idea is that even if the view dose not match the query precisely, if the view can be used as intermediate results, computing statistics does not need to start from scratch. We give formal definition of views, and study how to compute statistics using them. We also study view selection problem and propose selection algorithms.

Given that two forms of views serve the same goal, we conduct a comparative study in Chapter 7. We show that while essentially the two approaches for materialization are not different, with some special constraints, views as intermediate results can yield better performance. The experimental results present a guidance on under what circumstances one approach is superior to the other.

In Chapter 8, we present a diagnostic tool that inputs a user-specified maximal query execution time and outputs the context size threshold. The output threshold is fed into view selection algorithms to materialize views. The goal is that with these selected views, queries can finish within the maximal query execution time. Such an automatic end-to-end tool greatly improves usability and frees administrators from the pain of tuning parameters.

Experimental results on ranking quality and query efficiency are reported

in Chapter 9. We use data from PubMed for evaluation. We concentrate on three measures: ranking quality, query performance and view selection efficiency. The first metric validates the value of context-sensitive ranking. The last two metrics show the efficiency of our techniques.

We conclude the dissertation in Chapter 10. We also discuss several future directions along the line of this work.

Chapter 2

Data Model & Query Semantics

In this chapter, we formally define data model (Section 2.1), query semantics (Section 2.2), and ranking model (Section 2.3) for querying structured data and unstructured text.

2.1 Data Model

We model the combination of structured data and unstructured text as *structured documents*, whose definition is given in Figure 2.1. A structured document, denoted by $\mathtt{d}$, has two sections: (1) *document header* $\mathtt{d(h)}$ that wraps the structured data, and (2) *document content* $\mathtt{d(c)}$ that wraps the unstructured text.

$$
\begin{array}{lcl}
\mathtt{d} & ::= & \mathtt{d(h)}, \mathtt{d(c)} \\
\mathtt{d(h)} & ::= & \langle \text{record} \rangle \\
\mathtt{d(c)} & ::= & \langle \text{string} \rangle \ \big(, \langle \text{string} \rangle \big)^*
\end{array}
$$

Figure 2.1: Definition of structured document

The document content $\mathtt{d(c)}$ is modeled as a bag of (unordered) words, following the convention in information retrieval. A document header $\mathtt{d(h)}$ is a strong typed multi-field record. Its abstract syntax is given in Figure 2.2. A structured record consists of an unordered collection of fields. Every field has a description, i.e., $\langle \text{name} \rangle$. The value of a field can be an atomic value or a list of

atomic values. Atomic values include integers, floating-point numbers, strings, and etc. Values in a list can be order-sensitive or order-insensitive, whereas fields in a record are always order-insensitive.

$$
\begin{aligned}
\langle \text{record} \rangle &::= \langle \text{field} \rangle \ (, \langle \text{field} \rangle)^* \\
\langle \text{field} \rangle &::= \langle \text{name} \rangle : \langle \text{value} \rangle \\
\langle \text{name} \rangle &::= \langle \text{string} \rangle \\
\langle \text{value} \rangle &::= \langle \text{atom} \rangle \mid \langle \text{list} \rangle \mid \langle \text{record} \rangle \\
\langle \text{list} \rangle &::= \langle \text{value} \rangle \ (, \langle \text{value} \rangle)^* \\
\langle \text{atom} \rangle &::= \langle \text{integer} \rangle \mid \langle \text{double} \rangle \mid \langle \text{string} \rangle \mid \ldots\ldots
\end{aligned}
$$

Figure 2.2: Definition of document header

The definition of document header coincides with existing semi-structured data models such as XML [5] and JSON [3], but without nesting. It is expressive enough to capture a wide range of applications. For example, in email systems, an email header can be represented by three fields, ⟨sender⟩, ⟨receiver⟩, ⟨time⟩. ⟨sender⟩ and ⟨time⟩ are single-valued fields. ⟨receiver⟩ is a multi-valued field; each value in the field represents an email address. As another example, for academic citations in Table 1.1, structured data associated with citations can be represented by ⟨year⟩, ⟨author⟩ and ⟨category⟩, where the latter two represent lists of atomic values.

2.2 Query Syntax & Semantics

Let $\mathcal{D}$ be a collection of structured documents. A hybrid query $Q = Q_k|Q_s$ over structured documents consists of two sub-queries, the structured query Q_s over structured header $d(h)$ and the keyword query Q_k over unstructured text $d(c)$. Q_k is a conventional keyword query consisting of terms $Q_k = \{w_1, \ldots, w_n\}$. Q_s is a structured query whose formal syntax will be defined shortly. Each sub-query specifies a subset of documents.

Definition 2.2.1. *The keyword query* $Q_k = \{w_1, \ldots, w_n\}$ *over* $\mathcal{D}$ *specifies a subset of*

documents $Q_k(\mathcal{D}) \subseteq \mathcal{D}$ *such that for any document* $d_i \in Q_k(\mathcal{D})$*, there exists* $w_j \in Q_k$ *such that* $d_i(c)$ *contains* w_j*.*

The above semantics of Q_k follows the OR semantics in text search, i.e., a returned document contains at least one query term. While we focus on the OR semantics for keyword search all through this dissertation, the following discussion and proposed techniques can be easily extended to the AND semantics.

The structured query Q_s is a conjunction of predicates, each of which is over a field of the structured data. Its formal specification is given in Figure 2.3.

Q_s	::=	**SELECT** cid
		FROM $\mathcal{D}$
		WHERE (list)?
list	::=	cond (and cond)*
cond	::=	$d(h)$.field.atom op const
const	::=	string | integer | .....
op	::=	$= \mid > \mid < \mid \ldots$

Figure 2.3: Definition of the structured query Q_s

The specification of Q_s follows the SQL syntax, with a new construct $d(h)$.field.atom. The construct "$\cdot$" (dot) resembles the path expression in semi-structured data and references to an atomic value of a field. It syntactically eliminates joins for many-to-many and many-to-one relationships. For example, Category presents a many-to-many relationship between categories and documents. This relationship is usually maintained in RDBMSs in three tables: a Citation table, a Category table and a relationship table referencing to the two. A structured query searching citations from two categories need a three-way join between the Citation table and two aliases of the Category table, as shown as follows:

```
SELECT  C.did
FROM    Citation C
WHERE   EXISTS (
          SELECT C.did
          FROM Category C1
          WHERE C1.cid = C.cid AND C1.val = 'neoplasm'
        ) AND EXISTS (
          SELECT C2.cid
          FROM Category C2
          WHERE C2.cid = C.cid AND C2.val = 'brain'
        )
```

With the syntax introduced in Figure 2.3, the query has no relational joins, but path expressions in the `WHERE` clause:

```
SELECT   C.did
FROM     Citation C
WHERE    C.category.atom = 'neoplasm' AND
         C.category.atom = 'brain'
```

The dot ($\cdot$) and path expressions still require joins for query execution. But they facilitate problem formalization and reduction. Their effect will be clearer in following chapters.

Definition 2.2.2. *The structured query* Q_s *over* $\mathcal{D}$ *specifies a subset of documents* $Q_s(\mathcal{D}) \subseteq \mathcal{D}$ *such that for any document* $d_i \in Q_s(\mathcal{D})$, $d_i(h)$ *satisfies* Q_s.

Definition 2.2.3. *Given the query* $Q = Q_k|Q_s$ *and the document collection* $\mathcal{D}$*, the unranked result of* Q *is the intersection of* $Q_s(\mathcal{D})$ *and* $Q_k(\mathcal{D})$*, i.e.,* $Q(\mathcal{D}) = Q_s(\mathcal{D}) \cap Q_k(\mathcal{D})$.

The unranked semantics of Q is based on decoupling structured data $d(h)$ and unstructured text $d(c)$. It defines Q_s and Q_k on $d(h)$ and $d(c)$ respectively. However, as we argued earlier, the result ranking cannot be based on such decoupling, which will yield ranking functions violating IR heuristics. In next

section, we will discuss how to incorporate Q_s and Q_k into an IR-consistent ranking semantics.

2.3 Ranking Semantics

In this section, we define the ranking semantics of $Q = Q_k|Q_s$ over a collection of structured documents $\mathcal{D}$. We start by presenting a generic representation of ranking functions for conventional keyword queries. We then evolve it to the context-sensitive ranking function for the hybrid query Q. In the following, we use Q_t to denote conventional keyword queries, d_t to denote conventional documents, and $\mathcal{D}_t$ to denote a collection of d_t.

A variety of ranking functions were developed in information retrieval to rank documents with respect to keyword queries. In general, they combine keyword statistics to a single score to evaluate the relevance between a query Q_t and a document d_t. The statistics used in the ranking models can be classified into three categories: *query-specific*, *document-specific* and *collection-specific*:

- A *query-specific* statistic, denoted by $S_q(Q_t)$, is a statistic computed from the input query Q_t. For instance, query length counts how many terms in the query Q_t, which is only determined by Q_t.

- A *document-specific* statistic, denoted by $S_d(d_t)$, is a statistic computed from document d_t, e.g., term count of w_i in d_t. Every document has its unique document-specific statistics.

- A *collection-specific* statistic, denoted by $S_c(\mathcal{D}_t)$, is a statistic computed from the collection $\mathcal{D}_t$. They describe global characteristics of the entire collection. Conceptually, a collection-specific statistic is calculated by aggregating parameters of individual documents in the collection to a single value. For example, term count of w_i in the collection is calculated by summing up the number of occurrences of w_i in every document in the collection.

Table 2.1 summarizes atomic statistics used in a variety of ranking models, including vector space models (TF-IDF), language models and probabilistic

Table 2.1: Statistics used in ranking functions

Scope	*Statistics*	*Notation*
collection-specific	term count of w in the collection	$\mathtt{tc}(w, \mathcal{D}_t)$
	collection length	$\mathtt{len}(\mathcal{D}_t)$
	collection cardinality	$\lvert\mathcal{D}_t\rvert$
	document count (for term w)	$\mathtt{df}(w, \mathcal{D}_t)$
	unique term count in the collection	$\mathtt{utc}(\mathcal{D}_t)$
document-specific	term count in document	$\mathtt{tf}(w, d_t)$
	document length	$\mathtt{len}(d_t)$
	unique term count in document	$\mathtt{utc}(d_t)$
query-specific	term count in query (for w)	$\mathtt{tq}(w, Q_t)$
	query length	$\mathtt{len}(Q_t)$
	unique term count in query	$\mathtt{utc}(Q_t)$

relevance models. Note that some compound statistics used in the models can be computed by combinations of atomic statistics. For example, average document length ($\mathtt{avgdl}$) is calculated by collection length divided by collection cardinality:

$$\mathtt{avgdl} = \frac{\mathtt{len}(\mathcal{D}_t)}{|\mathcal{D}_t|}$$

Let $\mathcal{S}_q(Q_t)$ be a set of query-specific statistics for Q_t, $\mathcal{S}_d(d_t)$ be a set of document-specific statistics for d, and $\mathcal{S}_c(\mathcal{D}_t)$ be a set of collection-specific statistics for $\mathcal{D}_t$.

Given a query Q_t and a document $d_t \in \mathcal{D}_t$, a conventional ranking function $f(\cdot)$ takes as input statistics from $\mathcal{S}_q(Q_t)$, $\mathcal{S}_d(d_t)$, $\mathcal{S}_c(\mathcal{D}_t)$, and computes a score of d_t with respect to Q_t:

$$score(Q_t, d_t) = f\big(\mathcal{S}_q(Q_t), \mathcal{S}_d(d_t), \mathcal{S}_c(\mathcal{D}_t)\big) \tag{2.1}$$

Now we evolve Formula 2.1 to a function for context-sensitive ranking. For the hybrid query $Q = Q_k|Q_s$, the context-sensitive ranking views Q_s as a context specification that defines a set of documents $Q_s(\mathcal{D}) \subseteq \mathcal{D}$ of interest. Accordingly, the statistics used in the ranking function should be based on

Table 2.2: Evolved Statistics for Context-sensitive Ranking

Scope	*Statistics*	*Notation*
collection-specific	term count of w in the context	$\mathtt{tc}\big(w, \mathrm{Q_s}(\mathcal{D})\big)$
	context length	$\mathtt{len}\big(\mathrm{Q_s}(\mathcal{D})\big)$
	context cardinality	$\lvert\mathrm{Q_s}(\mathcal{D})\rvert$
	document count (for term w)	$\mathtt{df}\big(w, \mathrm{Q_s}(\mathcal{D})\big)$
	unique term count in the context	$\mathtt{utc}\big(\mathrm{Q_s}(\mathcal{D})\big)$
document-specific	term count in document	$\mathtt{tf}\big(w, \mathrm{d}(\mathrm{c})\big)$
	document length	$\mathtt{len}\big(\mathrm{d}(\mathrm{c})\big)$
	unique term count in document	$\mathtt{utc}\big(\mathrm{d}(\mathrm{c})\big)$
query-specific	term count in keyword query (for w)	$\mathtt{tq}(w, \mathrm{Q_k})$
	keyword query length	$\mathtt{len}(\mathrm{Q_k})$
	unique term count in keyword query	$\mathtt{utc}(\mathrm{Q_k})$

$\mathrm{Q_s}(\mathcal{D})$, rather than $\mathcal{D}$. Since query-specific and document-specific statistics are only determined by the input query and individual documents and have nothing to do with a document collection, they stay unchanged for the context-sensitive ranking. Only collection-specific statistics are updated based on a new collection of documents $\mathrm{Q_s}(\mathcal{D})$. The evolved statistics for query $\mathrm{Q} = \mathrm{Q_k}|\mathrm{Q_s}$ is shown in Table 2.2.

Given the hybrid query $\mathrm{Q} = \mathrm{Q_s}|\mathrm{Q_k}$ and the evolved statistics, the ranking score of a structured document $\mathrm{d} \in \mathrm{Q_s}(\mathcal{D})$, the d's ranking score is computed as:

$$\mathtt{score}(\mathrm{Q}, \mathrm{d}) = \mathrm{f}\Big(\mathcal{S}_\mathrm{q}(\mathrm{Q_k}), \mathcal{S}_\mathrm{d}\big(\mathrm{d}(\mathrm{c})\big), \mathcal{S}_\mathrm{c}\big(\mathrm{Q_s}(\mathcal{D})\big)\Big) \tag{2.2}$$

The ranking model uses the same computation function f as the conventional IR model, but different inputs—context-sensitive statistics. This enables the ranking to automatically inherit all merits of conventional IR models for relevance ranking and satisfy ranking heuristics specified by the structured query. Specifically, $\mathrm{Q_k}$ is a conventional keyword query, and query-specific statistics $\mathcal{S}_\mathrm{q}(\mathrm{Q_k})$ in Formula 2.2 is equivalent to $\mathcal{S}_\mathrm{q}(\mathrm{Q_t})$ in Formula 2.1. Since $\mathrm{d}(\mathrm{c})$ is the document's content and is equivalent to $\mathrm{d_t}$, document-specific statistics

$\mathcal{S}_d\big(d(c)\big)$ in Formula 2.2 is equivalent to $\mathcal{S}_d(d_t)$ in Formula 2.1. The only difference between Formula 2.1 and 2.2 is collection-specific statistics $\mathcal{S}_c$: $\mathcal{S}_c\big(Q_s(\mathcal{D})\big)$ in Formula 2.2 collects statistics from $Q_s\big(\mathcal{D}\big)$, a subset of the collection, whereas $\mathcal{S}_c(\mathcal{D})$ in Formula 2.1 collects statistics from the entire collection $\mathcal{D}_t$.

Example 2.3.1. *TF-IDF weighting is a well-known ranking model. Among its variants, the pivoted normalization formula [89] is considered to be one of the best performing vector space models and is widely used in many text search systems. Its mathematical representation is shown in Formula 2.3, where* s *is a constant and is usually set to 0.2. The other variables' meanings can be found in Table 2.1.*

$$\begin{aligned} score(Q_t, d_t) \quad = \quad & \sum_{w \in Q_t} \frac{1 + \ln(1 + \ln(tf(w, d_t)))}{(1-s) + s \cdot \dfrac{len(d_t)}{avgdl}} \cdot \\ & tq(w, Q_t) \cdot \ln \frac{|\mathcal{D}_t| + 1}{df(w, \mathcal{D}_t)} \end{aligned} \tag{2.3}$$

where $avgdl = \dfrac{len(\mathcal{D}_t)}{|\mathcal{D}_t|}$.

The statistics used in the pivoted normalization formula are classified as follows:

- $tq(w, Q_t)$ *is a query-specific statistic.*
- $tf(w, d_t)$, $len(d_t)$ *are document-specific statistics.*
- $df(w, \mathcal{D}_t)$, $|\mathcal{D}_t|$, $len(\mathcal{D}_t)$ *are collection-specific statistics.*

The context-sensitive version of the pivoted normalization formula for $Q = Q_k|Q_s$ *replaces every* $S_c(\mathcal{D}_t)$ *with* $S_c\big(Q_s(\mathcal{D})\big)$*, i.e.,*

$$score(Q, d) = \begin{cases} \sum_{w \in Q_k} \dfrac{1 + \ln(1 + \ln(tf(w, d(c))))}{(1-s) + s \cdot \dfrac{len(d(c))}{avgdl_s}} & \text{if } d \in Q_s(\mathcal{D}) \\ \cdot tq(w, Q_k) \cdot \ln \dfrac{|Q_s(\mathcal{D})| + 1}{df(w, Q_s(\mathcal{D}))} & \\ 0 & \text{if } d \notin Q_s(\mathcal{D}) \end{cases}$$

2.4 Related Work

Personalized ranking is a problem whose motivation is similar to context-sensitive ranking. It aims to bring personalizations to ranking so that users issuing the same query get different rankings for their own preferences, interests or search contexts. Personalized ranking has been extensively studied in IR community, especially in web search. A wide spectrum of personalization models were proposed, including users' profile information [88, 76, 100], and prior search behavior (e.g., query history, click logs) [92, 97, 80, 86]. They either re-rank top K results returned by standard search systems, or reformulate queries before sending to the search systems. The fundamental difference between these models and our ranking model is that they still rely on conventional ranking models and do not "personalize" underlying statistics.

Personalized PageRank [56, 62, 27, 64] is a personalization model for link analysis in web search. PageRank is a probabilistic model that simulates a pseudo user randomly walking on the web by following the links between web pages. The more likely the user is to visit a web page, the higher ranking scores that page has. A standard setting of PageRank is that the user may start from any page in the web. Instead of using the uniform distribution for all pages at the initial state, personalized PageRank uses a set of query or user-specific nodes as the random walk starting points. Since the initial state is query- or user-specific, PageRank scores are user- or query-sensitive, and must be computed at runtime. The main challenge is how to compute personalized scores efficiently, as online computations usually involve expensive fixpoint iterations over a very large graph. Among the proposed solutions, the algorithms in [62, 64] share the same spirit of the materialized view technique: some small subgraphs are precomputed in advance. Online computations use the materialized subgraphs to improve efficiency.

Personalized ranking has also been studied in database community. Paper [67] defines a preference model, where preferences are expressed as predicates associated with interest scores. Users' preferences are stored in their profiles and are used to rewrite SQL queries. Paper [6] defines a preference as an order of

two tuples when their attributes satisfy some conditions. Preferences are not commutative, and may conflict with each other. Given a selection SQL query, the goal is to compute an order of the retrieved tuples that is consistent with the predefined preferences as much as possible. Unlike the model in [67], this model does not personalize rankings for individual users.

We are not the first one to utilize domain-specific statistics to improve ranking effectiveness. For example, the clustered-based retrieval [72] clusters documents that are semantically related and uses statistics within individual clusters to improve the smoothing of language models. In machine learning, topic model was also studied [82, 21]. However, none of the existing work ever considers dynamic context/domain/topic specifications. Given that static contexts/domains/topics are chosen in advance, they cannot satisfy diverse users' needs. Our query model provides much more power to domain experts.

There is a wealth of literature on integrating structured queries with keyword search in XML databases. Indeed, our data model and query semantics share many resemblances with XML, except that our data model does not allow nesting. Three previous XML works stand out for considering structural aspects of the query for ranking statistics. In [98], each XML element name defines a context, and IDF is computed with respect to all other elements with the same name. In [44], a subset of element names are designated as defining contexts. Each other element belongs to its lowest context-defining ancestor. In [22], XPath expressions define search contexts, much like in our own work, though they use an inefficient evaluation strategy.

Content relevance ranking for XML are studied in IR community as well. We distinguish between our data model and structured retrieval. In the former structured data can be decoupled from text, whereas in the latter structures are embedded in text, e.g., <subject>this</subject>is a <bold>cat</bold>. Because of the difference in data modelling, our ranking model uses structured data as contexts, whereas the latter uses structure embedding to break text into smaller pieces and designs ranking schemes at finer granularity [26, 23, 32, 81].

2.5 Acknowledgments

Parts of Chapter 2 were published in ACM SIGMOD International Conference on Management of Data (SIGMOD), 2011, entitled "Context-sensitive Ranking for Document Retrieval". This is joint work with Yannis Papakonstantinou. The other parts are currently being prepared for submission for publication. This is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the two papers.

Chapter 3

Query Evaluation

In this chapter, we discuss naive query evaluation, and analyze its performance bottlenecks, which will be tackled in later chapters.

The evaluation of ranked queries involves two tasks: (1) compute unranked results and (2) compute the results' ranking scores and return a ranked list. For the first task, a canonical query optimization technique is to evaluate the most selective sub-expression first, so that the number of intermediate results is dramatically reduced. It means that in our problem setting, a query optimizer may interleave keywords from Q_k and structured predicates from Q_s in a query execution plan. Furthermore, for ranked semantics, computing top-K results is generally more important than returning complete results, because ranking scores measure the relevance between results and queries and users usually only examine top ranked results. Hence, computing ranking scores are often incorporated into the execution plan by an appropriate use of top-K algorithms [40, 94] to return top ranked results first.

In conventional ranking schemes that decouple structured data and unstructured text, computing ranking scores is computationally cheap because statistics needed by a ranking function are all pre-computed at indexing time, i.e., before receiving queries. Once a structured document is evaluated to satisfy the query, its ranking score is computed instantly by aggregating the precomputed statistics.

The ranking model defined in Formula 2.2, however, poses a computa-

tional challenge, since the search context and ranking statistics are specified by the structured query and must be computed at execution time. For the hybrid query $Q = Q_k|Q_s$, its search context is specified by the following SQL query:

```
SELECT   *
FROM     D
WHERE    EXISTS Qs(d(h))
```

This query must be fully evaluated, after which documents in the context are collected to compute collection-specific statistics, which are further used to compute ranking scores.

Example 3.0.1. *Consider the query* Q_1 *from Chapter 1 that searches citations containing "*`leukemia, pancreas`*". Document Frequency (DF) characterizes how discriminate a term is with respect to the papers published in the domain of "digestive system". This value cannot be computed until we evaluate the structured predicate* Category = '`digestive system`'*, and count how many returned citations contain the two terms respectively.*

A high-level evaluation scheme of $Q = Q_k|Q_s$ is shown in Algorithm 1. The first two steps evaluate the structured query Q_s and compute collection-specific statistics. Once all the statistics are prepared, the following steps further evaluate the keyword query Q_k and return ranked results.

Algorithm 1 poses efficiency issues when the structured query is not selective. First, query engine must evaluate Q_s, and cannot interleave Q_s's predicates and keywords, even if some keywords are highly selective (otherwise, statistics are not complete, and there will be no ranking scores!). Second, computing context statistics demands aggregations of the context collection. Online aggregations over a large set can be fairly expensive.

In the following, we dive into details of physical executions and quantitatively analyze its performance.

Algorithm 1: Query evaluation of $Q = Q_k|Q_s$

1 Evaluates the structured query Q_s over the document collection $\mathcal{D}$, and returns the search context $Q_s(\mathcal{D}) = \{d_1, \ldots, d_m\}$.
2 Scans $Q_s(\mathcal{D}) = \{d_1, \ldots, d_m\}$, and aggregates collection-specific statistics $\mathcal{S}_c\big(Q_s(\mathcal{D})\big)$ over $\{d_1(c), \ldots, d_m(c)\}$.
3 Evaluates the keyword query Q_k over $Q_s(\mathcal{D})$, and returns $Q_k\big(Q_s(\mathcal{D})\big) \subseteq Q_s(\mathcal{D})$ such that $\forall d \in Q_k\big(Q_s(\mathcal{D})\big)$, $d(c)$ satisfies Q_k.
4 $\forall d \in Q_k\ \big(Q_s(\mathcal{D})\big)$, computes the ranking score $f\Big(\mathcal{S}_q(Q_k), \mathcal{S}_d\big(d(c)\big), \mathcal{S}_c\big(Q_s(\mathcal{D})\big)\Big)$.
5 Returns top ranked results.

3.1 Physical Query Execution

The structured query Q_s consists of conjunctive predicates, each referencing to an atomic value in a field. For easy of exposition and representation, we assume all predicates test equality between an atomic value and a literal, and refer to the same field. The structured query Q_s is simplified to $Q_s = v_1 \wedge \ldots \wedge v_n$, where $v_1, \ldots, v_n$ are literals of a fixed field. A structured document d satisfies Q_s if $d(h)$ contains all the literals in the field.

By Algorithm 1, evaluation of $Q = Q_k|Q_s$ first materializes the context collection $Q_s(\mathcal{D})$ and computes required statistics accordingly. Thereafter evaluation is the same as conventional keyword queries. Let $L_w = \sigma_w(\mathcal{D})$ be the inverted list of keyword w, and L_v be the inverted list of literal v. Consider the query $Q = Q_k|Q_s = w_1 \wedge w_2|v_1 \wedge v_2$ and the TF-IDF ranking function in Formula 2.3. By query semantics in Section 2.2, the unranked result of Q is evaluated as the intersection of the structured query and the keyword search, i.e.,

$$L_{w_1} \cap L_{w_2} \cap L_{v_1} \cap L_{v_2}$$

To compute collection-specific statistics, the query plan must satisfy the following constraints:

1. Document frequency (DF) for w_i, $df\big(w_i, Q_s(\mathcal{D})\big)$, is the number of docu-

ments in the context that contain w_i, which is evaluated as $\left|\sigma_{w_i}(\mathcal{D}) \cap Q_s(\mathcal{D})\right|$. Therefore, the query plan must include $L_{w_1} \cap L_{v_1} \cap L_{v_2}$ and $L_{w_2} \cap L_{v_1} \cap L_{v_2}$, where the first expression computes document count for w_1, and the second expression computes document count for w_2.

2. Collection cardinality $|Q_s(\mathcal{D})|$ is evaluated as $|\sigma_{v_1}(\mathcal{D}) \cap \sigma_{v_2}(\mathcal{D})|$. Hence, the query plan must include $L_{v_1} \cap L_{v_2}$, and a COUNT aggregation on top of it.

3. Collection length $\mathtt{len}\big(Q_s(\mathcal{D})\big)$ requires a SUM aggregation on the lengths of the documents in the context, i.e., $\gamma_{\mathsf{sum}}(\sigma_{v_1}\big(\mathcal{D}) \cap \sigma_{v_2}(\mathcal{D})\big)$ where γ denotes an aggregation operator.

Putting the above constraints together, the execution plan of Q is shown in Figure 3.1, where $\cap_\gamma$ means "intersection with aggregation". At the bottom level, L_{v_1} and L_{v_2} are intersected to return documents in the context. Two aggregations, denoted by $\gamma_{\mathtt{count}}$ and $\gamma_{\mathtt{sum}}$, are performed upon $L_{v_1} \cap L_{v_2}$ to compute collection cardinality and collection length. The result of $L_{v_1} \cap L_{v_2}$ is further intersected with L_{w_1} and L_{w_2} respectively to obtain document frequency of w_1 and w_2. The final result is computed by the highest intersection operator.

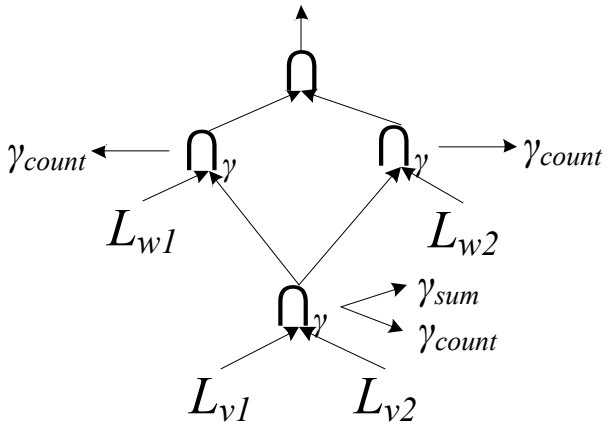


Figure 3.1: The physical execution plan of $Q = w_1 \wedge w_2 | v_1 \wedge v_2$

3.2 Cost Analysis

We introduce a simple cost model to quantify the cost of the physical execution. The goal of the model is not to estimate the cost as accurate as possible, but to analytically demonstrate the bottlenecks of the straightforward evaluation.

Cost Models for Inverted List Intersection and Aggregation

The core operation of the query plan in Figure 3.1 is the intersection of inverted lists. In standard text search systems, a simple representation of an entry in an inverted list is a pair of document ID and term frequency (TF), i.e., $\langle \texttt{docid}, \texttt{tf} \rangle$. Inverted lists are ordered by document ID so that two lists can be merged efficiently. A simple cost model for the merge join is $|L_i| + |L_j|$, where L_i and L_j are two inverted lists.

In addition to the standard merge join, there are many sophisticated optimization techniques that leverage low join selectivity [37, 17, 38, 13, 14, 18]. These algorithms use the total number of comparisons as the measure of the algorithm's complexity and aim to use the minimum number of comparisons ideally required to establish the intersection. The benefit of these algorithms is that when join selectivity is low, the execution of intersections can be very close to $|L_i \cap L_j|$, which is much faster than $|L_i| + |L_j|$.

A simple technique belonging to this category and widely used in many text search systems is *skip pointers* [78]. Specifically, inverted lists are partitioned into segments and skip pointers are maintained to jump between consecutive segments. When two inverted lists are scanned, if the current document ID of the first inverted list does not fall in the segment of the second inverted list, the whole segment of the second inverted list can be skipped.

Example 3.2.1. *Consider two inverted lists of integers* $L_1 = \{1, \ldots, 100\}$ *and* $L_2 = \{100, \ldots, 199\}$. L_1 *and* L_2 *are partitioned into segments:*

$$L_1 = \{1, \ldots, 20\}\{21 - 40\} \ldots \{81 - 100\}$$
$$L_2 = \{100 - 119\}\{120 - 139\} \ldots \{180 - 199\}$$

When the first element of L_2 *(i.e., 100) is processed, only last segment of* L_1 *is processed. All previous segments are skipped, because 100 is out of their ranges. After the first segment of* L_2 *is processed, all the following segments of* L_2 *are skipped as well, because* L_1 *has reached the end. Overall, only two segments from* L_1 *and* L_2 *are processed. The complexity of the execution can be characterized by* $|L_1 \cap L_2|$*, rather than* $|L_1| + |L_2|$.

Let M_0 be the number of entries in one segment, N_i^o be the number of segments in L_i whose ranges overlap with some segment(s) in L_j, and N_j^o be the number of segments in L_j whose ranges overlap with some segment(s) in L_i. Then the cost of the intersection with skip pointers is

$$M_0 \cdot (N_i^o + N_j^o)$$

Since $N_i^o \leqslant \frac{|L_i|}{M_0}$ and $N_j^0 \leqslant \frac{|L_j|}{M_0}$, we have

$$M_0 \cdot (N_i^o + N_j^o) \leqslant |L_i| + |L_j|$$

Therefore, the cost model of the intersection is:

$$\mathrm{cost}(L_i \cap L_j) = M_0 \cdot (N_i^o + N_j^o)$$

As we can see, when the join selectivity is low, N_i^o or N_j^o can be very close to the number of results. The algorithm's complexity can be approximated by a linear function with respect to the result size $|L_i \cap L_j|$.

An aggregation over a list requires a full scan of the elements in the list. Hence, the cost model of the aggregation is:

$$\mathrm{cost}\Big(\gamma\big(Q_s(\mathcal{D})\big)\Big) = \Big|\bigcap_{v_i \in Q_s} L_{v_i}\Big|$$

Analysis

Intersecting inverted lists is generally considered to be efficient. The skip pointer optimization and other techniques [37, 17, 38, 13, 14, 18] improve the efficiency significantly when the join cardinality is low, as many segments can be skipped. In particular, when $|L_i|$ is orders of magnitude smaller than $|L_j|$, L_i's entries span at most $|L_i|$ segments of L_j, i.e., each entry in L_i falls in a

separate segment of L_j. In such a case, the cost for the intersection of L_i and L_j is $|L_i| + |L_i| \cdot M_0$, which can be much less than $|L_i| + |L_j|$.

However, intersecting very long inverted lists may not cheap [30]. In particular, when the join cardinality is not low, the intersection cannot take advantages of skip pointers and all segments must be scanned. In these cases, the cost of the context materialization is bounded by $\sum_{v_i \in Q_s} |L_{v_i}|$.

While inverted-list intersections in conventional keyword query evaluation can start from the most selective keyword, the evaluation of context-sensitive ranking must fully materialize the context. Intuitively, the context size tends to be fairly large, because the purpose of the context specification is to define a general search scope, rather than to filter out specific information as the keywords in conventional queries.

In standard text search systems, when the keywords in the query are not selective and the result size (i.e., the join cardinality) is very large, top-K processing techniques have been developed to reorder inverted lists so that only a small fraction of the lists are processed to generate top K results. This strategy, however, is not applicable for context-sensitive ranking before all collection-specific statistics are computed. The performance of the query will still be bounded by the complexity of the context materialization.

In addition to the cost of intersections, the cost of aggregations is proportional to the context size which is less than $\sum_{v_i \in Q_s} |L_{v_i}|$.

Proposition 3.2.1. *The cost of a context-sensitive query* $Q = Q_k|Q_s$ *is bounded by* $O(\sum_{v_i \in Q_s} |L_{v_i}|)$ *in the worst case.*

The above analysis shows that context-sensitive ranking can be fairly expensive when the structured query is not selective and the context is fairly large. The performance of $Q = Q_k|Q_s$ can be orders of magnitudes slower than the unranked query $Q_t = Q_k \cap Q_s$, which by query semantics presents the same unranked results as $Q = Q_k|Q_s$. The performance drop makes context-sensitive ranking unacceptable, as the ranking sacrifices efficiency too much in order to deliver a better ranking quality.

3.3 Related Work

Query evaluation for the integration of structured queries and keyword search has attracted a lot of research attention in the past few years. Considerable works have been dedicated to relational and XML databases respectively. In XML databases, much effort has been devoted to query semantics and efficient evaluation [33, 94, 11, 10, 45, 93]. Though query syntaxes and semantics considered by these works are not strictly the same, they all interleave XML structured queries (in forms of XPath or XQuery) and keyword search (through special query constructs, e.g., `contains`). It is commonly agreed that query evaluation is handled by a generic optimizer that processes structured queries and keyword queries uniformly. Technical contributions involve how to combine the evaluation algorithms of the two kinds of queries seamlessly.

XML keyword search is a line of works parallel to combining structured queries and structured search [49, 101, 102, 73, 91, 69, 31]. Though queries considered in these works are pure keyword queries, the structure of XML documents raises unique opportunities for structured search. Specifically, they define query semantics to be a fragment of XML documents containing all keywords. This is formalized by the notion of *Lowest Common Ancestor* (LCA). Various LCA-based semantics were proposed and their evaluation algorithms are studied. In addition to query semantics and evaluation algorithms, XML tree structures are also studied to improve other aspects of search. For example, paper [85] studies query evaluation over virtual views of XML. The major difference with the LCA-based keyword search is that given the view definition, the returned elements are fixed, whereas the returned results of the LCA-based semantics can be arbitrary fragments in XML tree. Papers [73, 74] study the problem of how to return results with more semantics. They leverage XML structures to infer relevant results by analyzing matched patterns.

Keyword search is also studied in relational databases. DISCOVER [61], DBXplorer [8] and BANKS [20] are pioneer systems that support keyword search in relational databases. Their query semantics is that results of keyword queries are sets of tuples that contain all the keywords and are connected through

primary keys and foreign keys. Later works follow this semantics and further focus on two aspects: efficiency [61] and effectiveness [71, 75].

Among all the above systems and algorithms, top-K processing is an important technique in keyword-oriented search. Since ranking mechanism is introduced, users are likely to only examine a few relevant results. There is no need to compute complete results. The philosophy of top-K processing is to focus on those "promising" results, so that processing can stop as long as first K results are available. The Threshold Algorithm [40] is the most well-known instance in this category. In XML databases, several works [94, 65, 31] extend the idea of TA and adapt it for the integration of XML structured queries and keyword search. Top-K problems are also studied from other dimensions: monotonic ranking functions [41, 29, 19, 77], non-monotonic ranking functions [99, 75], existence of materialized views [60, 36, 15], and top-K joins [63, 59].

All the existing top-K query processing techniques rely on the assumption that keyword statistics are prepared in advance. Under this assumption, top-K processing naturally fits in query evaluation, because once a result is computed, its ranking score is immediately computed from pre-computed scores or statistics. In our ranking framework, however, top-K processing is only useful when all statistics are available and is not helpful in relieving the performance pressure when the structured query is not selective.

3.4 Acknowledgments

Chapter 3 was published in IEEE International Conference of Data Engineering (ICDE) 2010, entitled "Supporting Top-K Keyword Search in XML Databases", and in ACM SIGMOD International Conference on Management of Data 2011, entitled "Context-sensitive Ranking for Document Retrieval". Both papers were joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the two papers.

Chapter 4

Computing Statistics Using Views: Overview

While context-sensitive ranking provides a clean integration of structured queries and relevance ranking, current database or text search systems cannot efficiently evaluate queries specifying very large contexts. The technical challenge is to maintain query efficiency as close as possible to query semantics using conventional ranking schemes.

At the core of our solution to overcoming the efficiency issue is to reduce the computation of collection-specific statistics to aggregation queries. Computing collection-specific statistics essentially involves online aggregations. For instance, document frequency $\mathrm{df}(w)$ is equivalent to a COUNT aggregation over documents containing the term w and satisfying the structured query Q_s. Similar problems were encountered in OLAP [46], an approach to quickly analyze multi-dimensional data. A large body of OLAP queries involve expensive aggregations which must be answered in a short time. The most important mechanism in OLAP that allows such performance is the use of materialized views. A materialized view is a database object that contains the results of a query. The high level idea is that if the view is pre-computed in advance and online queries can use it to save computations, query evaluation does not need to start from scratch and query performance can be improved significantly.

In this chapter, we provide an overview of our view-based techniques.

We discuss two forms of views we exploit in this dissertation to improve query performance. We further analyze technical challenges and present their formalization and quantification. Algorithms and data structures will be discussed in the following chapters.

4.1 Two Forms of Views

A materialized view is a database object that contains the results of a query. Intuitively, a materialized view can be beneficial to queries in two forms. First, the view stores collection-specific statistics of a context-sensitive query. In this case, online evaluation simply retrieves statistics. Second, though the view does not contain the result directly, it can be used as intermediate results. Query evaluation does not need to start from scratch and hence is efficient than the approach without using any views. In the following, we refer to the first form as *views as caching statistics* and the second as *views as intermediate results*.

The nature of the two forms of views presents unique trade-offs: views as statistics caching provide best online performance, because there is basically no online computation. However, it potentially involves more offline computations when we materialize views. Views as intermediate results, on the other hand, need some online computations, because they are "half-baked" results and further computations are needed to compute final statistics. As we will see later, offline computations in this case are usually lighter than that of views as statistics caching.

When we consider views as intermediate results, a special question arises: what kinds of views can be used to answer the query? Since in this case views do not match queries precisely, special attention must be paid to determine whether we can use this view to answer the query. This problem is also known as *view usability*. An intuition to the question is that the view must preserve all information required by the query for evaluation.

4.2 Performance Goal

The goal of using views is to improve query performance. In this dissertation, we set our performance goal as guaranteeing the performance of worst-case queries. Query performance is measured by online execution time. In text search systems, response time is the most critical factor for user experience, as users are usually expecting instant responses [24, 70]. Ideally, by using views to answer queries, we want to ensure that all queries can be executed within a fixed amount of time.

We assume that all queries are run individually, i.e., a single query environment. We transform the performance goal into two parameters, *context size* and *view size*. Context size is used to quantitatively characterize query execution time. As discussed in 3.2, computing collection-specific statistics involves two steps: intersecting inverted lists of conjunctive predicates and aggregating documents in the context. Both steps' complexities can be approximated by the context size.

- When the predicate inverted lists are short, aggregation queries that compute statistics are evaluated from scratch. Evaluation of intersections is efficient. Also since the result size must be small, the cost of online aggregation is small too. Overall, the query execution time can be guaranteed.

- When the input inverted lists are long and the context size is large, both intersections and aggregations are expensive. We refer to this type of queries as worst-case queries. Every worst-case query must be answered by at least one view so that its execution time is within the specified limit.

- When the input inverted lists are long but the context size (which is the intersection of the inverted lists) is small, statistics will be computed from scratch. Aggregation cost is small, because of the small context size. Evaluation of intersections is also efficient in this case, because the join selectivity is low and various techniques are designed to reduce the complexity of intersections as close as possible to the result size, as discussed in Section 3.2. Overall, the execution time of this type of queries can also be guaranteed.

Based on this modeling, a structured query specifying a context greater than a certain threshold cannot finish within a short time. There must be at least one view to improve its performance. Queries specifying small contexts are evaluated from scratch and statistics are computed at run time.

In Chapter 8, we will present a diagnostic tool that selects a context size threshold automatically. Specifically, the tool inputs a user-specified maximal query execution time, and returns a context size threshold. The tool is based on the analysis that the connection between query execution time and context size fits in a linear function. The selected threshold is further used to select views.

In addition to the context size, online computation cost is the second parameter that needs to be constrained. The goal of using views to compute statistics is to improve search performance and response time. If the online computation is too expensive, the performance goal will not be achieved. Specifically, for views as statistics caching, the performance goal can always be achieved, because there is basically no online computation, but directly retrieval. For views as intermediate results, as we will see later, its online computation cost is directly related to the view size (in terms of number of tuples). Therefore, we require that views be smaller than a certain threshold. (Views as statistics caching can be viewed as a special case, where each view has only one tuple storing different statistics in different columns.)

4.3 View Selection

Given a context size threshold T_C and a view size threshold T_V, view selection aims to select views to answer queries whose context sizes are greater than T_C. We formalize the view selection problem as follows:

Problem Statement 4.3.1. *Given a context size threshold T_C (in terms of percentage of the data size) and a view size threshold T_V (in terms of number of tuples in the view), find a set of views $\mathcal{V} = \{V_{K_1}, V_{K_2}, \ldots\}$ such that*

1. $\forall V_{K_i} \in \mathcal{V}$, $ViewSize(V_{K_i}) \leqslant T_V$.

2. *For every possible context specification* Q_s*, if* $|Q_s(\mathcal{D})| \geqslant T_C \cdot |\mathcal{D}|$*, then* $\exists V_{K_j} \in \mathcal{V}$ *such that* V_{K_j} *can be used to compute the* Q_S*'s collection-specific statistics.*

View selection algorithms will be dependent on the view forms. For views as statistics caching, since a view caches the query's statistics directly, the essential problem is to identify target queries (i.e., worst-case queries). For views as intermediate results, since they maintain intermediate results, worst-case queries may not be identified explicitly. However, it must be guaranteed that these queries must be covered by at least one view.

It is worth noting that our problem formalization of view selection is different from that of conventional RDBMSs. In RDBMSs, view selection is formalized as a combinatorial problem: given a query workload and some resource constraints, find a set of views to materialize so that the performance improvement for the workload is maximized. Our goal of view selection, however, is to improve the performance of worst-case queries. Such a formalization is based on the following considerations. First, no query workload is available for this new query model. Second, even if the query workload is available, it is still dangerous to only rely on it. Unlike RDBMS queries which are fairly stable, queries of keyword search systems are typically unpredictable and may evolve over the time.

4.4 Acknowledgments

Chapter 4 is currently being prepared for submission for publication, which is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the paper.

Chapter 5

Views as Statistics Caching

In this chapter, we consider using views as statistics caching. A view stores collection-specific statistics of the context $Q_s(\mathcal{D})$. When a user's structured query is an exact match of the view definition, pre-computed statistics are directly retrieved to compute ranking scores. When no view definitions match the structured query, statistics must be computed online.

Views as statistics caching provide best online performance, because there is no online computation, but only retrieval. However, there is basically an infinite number of queries. Since each view can only serve one query, we must carefully choose which views to materialize. In following, we discuss algorithms to select views and further optimizations.

5.1 Selection of Views as Statistics Caching

To select views for statistics caching, the key is to identify target queries. We quantify the cost of a query by its context size. All queries specifying contexts larger than the threshold $T_C \cdot |\mathcal{D}|$ are the targets for materialization.

In our query semantics, a structured query Q_s is a conjunction of predicates on fields in structured data. Q_s is evaluated by intersecting the inverted lists of the predicate values. The result size, which is the context size, is the number of occurrences of predicate values.

Example 5.1.1. *Consider the structured query* `Category.atom = 'neoplasm' AND Category.atom = 'brain'`*, which specifies the context as a collection of documents belonging to the categories of "neoplasm" and "brain". The context size is the number of documents in which the values "neoplasm" and "brain" co-occur.*

Finding value combinations co-occurring in many documents is equivalent to mining association rules of values such that their supports, in terms of the number of documents in which they co-occur, are greater than $T_C \cdot |\mathcal{D}|$. For each combination returned by the mining algorithms, the corresponding query specifies a large context and its collection-specific statistics will be materialized as a tuple.

A number of algorithms for association rule mining (ARM) have been proposed in the data mining literature, e.g., Apriori [7], FP-growth [54], Eclat [105]. Given a set of items and a set of transactions, the algorithms scan the transaction set one or more times and return combinations of items whose occurrences in transactions (called support) are greater than a pre-specified threshold (called minimum support). In our problem setting, an item is mapped to a field value, and a transaction is mapped to a document. An association rule mining algorithm returns a set of values combinations, whose supports are greater than $T_C \cdot |\mathcal{D}|$.

5.2 Optimizations based on Dependencies

A view selection algorithm based on conventional ARM algorithms treats predicate values as a flat set, and does not leverage their relationships. In applications such as PubMed [4], application-level optimizations can be used to achieve better selection efficiency.

Databases usually contain various constraints enforced by applications. These constraints present additional information that can be used for various optimizations. In this section, we discuss a special form of constraints—functional dependencies—to optimize mining value combinations for view selection.

5.2.1 Illustrative Example

We start by illustrating an example in PubMed. MeSH terms in PubMed are structured data associated with citations, indicating which categories or domains a citation belongs to. Since each citation may have one or more MeSH terms, the MeSH field is a list. To select views as statistics caching, an ARM algorithm needs to compute which MeSH terms co-occur frequently.

MeSH terms in PubMed are highly co-related. Specifically, all MeSH terms represent a biomedical ontology and are organized in a hierarchy, as shown in Figure 1.1. Given such a hierarchy structure, an important property is that when a citation is annotated with the MeSH term m_1, it is automatically annotated with all the m_1's ancestors $\{m_1^a, \ldots, m_n^a\}$. For any combinations containing m_1, the support stay unchanged after adding one or more ancestors of m_1 to the combination. In general, we have:

$$supp\big(\{m_1\} \cup S\big) = supp\big(\{m_1\} \cup S \cup M^a\big), \;\; M^a \subset \{m_1^a, \ldots, m_n^a\}$$

where S is a set of MeSH terms. In other words, if the support of $\{m_1\} \cup S$ is greater than the threshold, the support of the new combination $\{m_1\} \cup S \cup M^a$ is also greater than the threshold. No additional computation is required.

The above observation implies an optimization that can dramatically reduce the computations of mining association rules. To discover high-support combinations of MeSH terms, we only need to consider combinations in which no two MeSH terms have an ancestor-descendant relationship. When a combination $\{m_1, m_{j_1}, m_{j_2}, \ldots, m_{j_k}\}$ is determined to have high support, the combination is automatically expanded by adding all the m_1's ancestor into it, i.e., $\{ m_1, m_1^a, \ldots, m_n^a, m_{j_1}, \ldots, m_{j_k} \}$. Since citations annotated with m_1 is also annotated with $\{m_1^a, \ldots, m_n^a\}$, the expanded combination also has a high support.

5.2.2 Mining Column Combinations with Functional Dependencies

The hierarchical structure of an ontology can be generalized to *functional dependencies*. A functional dependency (FD) is a constraint between two sets of

attributes in a relation from a database. Given a relation R and a set of attributes X, X is said to functionally determine another attribute Y, denoted by $X \to Y$, if and only if, each X value is associated with precisely one Y value in R. In simple words, if the X value is known, the Y value is also known.

Functional dependencies are common in real-life examples. For instance, consider two columns ZipCode and State in a relational table. Given a zip code value, its state location is fixed, i.e., ZipCode $\to$ State. Such constraints play an important role in conventional database design and query optimization. In our problem setting, the goal is to discover high-support column combinations. When a column set X fully determines Y, we only need to consider X for the mining purpose. Any high-support combinations including X can be expanded by adding a subset of Y.

In the PubMed example, hierarchical relationships in an ontology can be viewed as a special form of functional dependencies. For a MeSH term m_1, since all the citations annotated with m_1 are also annotated with its ancestor m_1^a, the occurrence of m_1 implies m_1^a. Theoretically speaking, m_1 does not fully determine m_1^a, because the absence of m_1 indicates neither the occurrence nor the absence of m_1^a. However, for the purpose of mining value combinations, we are only interested in whether m_1 appears in a combination whose support is greater than the threshold: if a combination with m_1 has a high support, m_1^a is automatically added to the combination; otherwise, m_1 is replaced with its ancestor m_1^a, and the support of the new combination is computed.

Algorithmically, we process functional dependencies in a generic way. All functional constraints, whether defined as conventional dependencies or derived from the ontology, are organized into a dependency graph: every node in the graph represents a value or a set of values. An edge points from u to v, if u and v are on left and right hand sides of a constraint. For a value combination p,

1. for every functional dependency $X \to Y$ and p fully contains X, add values in Y into p.

2. Go to step 1, until no more values can be added based on the dependency graph.

The pseudo code of the mining algorithm that leverages functional dependencies is shown in Algorithm 2.

5.3 Limitations of Original Association Rule Mining

Selecting views as statistics caching relies on association rule mining (ARM). ARM is a classical problem in the data mining literature, and many algorithms have been proposed, e.g., e.g., Apriori [7], FP-growth [54], Eclat [105]. These algorithms can achieve good performance in many scenarios.

While a lot of progress has been made for ARM, the problem is intrinsically expensive in some cases. In addition to well-known factors—data size and minimal support threshold—we identify another factor that is common in our problem setting but rare in conventional scenarios such as retail transactions. The ARM problem is expensive when each transaction (or a document in our problem setting) involves a large number items. In retail transactions, one transaction usually involves at most dozens of items. The average number of items in a transaction is not large. This means that the probability of a combination of items appearing together is not very high, especially when the combination's size is close to the average transaction size. For example, assume the average number of items in a transaction is 10. Then the number of high-support combinations with 8 items must be small, because it is unlikely all these transactions are highly similar. Hence, as the ARM algorithms proceed to mine combinations with larger sizes, the number of high-support combinations and candidates decreases very fast; the algorithms will terminate soon.

When each transaction involves a large number of items, however, the number of high-support combinations with 8 items can still be very high: it is not uncommon for 8 items appearing together in 100-item transactions. Hence, the number of high-support combinations with 8 items will still be very high.

While large transactions are uncommon in retail transactions, they are common in IR applications. For example, using keywords in abstracts to specify contexts is supported in our query semantics for context-sensitive ranking. The

Algorithm 2: Mining column combinations using functional dependencies

input : A dependency graph and a cardinality threshold $T_C \cdot |\mathcal{D}|$ on value combinations

output : A set of values combinations whose supports are greater than the threshold

```
1  rs ← ∅ ;
2  Build a lattice graph defined by the standard ARM algorithm ;
3  Traverse the lattice using breadth-first search ;
4  foreach combination C in the lattice do
5      foreach dependency X → Y in the dependency graph do
6          if C contains X and Y then
7              The support of C can be derived from another combination
               that has been processed before, and therefore skip C and
               move to the next combination in the lattice.
8          end
9      end
10     Compute the C's support supp(C) ;
11     if supp(C) > T_C · |D| then
12         rs ← rs ∪ {C} ;
13         foreach dependency X → Y do
14             if C contains X then
15                 foreach subset Y_s ⊆ Y do
16                     rs ← rs ∪ {C ∪ Y_s} ;
17                 end
18             end
19         end
20     end
21 end
22 return rs
```

average number of terms in the abstract in PubMed is 70, much larger than everyday retail transactions. Using terms in citations' full-text content is also an option for context specification. In this case, the average number of terms is 1000. In both cases, ARM algorithms are expensive, and may not be efficient enough for our view selection purpose.

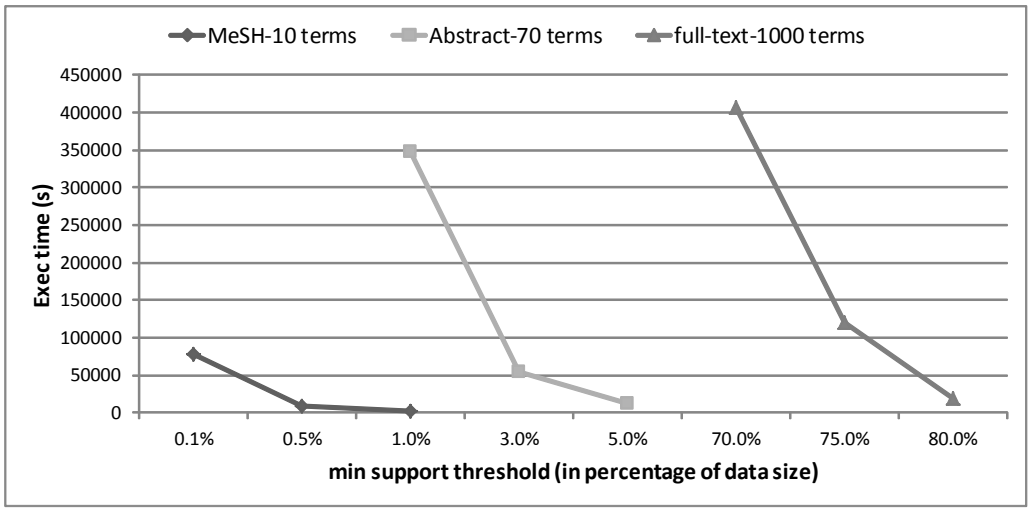


Figure 5.1: Efficiency comparison of ARM

Figure 5.1 shows execution times of ARM algorithms in different transaction sizes. The data set we use is PubMed. We consider three types of transactions: (1) MeSH terms. Every document is annotated with one or more MeSH terms. The average number of MeSH terms in a document is 10. (2) Abstract terms. The average number of abstract terms in a document is 70, significantly larger than MeSH terms. (3) Full-text terms. The average number of terms in the full-text content is 1000.

As we can see from the figure, when each document (or transaction) is small, the minimal support threshold can go very low. It is widely acknowledged that the lower the threshold, the more high-support combinations will be generated and thus the more computations are needed. And yet, the minimal threshold for MeSH terms can go as low as 0.1% of the data size, and execution time is still acceptable. However, when we consider abstract, the execution time is significantly larger. Note that it increases exponentially! When we consider an

extreme case, i.e., the full-text content, the execution time is very high, even if the minimal support threshold is uncommonly high.

Maintaining a low support threshold (or context size threshold) is critical to query execution time. The higher the context size threshold is, the fewer views are materialized, and therefore the more expensive queries are evaluated from scratch. As we will see in later Chapter 8, when the context size threshold is around 0.5% of the data size, materialized views can ensure that queries can finish within a few hundred milliseconds, which is acceptable for text search systems. When the context size threshold is as high as 70% of the data size, queries can run up to tens of seconds, which is considered unacceptable for text search systems.

The limitations of ARM-based selection algorithms lead us to the second option: views as intermediate results. Intuitively, views as intermediate results do not directly compute statistics, and thus should require cheaper offline computations. However, views as intermediate results raise new problems because online computations are not trivial and must be bounded. In the next chapter, we present techniques for views as intermediate results. We will show that this technique can leverage unique opportunities and present better performance when the opportunities are available.

5.4 Acknowledgments

Chapter 5 is currently being prepared for submission for publication, which is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the paper.

Chapter 6

Aggregation Views as Intermediate Results

In this chapter, we present using views as intermediate results to compute statistics. We first present the view definition. Then we study *view usability*, i.e., how to find a match between a view and a query and how to use the view to answer the query. We further analyze the complexity of answering queries using these views, to quantify our performance goal (Section 4.3) in this setting. Finally, we discuss how to select aggregation views to satisfy the performance goal.

6.1 From Statistics to Aggregation Queries

Given a context-sensitive query $Q = Q_k|Q_s$, collection-specific statistics are defined based on the search context specified by the structured query Q_s. A collection-specific statistic $S_c\big(Q_s(\mathcal{D})\big)$ is computed by aggregating parameters of individual documents returned by the structured query. It is equivalent to an aggregation query:

$\mathbf{Q_a(Q_s)}$:

```
SELECT  AggrFuc(parameter)
FROM    D
WHERE   field_1.atom = v_1 AND ...AND field_n.atom = v_n
```

in which `AggrFuc` is an aggregation function and `parameter` is a parameter of individual documents.

Query $Q_a(Q_s)$ is not a standard SQL aggregation query, because as discussed in Section 2.2, path expressions syntactically remove joins. But the physical execution may still involve joins. Specifically, when a predicate is over a field that is a list, a join is required because of many-to-one or many-to-many relationships. For such multi-way join aggregation queries, it is not straightforward what kinds of views to materialize to facilitate query evaluation.

We use the pivot technique to transform the documents' structured data to a pivot table to facilitate problem modeling. The document collection is modeled as a wide sparse table T. Every row corresponds to a document d_i. Every field f of document header $d(h)$ is mapped to one or more columns:

1. if the field's type is atomic, T populates one column col_f for it. The entry in row d_i and column col_f is the atomic value of field f in $d_i(h)$.

2. if the field's type is list, T populates M columns $\{col_{f:v_1}, \ldots, col_{f:v_M}\}$, each of which corresponds to a distinct value in the field's domain. The entry in row d_i and column $col_{f:v_i}$ is 1 if the field f contains the value v_i in the list; otherwise, the entry is 0.

Table 6.1 shows the pivot table of the structured data of PubMed. The Year field is integer-typed, and thus corresponds to one column in the pivot table. The Category field is list-typed. Hence, each distinct value in the field, such as "leukemia" and "digestive system", is mapped to a separate column. Citation d_{86} contains MeSH term "digestive system" in its Category field, and thus the corresponding entry is 1. d_{86} does not contain MeSH term "leukemia" in the Category field. Hence, its corresponding field is 0.

Given the pivot table T, the aggregation query that computes the statistics is transformed to the following SQL query:

Table 6.1: Pivot table of structured data $d(h)$

CID	$len(d(c))$	$tf(w, d(c))$	Year	leukemia	digestive system	...
86			2006	0	1	...
200			2004	1	1	...

$\overline{\mathbf{Q}}_{\mathbf{a}}(\mathbf{Q}_{\mathbf{s}})$:

```
SELECT  AggrFuc(parameter)
FROM    T
WHERE   col_{f_1} = v_1 AND ...col_{f_m} = v_m AND
        col_{f_{m+1}:v_{m+1}} = 1 AND ...col_{f_n:v_n} = 1
```

In the query, the first m predicates are over atomic fields and the remaining are over list-typed fields. Compared with the old query $Q_a(Q_s)$, the new query $\overline{Q}_a(Q_s)$ transforms the last $n - m + 1$ predicates by moving predicate literals to column names and replacing the literals with 1. This is because after pivoting, a list-typed field is expanded to multiple field-value columns.

Column `parameter` in $\overline{Q}_a(Q_s)$ represents a parameter of individual documents required by collection-specific statistics. For ease of exposition, we populate them as columns in the pivot table as well. For example, Table 6.1 populates two parameter columns, document length $len(d(c))$ and term frequency $tf(w, d(c))$.

Example 6.1.1. *Consider the PubMed database and a user's structured query specifying two MeSH terms* `neoplasm` *and* `brain` *for the* **Category** *field. After pivoting the* **Category** *field into multiple columns, two parameters, ollection length* $len(Q_s(\mathcal{D}))$ *and collection cardinality* $|Q_s(\mathcal{D})|$*, are translated to a SUM and a COUNT aggregation on the pivot table* T *as shown as follows:*

$len(Q_s(\mathcal{D}))$:

```
SELECT  SUM(len(d(c)))
FROM    T
WHERE   col_neoplasm = 1 AND col_brain = 1
```

$|Q_s(\mathcal{D})|$:

```
SELECT  COUNT(CID)
FROM    T
WHERE   col_neoplasm = 1 AND col_brain = 1
```

Query $\overline{Q}_a(Q_s)$ is a single-block conjunctive SQL query, where the `SELECT` clause contains SQL aggregate functions, e.g., SUM and COUNT. For such a query form, we can leverage existing works on answering queries using views in the relational world. In the following, we consider two forms of views to improve query evaluation.

6.2 Aggregation Views

An aggregation view is defined by a SQL query with aggregation functions. In particular, we consider aggregation views defined by single-block queries. Let $K = \{col_{f_1}, \ldots, col_{f_k}\}$ be a subset of columns in T. V_K^a is a materialized view that groups by K and aggregates the documents' parameters of every group:

V_K^a:

```
SELECT    col_f1,...,col_fk,
          AggrFunc(para(d)) AS ContxPara
FROM      T
GROUP BY  col_f1,...,col_fk
```

6.3 Usability of Aggregation Views

A view V is *usable* for query Q if V can be used to compute complete or partial results of Q. In determining whether V_K^a is usable for $\overline{Q}_a(Q_s)$, we need to consider the mapping from V_K^a to $\overline{Q}_a(Q_s)$, as studied in the following.

We first define some notation. For notation convenience, we assume that each column in the pivot table T has a unique name. For a SQL query Q, we use Q.GroupCols to denote a set of columns in the `GROUPBY` clause, Q.WHERECols to denote a set of columns in the `WHERE` clause, and Q.AggrCol

to denote the aggregate column. For the aggregation view V_K^a, V_K^a.AggrCol is $K = col_{f_1}, \ldots, col_{f_k}$. Here we assume each query only contains at most one aggregation function. Queries having more than one aggregations can be easily decomposed into multiple single-aggregation queries.

Definition 6.3.1 (Column Mapping)**.** *A 1:1 column mapping ϕ is a mapping between column set Col_A and column set Col_B such that (1) for every column $col_i \in Col_A$, there exists a column $col_j \in Col_B$, $col_i = \phi(col_j)$, and (2) for every column $col_j \in Col_B$, there exists a column $col_i \in Col_A$, $col_j = \phi^{-1}(col_i)$.*

Theorem 6.3.1. *Given a collection-specific statistic S_c specified by the structured query Q_s and its aggregation query $\overline{Q}_a(Q_s)$, view V_K^a can compute S_c such that*

1. *there exist an 1:1 column mapping between $\overline{Q}_a(Q_s)$.WHERECols and a subset of V_K^a.GroupCols.*
2. *For the aggregation function* `AggrFunc`
   (a) *if* `AggrFuc` *is SUM, MAX or MIN, V_K^a has the same aggregation function, and there exists an 1:1 column mapping between $\overline{Q}_a(Q_s)$.AggrCol and V_K^a.AggrCol.*
   (b) *if* `AggrFuc` *is COUNT, V_K^a has a COUNT aggregation over any column.*

An intuitive understanding of the view usability is as follows: the `GROUP BY` clause in the view definition essentially partitions the document collection. Every tuple in the view is an aggregation on one partition—a statistic of documents in one partition. The evaluation of the rewritten query aggregates pre-computed results of those partitions that satisfy the query condition and avoids scanning the raw table.

Formally, for a view to be usable for a query, it must not project out any column needed in the query (and is not otherwise recoverable). When a view performs a group-by on one or more columns, we lose some information about all the other columns in the raw table. Thus, the query must require the same or a coarser grouping than performed in the view. Otherwise, we cannot drill

down to details from the view and the view is unusable. The first condition in the theorem ensures that the view's groups are finer enough to evaluate all the predicates in the original query $\overline{Q}_a(Q_s)$. Furthermore, the aggregated column must be either available or can be reconstructed from other columns, which is guaranteed by the second condition.

When the view is usable, the query that computes the statistic S_c can be rewritten as follows:

```
SELECT  AggrFuc2(ContxPara)
FROM    V_K^a
WHERE   φ(col_f1) = v1 AND ...φ(col_fm) = vm AND
        φ(col_fm+1:vm+1) = 1 AND ...φ(col_fn:vn) = 1
```

in which $\texttt{AggrFuc}_2$ is SUM if AggrFuc in $\overline{Q}_a(Q_s)$ is COUNT; otherwise, $\texttt{AggrFuc}_2$ is the same AggrFuc.

A special case of the view usability is that $\overline{Q}_a(Q_s)$.WHERECols is exactly the same as V_K^a.GroupCols. In this case, the view V_K^a directly contains the tuple that contains the collection-specific statistics of the context specified by Q_s. Hence, no additional computation is needed, and query evaluation simply need to retrieve the required statistics.

Example 6.3.1. *Consider the structured query Q_s with two predicates on the Category field: Category.atom = 'neoplasm' $\wedge$ Category.atom = 'brain'. Two statistics, collection length $\mathrm{len}(Q_s(\mathcal{D}))$ and collection cardinality $|Q_s(\mathcal{D})|$, can be translated to a SUM and a COUNT aggregation query. Let $\mathrm{col}_{neoplasm}$ and col_{brain} be two columns in T to which the two values of the Category field are pivoted. The transformed aggregation query $\overline{Q}_a(Q_s)$ that computes the statistics is shown as follows:*

```
SELECT  COUNT(CID), SUM(len(d(c)))
FROM    T
WHERE   col_neoplasm = 1 AND col_brain = 1
```

Consider a column set $K = \{\mathrm{col}_{neoplasm}, \mathrm{col}_{brain}, \mathrm{col}_{diagnosis}\}$, three pivot columns of the Category field. The view

V_K^a: *SELECT* $col_{neoplasm}, col_{brain}, col_{diagnosis}$,
$SUM\big(len(d(c))\big)$ *AS* ***ContxtLen***,
$COUNT(*)$ *AS* ***ContxCount***
FROM T
GROUP BY $col_{neoplasm}, col_{brain}, col_{diagnosis}$

partitions $\mathcal{D}$ *into* 2^3 *partitions. Tuple*

$$V(col_{neoplasm} = 0, col_{brain} = 1, col_{diagnosis} = 1)$$

aggregates the parameters of the documents that are annotated with ***brain*** *and* ***diagnosis*** *but not* ***neoplasm****. Similarly, tuple*

$$V(col_{neoplasm} = 0, col_{brain} = 0, col_{diagnosis} = 0)$$

aggregates the parameters of the documents that are not annotated with any of ***neoplasm****,* ***brain*** *or* ***diagnosis****.*

Given view V_K^a, ***collection length*** *and* ***collection cardinality*** *for the structured query can be computed as follows:*

$$\begin{aligned} len\big(Q_s(\mathcal{D})\big) &= V_K(col_{neoplasm} = 1, col_{brain} = 1, col_{diagnosis} = 1).\textit{ContxLen} \\ &\quad + V_K(col_{neoplasm} = 1, col_{brain} = 1, col_{diagnosis} = 0).\textit{ContxLen} \\ |Q_s(\mathcal{D})| &= V_K(col_{neoplasm} = 1, col_{brain} = 1, col_{diagnosis} = 1).\textit{ContxCount} \\ &\quad + V_K(col_{neoplasm} = 1, col_{brain} = 1, col_{diagnosis} = 0).\textit{ContxCount} \end{aligned}$$

6.4 Complexity of Rewritten Queries

The complexity of an aggregation query is measured in terms of the number of tuples/documents. If a materialized view is usable, collection-specific statistics can be computed by aggregating the materialized view, whose complexity is only determined by the view size, regardless of the context size. In other words, by choosing appropriate view sizes, query performance of context-sensitive ranking can be guaranteed.

Theorem 6.4.1. *If view* V_K^a *is usable for* $S_c(Q_s(\mathcal{D}))$ *in the context* $Q_s(\mathcal{D})$*, the complexity of computing* $S_c(Q_s(\mathcal{D}))$ *is* $O(ViewSize(V_K))$*, which is bounded by* $O(\Pi_{col_i \in K} |\mathbf{dom}(col_i)|)$*, where* $|\mathbf{dom}(col_i)|$ *is the domain cardinality of column* col_i*. In particular,* $|\mathbf{dom}(col_i)| = 2$ *if the column is pivoted from a value of a multi-valued field.*

When no additional indexes are built on the view, computing collection-specific statistics using a view requires a full scan of the view. Theoretically, the number of tuples in the view is exponential to the number of `GROUPBY` columns in the view. However, the actual number of non-empty tuples can be much smaller. Consider two Category columns col_{m_1}, col_{m_2} and the two MeSH terms m_1, m_2 always appear together in the same documents. Then tuples $V_K^a(col_{m_1} = 1, col_{m_2} = 0)$ and $V_K^a(col_{m_1} = 0, col_{m_2} = 1)$ are always empty. Similarly, if m_1 and m_2 never appear in the same document, tuple $V_K^a(col_{m_1} = 1, col_{m_2})$ is always empty.

Given a materialized view V_k^a, a naive approach to compute view size is to scan the entire document collection and evaluate the view expression. While accurate computation can be expensive, a simple alternative is to estimate the view size by sampling: a small number of documents are sampled and mapped to V_K. The number of non-empty tuples after the mapping is estimated as the view size. In the following, we use $ViewSize(\cdot)$ to denote a function that returns the size of a given view, either by sampling or by scanning.

6.5 Non-Aggregation Views

A materialized SQL query without any aggregation functions is called a non-aggregation view. There is a large body of work in the database literature studying answering aggregation queries using non-aggregation views [90, 50, 104]. Our goal, however, is not to study a generic rewriting algorithm to answer queries using views. Hence, we concentrate on a special form of non-aggregation views.

Let $K = \{col_1, \ldots, col_k\}$. Consider the view in the following form:

```
SELECT    CID, parameter, col_1,...,col_k
FROM      T
ORDER BY  col_1, ..., col_k, CID
```

If there is an one-to-one mapping ϕ between predicates in $\overline{Q}_a(Q_s)$ and a subset of K, the view can be used to *fully* answer the query. The rewritten query is:

```
SELECT   AggrFuc(parameter)
FROM     V_K
WHERE    φ(col_f1) = v_1 AND ...φ(col_fm) = v_m AND
         φ(col_fm+1:vm+1) = 1 AND ...φ(col_fn:vn) = 1
```

If there only exists an 1:1 column mapping ψ between a subset of predicates in $\overline{Q}_a(Q_s)$ and a subset of K, then the view cannot answer the query completely. Intuitively, since some of the predicates in the query cannot be mapped to columns in V_K, V_K does not contain all information to answer the query. However, V_K can yield a partial rewriting. Together with the base table T, we can still answer the query using the views.

```
SELECT   AggrFuc(parameter)
FROM     V_K JOIN T on T.cid = V_K.cid
WHERE    V_K.φ(col_1) = val_1 ...AND V_K.φ(col_n) = val_n
         AND T.col_n+1 = val_n+1 ...AND T.col_m = val_m
```

Compared with the usability of aggregation views studied in Section 6.3, a non-aggregation view may still be *partially* usable even if there exists no 1:1 column mapping between columns in the non-aggregation view and the query. This is because the view populates the `CID` column and does not lose multiplicity of citations. By joining with the pivot table, all information required by the query can be reconstructed. The aggregation view, however, loses multiplicity of citations after aggregating parameter columns. Hence, it cannot be joined with the base table to reconstruct lost information.

6.6 Selection of Aggregation Views

In this section, we discuss aggregation view selections. Similar to views as statistics caching, for every structured query specifying a large context ($> T_C \cdot |\mathcal{D}|$), there must be an aggregation view that is able to answer it. A structured query is a conjunction predicates. By view usability theorem in the prior section, these predicate columns must be covered by a view's `GROUPBY` clause. We re-state the view selection formalization in the context of aggregation views as follows:

Problem Statement 6.6.1. *Given a threshold of context size* T_C *and a threshold of view size* T_V*, find a set of views* $\mathcal{V} = \{V_{K_1}, V_{K_2}, \ldots\}$ *such that*

1. $\forall V_{K_i} \in \mathcal{V}$, $ViewSize(V_{K_i}) \leqslant T_V$.
2. *For every possible context specification* P *(*P *is a column set in the structured query), if* $ContextSize(P) \geqslant T_C$*, then* $\exists V_{K_j} \in \mathcal{V}$ *such that* $P \subseteq K_j$ *(*K_j *is the column set of the view's* `GROUPBY` *clause).*

6.6.1 Greedy Selection

Given the problem statement, a straightforward approach is to first find these column combinations specifying large contexts, and then choose a set of views to cover them. Similar to views as statistics caching, finding column combinations that specify large contexts is equivalent to mining association rules of pivot columns such that their support, in terms of the number of documents in which they co-occur, are greater than $T_C \cdot |\mathcal{D}|$. Hence, existing association rule mining algorithms can return a set of combinations that must be covered by aggregation views.

Given a set of high-support column combinations, while each combination can be materialized as one tuple, storing statistics of the corresponding context, we may also consolidate several combinations and their materialized tuples into one view. The effect is equivalent to horizontal partitioning of a very large table, which improves online access time. There could be a very large number of combinations and materialized tuples. While a single retrieval operation may be

efficient, matching a materialized tuple against the given structured query at run time would be still expensive. Therefore, we can consolidate combinations into smaller number of views so that the matching phase will be more efficient. We reformulate the problem as follows:

Problem Statement 6.6.2. *Given a set of high-support column combinations* $\mathcal{P} = \{P_1, P_2, \ldots\}$*, find the minimal number of views* $\mathcal{V} = \{V_{K_1}, V_{K_2}, \ldots\}$ *such that*

1. $\forall V_{K_i} \in \mathcal{V}$, $ViewSize(V_{K_i}) \leqslant T_V$.
2. $\forall P \in \mathcal{P}$, $\exists V_{K_j} \in \mathcal{V}$ *such that* $P \subseteq K_j$ *(*K_j *is the column set of the view's* `GROUPBY` *clause).*

Theorem 6.6.1. *Given a set of high-support column combinations, the view selection problem is NP-hard.*

Algorithm 3 presents a greedy algorithm that takes as an input a set of column combinations generated by association rule mining algorithms, and returns a set of views to materialize. Two heuristics are used for algorithm design. First, for two column combinations P_1, P_2, if $P_1 \subset P_2$, a view covering P_2 is usable for P_1. In other words, we only need to consider P_2 for the view selection purpose. Second, in order to reduce the total number of views, the overlap of the column combinations that are covered by a view is expected to be maximized.

The algorithm first removes column combinations that are subsets of other combinations (Line 1 in Algorithm 3), according to the first heuristic. For each newly created view V_{K_i}, the algorithm iteratively scans uncovered column combinations and adds the one that has the maximal overlap with K_i (Line 6-9 in Algorithm 3), until the size of V_{K_i} reaches T_V.

An implicit assumption of Algorithm 3 is that for any input column combination P, $ViewSize(V_P) < T_V$. This assumption can be guaranteed by setting an upper bound on the number of keywords when applying association rule mining algorithms. The upper bound on $|P|$ is reasonable in practice. Statistics from standard text search systems have shown that most user queries have no

Algorithm 3: Greedy View Selection

input : A set of column combinations $\mathcal{P} = \{P_1, P_2, \ldots\}$ generated by association rule mining algorithms

output: A set of views $\mathcal{V} = \{V_{K_1}, V_{K_2}, \ldots\}$

1 Scan $\mathcal{P}$ and remove P_i such that $\exists P_j \in \mathcal{P}, P_i \subset P_j$;
2 $i \leftarrow 0$;
3 **while** $\mathcal{P}$ *is not empty* **do**
4 Create a new view $V_{K_i}, K_i = \emptyset$;
5 Remove P_j with the largest size from $\mathcal{P}$, and add it to V_{K_i}, i.e., $K_i = P_j$;
6 **while** $ViewSize(V_{K_i}) \; ; \; T_V$ **do**
7 Remove P_m from $\mathcal{P}$ such that (1) $|K_i \cap P_m|$ is maximized, and (2) $ViewSize(V_{K_i \cup P_m}) < T_V$;
8 $K_i \leftarrow K_i \cup P_m$;
9 **end**
10 $\mathcal{V} = \mathcal{V} \cup \{V_{K_i}\}$;
11 $i \leftarrow i + 1$;
12 **end**
13 **return** $\mathcal{K}$

more than 5 keywords [12]. The number of keywords in context specifications is expected to be even smaller.

6.6.2 Graph-Decomposition-based Selection

Many existing algorithms for mining association rules achieve good efficiency. But mining association rules is still an expensive operation, as discussed in Section 5.3. In particular, to discover a combination of size k, $P_k = \{m_1, m_2, \ldots, m_k\}$, $k-1$ combinations must be visited, i.e., $P_1 = \{m_1\}, P_2 = \{m_1, m_2\}, \ldots, P_{k-1} = \{m_1, m_2, \ldots, m_{k-1}\}$, and their supports must be computed accurately, even though we are only interested in P_k to select aggregation views.

The selection algorithm in Section 6.6.1 essentially presents a bottom-up approach to select views: column combinations whose supports are greater than $T_C \cdot \mathcal{D}$ are generated first. Then a set of views are selected to cover all of them.

In this section, we present a top-down approach to select views. The idea is to decompose the column set to smaller subsets, until each column subset is small enough to be covered by one view whose size is less than T_V. The key of this approach is that the decomposition process does not violate the principle of view selection: column combinations with high supports should be covered by at least one view. Under this principle, the algorithm skips many combinations and only computes accurate supports when necessary.

Graph decomposition Schemes

Definition 6.6.1. *A* Column Association Graph (KAG) *is a graph of columns, where vertex* m_i *represents a column, and the weight of the edge* $e_{m_i - m_j}$ *represents the number of documents* m_i *and* m_j *co-occur. Edges with zero weight do not appear in the graph.*

A KAG constructs pair-wise relationships between columns, and implicitly captures k-ary $(k \geqslant 3)$ columns relationships: $m_1, m_2, \ldots, m_k$ co-occur in the same document only if $m_1, m_2, \ldots, m_k$ form a clique in the KAG. Initially, edges whose weights are less than $T_C \cdot \mathcal{D}$ can be removed from the graph, because

cliques containing such edges do not have high supports and therefore are not considered for view selection.

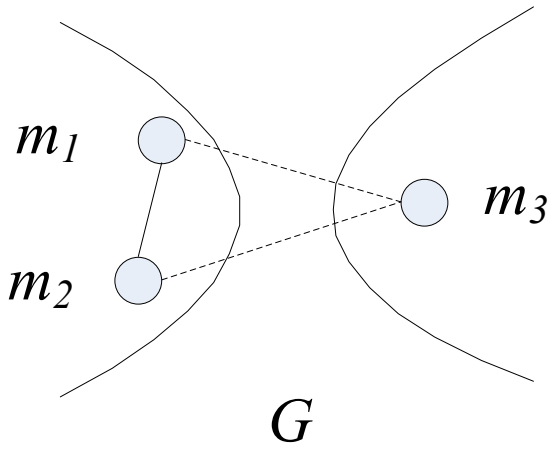
A *connected component* is a subgraph of KAG in which any two vertexes are connected to each other. As the first step, the KAG is decomposed to a set of connected components. We only need to consider views covering individual components. Without loss of generality, we assume the KAG is fully connected, and has only one connected component.

For a view that covers a subgraph, the view size is determined by the number of vertexes in the subgraph. Initially, the KAG has one component, which contains all vertexes. It is too large to be covered by one view. We need to decompose the KAG into subgraphs so that views covering individual subgraphs are smaller than T_V.

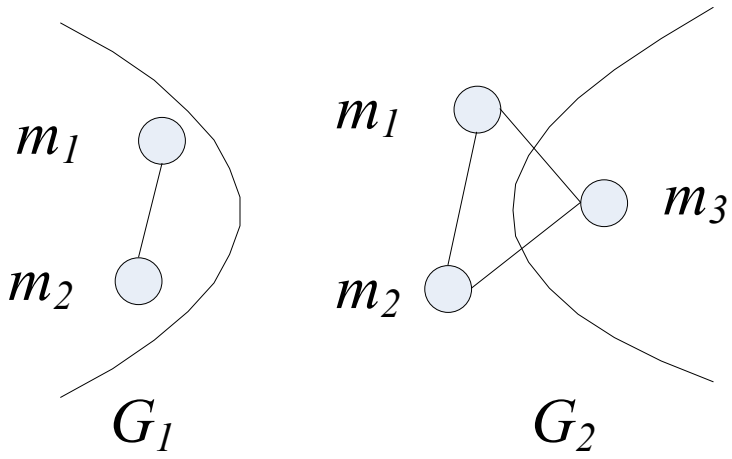
A cut divides the KAG $G = (V, E)$ into two parts, as shown in Figure 6.1(a). Since the graph is fully connected, some edges' endpoints are in different parts. In Figure 6.1(a), m_1, m_2, m_3 form a clique and some of its edges cross the two parts. The goal of the decomposition is to completely separate the graph. The question is: how to deal with the crossing edges?

The principle of the decomposition is that if the support of a clique (i.e., a columns combination) is greater than $T_C \cdot \mathcal{D}$, the clique must be kept holistically in one subgraph after the decomposition, so that at least one view will cover it. In Figure 6.1(a), if the support of $\{m_1, m_2, m_3\}$ is greater than $T_C \cdot \mathcal{D}$, after the decomposition, at least one subgraph needs to contain the clique. To this end, m_1, m_2 and the edge between them are replicated in G_2 after the decomposition, as shown in Figure 6.1(b). Notice that m_1, m_2 and the edge $e_{m_1-m_2}$ are kept in G_1 as well. The reason is that other vertexes in G_1 may form cliques with them. Removing m_1, m_2 and the edge $e_{m_1-m_2}$ from G_1 may lose column combinations that should be covered by views.

If the support of $\{m_1, m_2, m_3\}$ is less than $T_C \cdot \mathcal{D}$, the corresponding clique is decomposable, because we do not need any view to cover it. This is the second decomposition scheme, as shown in Figures 6.2. Compared with the first decomposition scheme, the edge $e_{m_1-m_2}$ is not replicated in G_2. Hence, G_2 in



(a) The original graph



(b) Subgraphs after decomposition

Figure 6.1: The first graph decomposition scheme

Figure 6.2(b) is sparser than G_2 in Figure 6.1(b).

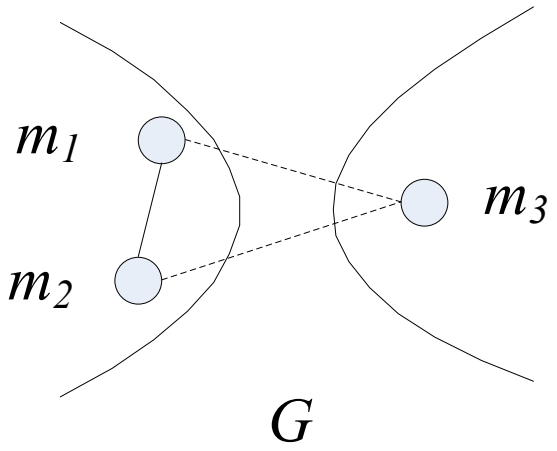
A formal representation of the decomposition schemes is described as follows.

Definition 6.6.2. *A* vertex separator *is a set of vertexes whose removal separates a graph into two distinct connected components.*

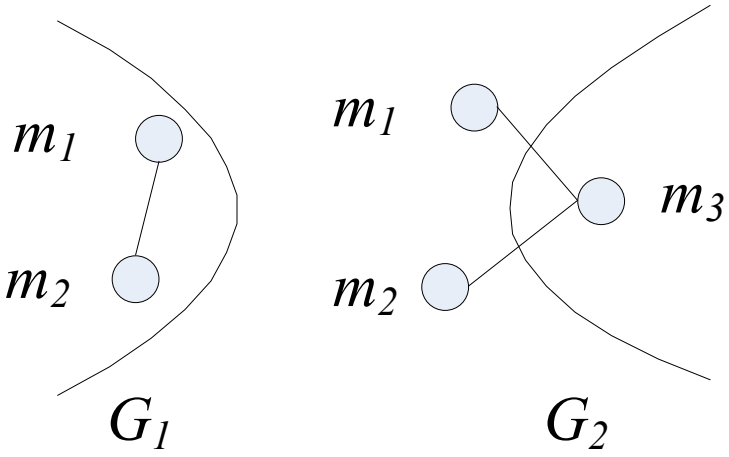
Let S_0 be a vertex separator whose removal separates the vertexes in the KAG $G = (V, E)$ into S_1 and S_2, i.e., $V = S_1 \cup S_2 \cup S_0$. Given S_0, $G = (V, E)$ is decomposed into $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ such that:

- $V_1 = S_1 \cup S_0$, $V_2 = S_2 \cup S_0$.
- $\forall m_i \in S_1, m_j \in S_1$, if $e_{m_i - m_j} \in E$, $e_{m_i - m_j} \in E_1$.
- $\forall m_i \in S_2, m_j \in S_2$, if $e_{m_i - m_j} \in E$, $e_{m_i - m_j} \in E_2$.
- $\forall m_0 \in S_0$, if $\exists m_i \in S_1, e_{m_0 - m_i} \in E$, then $e_{m_0 - m_i} \in E_1$; if $\exists m_j \in S_2, e_{m_0 - m_j} \in E$, then $e_{m_0 - m_j} \in E_2$.
- $\forall m_i \in S_0, m_j \in S_0$, if $e_{m_i - m_j} \in E$, $e_{m_i - m_j} \in E_1$.
- $\forall m_i \in S_0, m_j \in S_0$, if (1) there exists a clique containing m_i, m_j and vertex(es) in S_2, and (2) the support of the clique is greater than $T_C \cdot \mathcal{D}$, $e_{m_i - m_j}$ is replicated in E_2.

In the example in Figure 6.1 and 6.2, $S_0 = \{m_1, m_2\}$. Theoretically, whether to replicate the edge $e_{m_1 - m_2}$ in G_2 or not depends on whether the support of the clique containing $e_{m_1 - m_2}$ is greater than $T_C \cdot \mathcal{D}$. Since the support of the clique cannot be derived from the graph, we still need to compute support, which is similar to mining association rules. However, recall that as long as the view selection principle is satisfied, either decomposition scheme is correct. If the support of the clique is unknown, we may implicitly assume that the support is greater than $T_C \cdot \mathcal{D}$, and all the edges in the clique are replicated in G_2. In other words, using the first decomposition scheme always leads to a correct decomposition.



(a) The original graph



(b) Subgraphs after decomposition

Figure 6.2: The second graph decomposition scheme

The above analysis indicates that computing support is not always necessary for the view selection purpose, especially when the subgraphs are large and sparse. The first decomposition scheme becomes less effective when the graphs are smaller and denser, and eventually is invalid for the subgraphs that are cliques.

Graph Decomposition Algorithm

Having the decomposition schemes, the remaining question is how to choose the vertex separator S_0 so that the graph can be decomposed efficiently. Two factors are considered: first, S_1 and S_2 should be about the same size, so that the sizes of all subgraphs decreases fast as the decomposition proceeds. Second, the number of vertexes in S_0 should be minimized. Since vertexes in S_0 are replicated in G_1 and G_2, and the view size is directly related to the number of vertexes in a subgraph, we want to minimize the number of replicated vertexes.

The optimization function for the graph decomposition is defined as follows:

$$\min \frac{|S_0|}{\min\{|S_1|, |S_2|\} + |S_0|} \tag{6.1}$$

The numerator minimizes the number of vertexes to be replicated. The denominator ensures that neither of the subgraphs is too small.

Given the optimization function in Formula 6.1, the graph decomposition problem is NP-hard [25]. A number of approximation algorithms have been developed. Most recently, paper [43] exhibits an $O(\sqrt{\log n})$ approximation algorithm for finding balanced vertex separators in general graph, with approximation ratio of $O(\sqrt{\log \mathsf{opt}})$ where opt is the size of an optimal separator.

The pseudo code of the algorithm that decomposes the KAG is shown in Algorithm 4.

Algorithm 4: Graph decomposition

input : A KAG $G = (V, E)$

output: A vertex separator (S_1, S_2, S_0)

Let $V = \{v_1, v_2, \ldots, v_n\}$;

foreach $1 \leqslant i \leqslant n$ **do**

  Create the augmented graph by adding a source s and a sink t to G;

  Connect s to $v_j, 1 \leqslant j \leqslant i$, and connect t to v_k, $i < k \leqslant n$;

  Find the minimum capacity $s - t$ separator S_0^i ;

  Let $S_1^i = (V \cup \{s, t\}) - S_0^i$, $S_2^i = V - (S_1^i \cup S_0^i)$;

end

return (S_1^i, S_2^i, S_0^i) such that $\frac{|S_0^i|}{|E_{12}^i|}$ is minimal, where $|E_{12}^i|$ is the number of edges e_{u-v}, $u \in S_1^i \cup S_0^i$, $v \in S_2^i \cup S_0^i$;

6.6.3 Hybrid Approach

The greedy selection and the decomposition-based selection have strengths in different directions. The greedy approach is strict, and only covers column combinations that must be covered. Therefore, it is space efficient. However, it has to enumerate a very large number of column combinations. The decomposition-based selection, on the other hand, usually covers more column combinations than required. While it has high efficiency when the graph is large and sparse, its capability is limited when the graph is small and dense.

In implementation, we uses a hybrid approach to select aggregation views. Initially, the graph decomposition algorithm quickly decomposes the KAG into subgraphs, most of which can be covered by individual views. The greedy approach is used thereafter to further decompose the remaining sub-graphs, each of which is a clique and is still too large to be covered by one view.

6.7 Related Work

The reduction from collection-specific statistics to aggregation queries is the key insight in our solution to the efficiency issue of context-sensitive ranking. To our best knowledge, we are the first to envision this formalization. Once we have the reduction, using materialized views to improve efficiency is straightforward. And years of research on answering queries using views in the database community can be leveraged.

Answering queries using views has a long history, due to its relevance to a variety of data management problems, such as query optimization and data integration. Paper [53] presents a survey of most of these efforts. We pay a special attention on queries with aggregation functions, because collection-specific statistics always demand aggregations. Aggregation queries and views raise several additional subtleties on using views. The first difficulty arises in dealing with aggregated columns. For a view to be usable by a query, it must not project out an attribute that is needed in the query. When a view performs an aggregation on an attribute, we lose some information about the attribute, and in a sense partially projecting it out. If the query requires the same or a coarser grouping than performed in the view, and the aggregated column is either available or can be reconstructed from other attributes, then the view is still usable for the query. The second difficulty arises due to the loss of multiplicity of values on attributes on which grouping is performed. When we group on an attribute, we lose the multiplicity of the attribute, thereby losing the ability to perform subsequent sum, counting or averaging operations. Only in some cases, it may be possible to recover the multiplicity using additional information.

Paper [50] considers syntactic matching between views and queries by performing a set of transformations. The goal is to identify a sub-expression of the query that is identical to the view, and hence can be substituted by the view. The limitation of this approach is that it only considers at the syntax level and may not be applied to many scenarios.

Paper [90] presents a semantic approach for aggregation queries. It describes the conditions required for a view to be usable for answering an aggre-

gation query, and a rewriting algorithm that uses these conditions. A result that is not captured in [50] is that an aggregation view may be used to answer a query, only if the query removes duplicates in the `SELECT` clause. The work described in [104] extends the treatment of grouping and aggregation to consider multi-block queries and to multi-dimensional aggregation functions such as cube, roll-up, and grouping sets [46].

Papers [79, 35, 47, 48, 34] present formal aspects of answering queries using views in the presence of grouping and aggregation. They present cases in which it can be shown that a rewriting algorithm is complete, in the sense that it will find a rewriting if one exists. Their algorithms are based on insights into the problem of query containment for queries with grouping and aggregation.

In our problem setting, our goal is not to study query rewriting for generic queries and views with aggregations. Instead, aggregation queries for computing collection-specific statistics are in a fixed form: they are single-block conjunctive aggregation queries over a single table. Also, what views to materialize to improve performance is under our control. Hence, we choose a simple view form such that usability and query rewriting is mainly based on syntactic matching, which is a special case studied in [90].

In addition to answering queries using views, a lot of work has been done in OLAP query processing, e.g., [46, 55, 58, 28, 95]. Data cubes are as a special form of materialized views. Compared with works in answering queries using views, OLAP only deals with a fixed pattern queries, i.e., single-block queries over a star schema—a fact table referencing one or more attribute tables. Aggregations are always over the fact table. This query pattern greatly simplifies the rewriting problem. Works in OLAP in return mainly concerns about physical representations of cubes and how to maintain them efficiently.

View selection is an important problem in conventional relational database systems (RDBMSs) and there has been extensive works on it. The problem setting is as follows: given a database consisting of relations $\mathcal{R} = \{R_1, \ldots, R_r\}$ and a query workload $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ over $\mathcal{R}$, select a set of views $\mathcal{V}$ such that the query workload is answered with the lowest cost under limited amount of resources,

e.g., disk space or view maintenance cost.

Paper [55] presents a greedy algorithm for view selection for OLAP queries. The algorithm is in polynomial time, but does not consider view maintenance cost. Subsequent works also contribute to view selection in OLAP environment [16, 87].

Paper [103] proposes a greedy view selection algorithm for SPJ (SELECT, PROJECT AND JOIN) aggregation queries with minimized cost of query processing and view maintenance. However, this works does not consider resource constraint. A theoretical study on view selection in data warehousing is presented in [51], in which authors present a near-optimal exponential greedy algorithm for AND-OR view graph and a near-optimal polynomial algorithm for AND view graph and OR view graph. The work is extended in [52].

Physical database design is a large category of works in which view selection is treated as a sub-problem. For example, paper [9] presents a syntactical analysis of the workload to address the problem of selecting both views and indexes to be materialized. This approach proceeds in three main steps. The first step analyses the workload and chooses subsets of base relations with a high impact on the query processing cost. Based on the base relations subsets, the second step identify syntactically relevant views and indexes that can potentially be materialized. In the third step, the system runs a greedy enumeration algorithm to pick a set of views and indexes to materialize based on the result of the second step by taking into account the space constraint. However, this approach does not take into account the view maintenance cost.

The problem setting of view selection in this dissertation is essentially different from that of conventional databases. Our goal of view selection is to improve the performance of worst-case queries. This is based on the following considerations. First, no query workload is available for this new query model. Second, even if the query workload is available, it is still dangerous to only rely on it. Unlike RDBMS queries which are fairly stable, queries of keyword search systems are typically unpredictable and may evolve as time passes.

6.8 Acknowledgments

Chapter 6 was published in ACM SIGMOD International Conference on Management of Data, 2011, entitled “Context-sensitive Ranking for Document Retrieval”. This is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the paper.

Chapter 7

A Comparative Study of Two Forms of Views

We present two forms of views in prior chapters. While views as statistics caching aggressively materialize statistics of target queries and optimize online efficiency, aggregation views strike a middle ground between offline computations (i.e., view selections and materialization) and online computations (i.e., using intermediate results to compute statistics). Since both approaches aim to compute statistics and save online computations, a natural question to ask is which approach is better or are they completely equivalent?

In this chapter, we conduct a comparative study to compare the performance trade-offs of the two approaches. Though these two approaches have different outputs—fully materialized statistics vs. intermediate results, we set the measure as the execution time of association rule mining (ARM). For views as statistics caching, the results of the selection algorithm are the same as ARM. For aggregation views, the algorithm's outputs are intermediate results. We further use these intermediate results to compute the ARM's results, and count the total execution time.

In the following, we first describe the factor that determines the performance trade-offs. Then we show experimental results and a guidance on under which circumstances one approach is superior than the other.

7.1 Rules

Definition 7.1.1. *Given the structured document collection $\mathcal{D}$, a* rule *is a constraint between two sets of field values $\mathcal{X} = \{x_1, \ldots, x_n\}$ and $\mathcal{Y} = \{y_1, \ldots, y_n\}$. $\mathcal{X}$ and $\mathcal{Y}$ are either conjunctive or disjunctive. $\mathcal{X}$ infers $\mathcal{Y}$, denoted by $\mathcal{X} \rightarrow \mathcal{Y}$, iff all structured documents in $\mathcal{D}$ satisfying $\mathcal{X}$ also satisfy $\mathcal{Y}$. A document d satisfies $\{x\}$ iff the structured header $d(c)$ contains the field value x.*

A rule specifies a connection between two or more field values. The connection may be a ground truth or a data-specific attribute. For example, every citation in PubMed annotated with the MeSH term "leukemia" is also annotated with "neoplasm", because leukemia is classified as a disease of neoplasm. Hence, we have

$$\text{leukemia} \rightarrow \text{neoplasm}$$

As another example, the term "diabetes" in abstract of PubMed citations is always associated with "type I" or "type II". Though in public domains diabetes are often mentioned alone, in PubMed, a professional biomedical database, diabetes types are specifically mentioned, because they have different causes and lead to different studies. Hence, we have

$$\text{diabetes} \rightarrow \text{type I} \vee \text{type II}$$

7.2 Rules' Effects on Views as Statistics Caching

Rules present constraints between field values. These constraints can be used to save computations of association rule mining (ARM). To distinguish different savings, we further define *definite rules* and *indefinite rules*.

Definition 7.2.1. *A rule $\mathcal{X} \rightarrow \mathcal{Y}$ is a* definite rule *iff $\mathcal{Y}$ is purely conjunctive. Otherwise, it is an* indefinite rule.

A definite rule can save computations by the following property:

Property 7.2.1. *If a combination C's support is n and C satisfies $\mathcal{X}$, for any subset $\mathcal{Y}_s \subseteq \mathcal{Y}$, the support of $C \cup \mathcal{Y}_s$ is also n.*

Example 7.2.1. *Consider the above definite rule leukemia $\rightarrow$ neoplasm: any citations annotated with "leukemia" are also annotated with "neoplasm". For the problem of ARM, if the support of {leukemia, digestive system} is known, the support of {leukemia, digestive system, neoplasm} is also known without any computations.*

Indefinite rules, on other hands, are not effective in saving computations for ARM. Consider diabetes $\rightarrow$ type I $\vee$ type II. Given the support of {diabetes}, the supports of {diabetes, type I} and {diabetes, type II} are still unknown. Intuitively, citations annotated with "diabetes" can be annotated with either "type I" or "type II" or both. No definite answers can be derived from this rule.

In general, when $\mathcal{Y}$ involves disjunctions, the inference $\mathcal{X} \rightarrow \mathcal{Y}$ does not give definite answers when $\mathcal{X}$ is satisfied. Hence, the support of a combination derived from an indefinite rule still needs computations for its support.

7.3 Rules' Effect on Aggregation Views

Rules save computations for aggregation views by eliminating tuples in the view. View tuples that do not satisfy the rules are always empty, i.e., no documents will be mapped to the corresponding tuple. By Theorem 6.4.1, the complexity of using views to answer queries is proportional to the view size. The more tuples are empty, the smaller the view size is, and hence the more efficient online computations are.

Example 7.3.1. *Consider an aggregation view that groups by* $\mathtt{col}_{diabetes}$, $\mathtt{col}_{typeI}$ *and* $\mathtt{col}_{typeII}$, *and aggregates statistics of each group. Theoretically, this view will generate* 2^3 *tuples. Given the above indefinite rule diabetes $\rightarrow$ type I $\vee$ type II, the tuple*

$$V(\mathtt{col}_{diabetes} = 1, \mathtt{col}_{typeI} = 0, \mathtt{col}_{typeII} = 0)$$

is always empty, because citations with "diabetes" are either annotated with "type I" or "type II" or both. "Diabetes" are never mentioned alone. Hence, no documents will be mapped to this tuple; this tuple is always empty.

An important distinction between rules for ARM and rules for aggregation views is that both definite and indefinite rules create empty tuples, reducing

the view size significantly. The above example shows an indefinite rule, which eliminates one tuple in the aggregation view.

7.4 Experimental Comparisons

We experimentally evaluate the performance effect of rules. Analytically, the more definite rules there are, the more combinations can be skipped in ARM. Their supports can be derived directly. Similarly, for aggregation views, the more rules there are, the more empty tuples in the view, and thus the more efficient online computations are.

We create a synthesized data set and vary number of definite rules to show the performance trade-offs between the two forms of views. The data set contains is 3 GB documents. And the average number of terms of a document is 20.

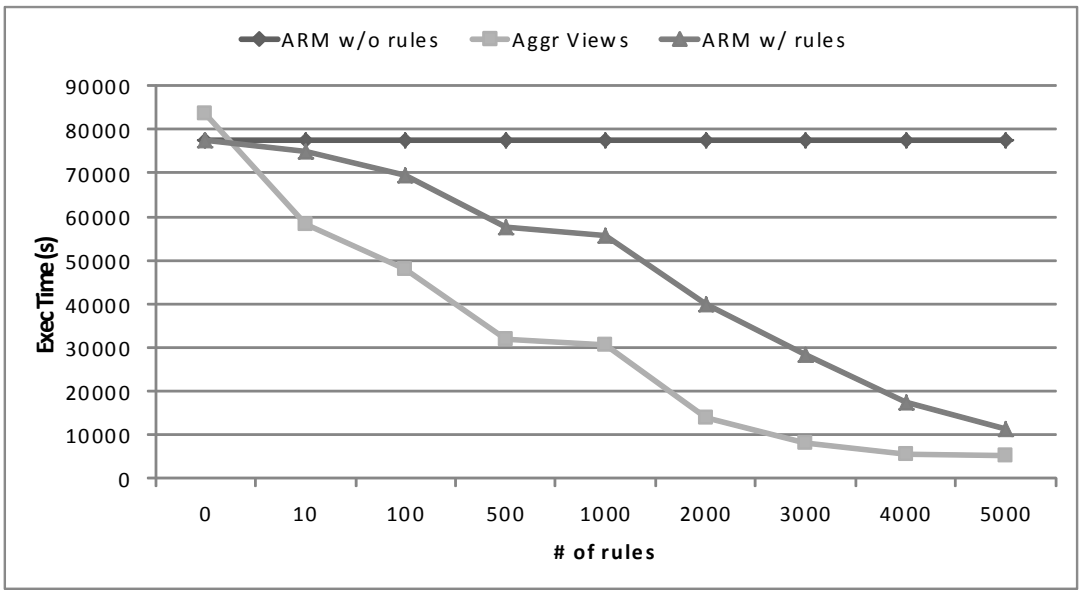


Figure 7.1: Efficiency comparison of ARM and Aggregation views

The execution time of the two approaches with different number of definite rules are shown in Figure 7.1. As we can see, when there are a large number of definite rules, using ARM algorithms to generate statistics is efficient, even

a little faster than aggregation views. As the number of rules decreases, aggregation views are more efficient. However, when there are very few rules, both approaches are not efficient in generating statistics for all queries specifying large contexts.

Another observation in Figure 7.1 is that when the number of rules is 0, using aggregation views is a little more expensive than ARM. This is due to the fact that when the number of rules is 0, every document is mapped to a separate view tuple and the aggregation view size is as large as the data size. Then using the aggregation view to find high-support combinations is equivalent to ARM. In such a case, materializing aggregation views presents no benefits, but the only materialization cost—reading and writing the data set once. Hence, the execution time of using aggregation views is slightly longer than ARM.

7.5 Acknowledgments

Chapter 7 is currently being prepared for submission for publication, which is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the paper.

Chapter 8

A Diagnostic Tool

In prior chapters, we discuss two forms of views, their usage to improve query performance and their selection algorithms. In this chapter, we present a diagnostic tool that (1) given a user-specified maximal query execution time, selects a context size threshold, and (2) chooses a view type for materialization based on the characteristics of the data.

The first functionality of the diagnostic tool transforms the performance goal to a single parameter: context size threshold. Selection algorithms for both forms of views use this parameter to select appropriate views so that queries specifying contexts larger than this threshold are answered by views. With the help of views, all queries are expected to finish within the specified time.

The second functionality of the diagnostic tool selects a view type for materialization. As demonstrated in the comparative study, two forms of views show different strengths when the number of rules varies. In this chapter, we present an automatic process that chooses an appropriate view type on the fly.

Overall, such an automatic end-to-end tool greatly improves usability and frees administrators from the pain of tuning parameters and making decisions.

8.1 Choosing a Context Size Threshold

To design a tool that selects a context size threshold, the key is to derive a cost model that connects the context size and the query execution time. In

the following, we first reiterate the theoretical cost model presented in Section 3.2 and present an experimental demonstration of the model. Then we propose a practical solution to accommodate inaccuracy of the theoretical model. We show experimentally that the revised model achieves a very high accuracy in predicting the query cost. Most queries can finish within the specified time given the context size threshold automatically selected by the tool.

8.1.1 Cost Model & Experimental Demonstration

Evaluation of a context-sensitive query involves two steps: materialize the context and aggregate collection-specific statistics. The second step is implemented by aggregating documents' parameters in the context. Since aggregation functions for statistics are arithmetic, accessing one document is $O(1)$, and hence the complexity of the second step is characterized by $O(\mathtt{contxSize})$, where $\mathtt{contxSize}$ is the context size, i.e., the number of documents in the context. The first step of query evaluation is modeled as intersections of inverted lists. Each inverted list maps a predicate value to a list of documents containing the value.

As discussed in Section 3.2, in addition to the standard merge join, many sophisticated optimization techniques have been proposed [37, 17, 38, 13, 14, 18], aiming to use the minimum number of comparisons ideally required to establish the intersection. Instead of measuring the algorithm's complexity by the input lists' sizes, these algorithms use the result size as the measure to evaluate the optimality. Hence, $O(\mathtt{contxSize})$ is a good approximation of the cost of materializing the context.

Overall, the cost of evaluating a context-sensitive query is approximated by the context size $\mathtt{contxSize}$. The query execution time is characterized by the following formula:

$$\text{exec time} = \alpha \cdot \mathtt{contxSize}$$

To validate the above formula, we perform an experimental study: we randomly sample a large number of context specifications (which are random combinations of MeSH terms in PubMed), and measure the correspondence

between the context size and query execution time. The results are shown in Figure 8.1.

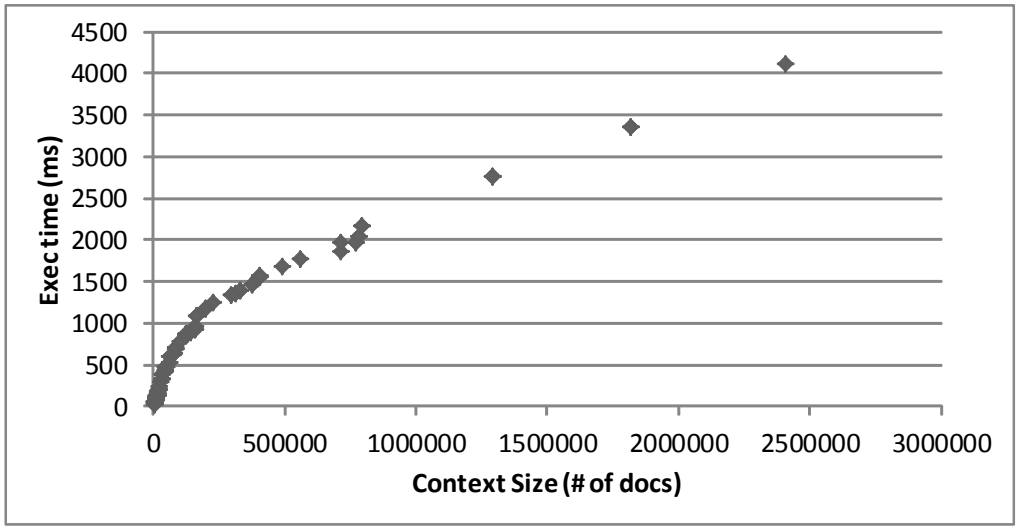


Figure 8.1: Correspondence between the context size and the query execution time

The relationship demonstrated in Figure 8.1 does not follow a linear function exactly; the cost per result—the slope of the linear function—decreases as the number of results (i.e., the context size) increases. An intuitive explanation is as follows. To evaluate a query and process a result, some basic units of data structures must be accessed. For example, to generate 1 result, at least two disk (or memory) pages must be accessed to establish an intersection of two inverted lists. One more disk page will be accessed to aggregate the document's parameters to compute statistics. However, it usually takes fewer than 100 disk pages for intersections to produce 100 results.

An important observation, however, is that even though we cannot use a single linear function to accurately characterize the full spectrum, the linear property still satisfy in smaller sub-ranges.

8.1.2 Parameter Selection

An ideal diagnostic tool for view selection inputs a maximal query execution time and outputs a context size threshold to materialize views so that all queries can finish within the specified time. This requires a cost function that characterizes the query execution time in terms of the context size. The linear cost function described above may not be accurate overall.

A practical solution to tackle the problem is to fit a linear function and derive the constant α based on query samples in a target range. As shown in Figure 8.1, a linear function is accurate enough for a specific range. Given the user-specific query time, we only need to sample a set of queries in the corresponding time range. The derived linear function can estimate the context size for the target.

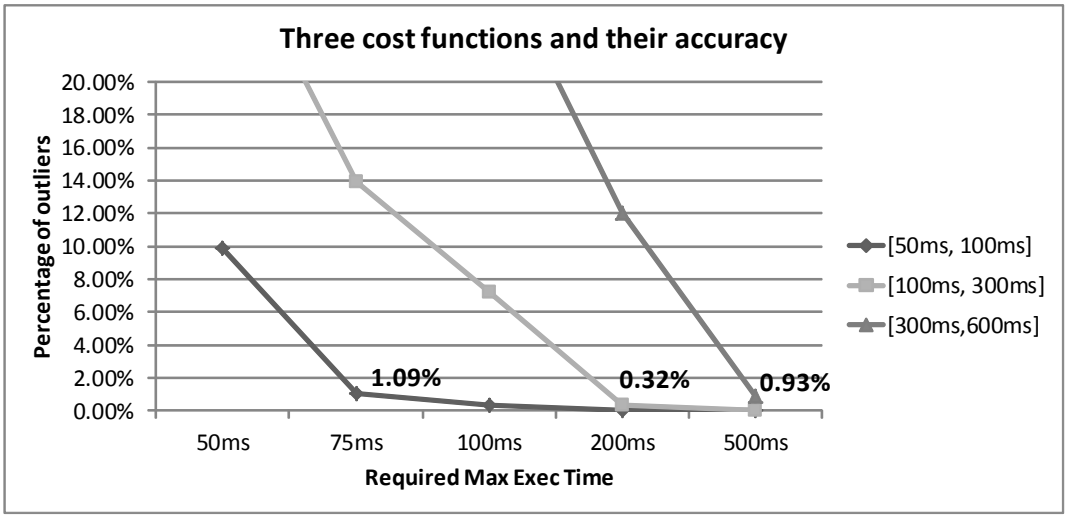


Figure 8.2: Three cost functions for the target ranges

To validate the solution, we derive three cost functions for three time ranges, namely 50-100 ms, 100-300 ms and 300 - 600 ms. Each function is evaluated by the percentage of outlier queries that cannot finish within the specified time. The results are reported in Figure 8.2. The X axis is the user-specified maximal query time, and the Y axis is the percentage of outlier queries out of 50,000 sampled queries.

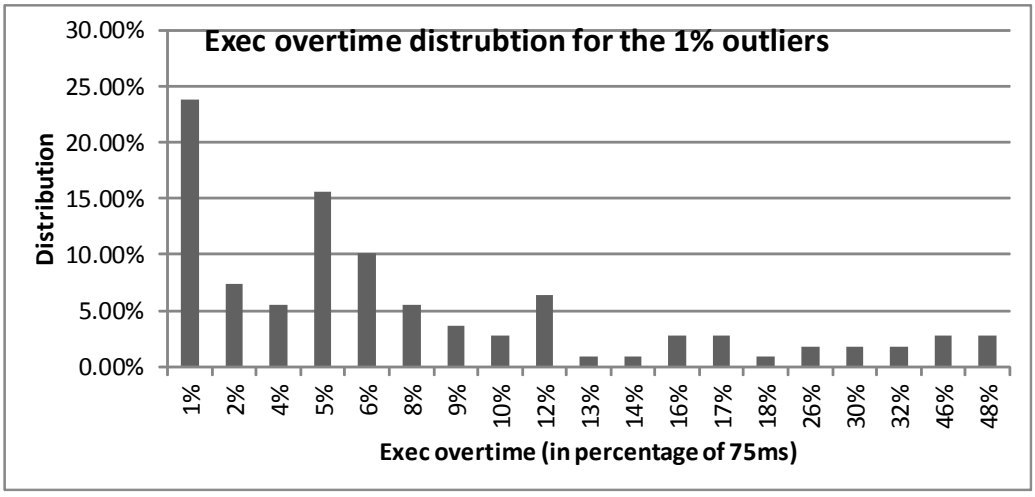
The major conclusion drawn from Figure 8.2 is that three cost functions achieve high accuracy for the target range. For instance, only 1.09% queries cannot finish within 75 ms when the function targets at 50-100 ms. The outlier queries exceeding 200 ms drop to 0.32% when the function targets at 100-300 ms. Overall, the percentage of outlier queries exceeding the target time is around 1% for all functions.

Another observation made from Figure 8.2 is that the cost function for a target time always overestimates the cost of queries far more expensive than the target. On the other hand, the cost function always underestimates the cost of queries cheaper than the target time. In Figure 8.2, when we use the function targeting at 100-300 ms to estimate queries around 75 ms, the percentage of outlier queries is much higher, i.e., 14%. This means that the function underestimates the query cost and hence does not choose a context size threshold small enough to cover all outlier queries. This observation is consistent with our prior observation: the cost per result—the slope of the linear function—decreases as the context size increases. Using a linear function with a smaller slope always underestimates the value when the function is supposed to have a larger slope.

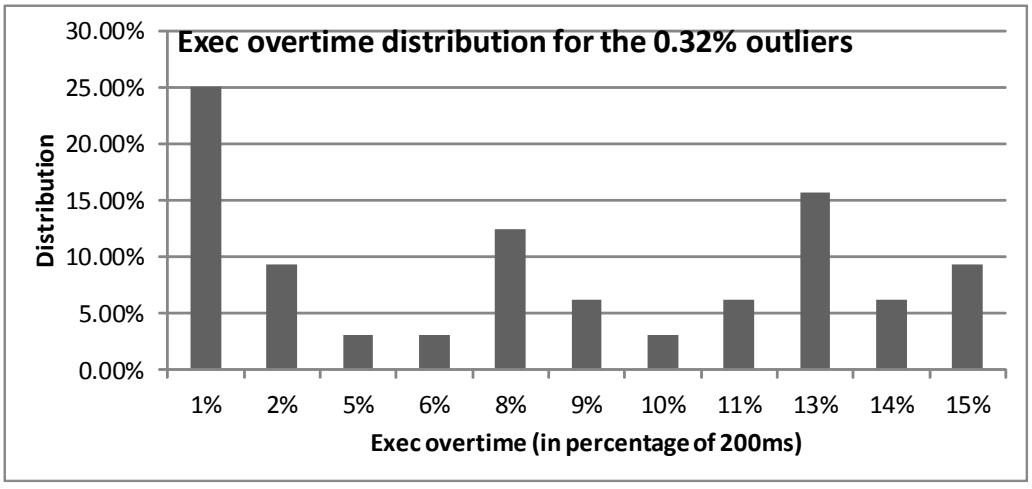
In Figure 8.3, we report the distribution of query execution time of outlier queries. The X axis is the execution overtime in percentage of the target time. Results in these two figures further demonstrate the superiority of our solution to estimate the context size threshold. In Figure 8.3(a), the cost function targets at the range of 50-100 ms and the user-specified maximal query time is 75 ms. Of all the outlier queries that cannot finish within 75 ms, more than 70% of them exceed within 10%. In other words, they can be finished within 82.5 ms. In Figure 8.3(b), the cost function targets at 100-300 ms. Of all the outlier queries that cannot finish within 200ms, more than 60% of them exceed within 10%.

8.2 Choosing a View Type

The observations in Figure 7.1 show that the number of explicit rules is the determining factor that which view form is better than the other.



(a)



(b)

Figure 8.3: The distribution of execution overtime of outlier queries

1. When there are a large number of explicit rules (e.g., ontology), ARM is good enough. As discussed in Section 5.2, the PubMed ontology specifies hierarchical relationships of MeSH terms and thus form definite rules. ARM algorithms can use these explicit rules to efficiently select queries and materialize their statistics.

2. When many rules are implicit (e.g., terms in abstract), aggregation views can be much better, because the view encoding captures all the rules implicitly. Materializing an aggregation view does not need knowledge of any rules. Rather, the mapping between documents and groups (determined by the GROUPBY clause of the view definition) automatically hides all empty tuples, which are essentially determined by the rules. For instance, the above example rule diabetes $\rightarrow$ type I $\vee$ type II are implicit in the data set. And yet aggregation views can leverage it to improve online computations.

3. When there are few rules, both views show expensive computations. In such cases, we need to use views as statistics caching and aggressively materialized statistics offline. Since each aggregation view is very large and aggregating the view online is expensive, aggregation views cannot provide efficient online computations any more and the performance goal in Section 4.3 will not be satisfied.

When no explicit rules are available, a natural question to ask is whether we can first discover rules and use them later to accelerate ARM algorithms. Discovering rules is only efficient for binary rules in the form of $A \rightarrow B$. $A \rightarrow B$ is derived iff $supp(A, B) = supp(A)$. In other words, discovery requires compute the supports of $\{A, B\}$ and $\{A\}$ respectively. Computing $supp(A, B)$ and $supp(A)$ coincides with ARM in the first two steps; ARM algorithms need to compute them anyway. Equality can be quickly checked once ARM algorithms return supports of all pairs.

Discovering rules involving more values, however, departs from ARM and may introduce higher cost. ARM algorithms prune many combinations directly by eliminating those containing any low-support subsets. To discover

rules, however, combinations with any supports must be considered. For example, $A \wedge B \rightarrow C$ iff $supp(A, B, C) = supp(A, B)$. The combination $\{A, B, C\}$ may already be pruned by ARM much earlier. Hence, discovering rules have more computations. The cost of discovering rules itself may already exceeds that of ARM algorithms.

An algorithmic procedure for choosing a view type as listed as follows:

1. If there is a large number of explicit rules, use ARM to choose queries for statistics caching.

2. Otherwise, compute all the supports of binary combinations and discover binary rules in the form of $A \rightarrow B$. If many binary rules are discovered, go to 1.

3. Otherwise, materialize aggregation views. If aggregation views are small, use these views to compute statistics at runtime.

4. Otherwise, choose ARM to aggressively materialize statistics for caching.

8.3 Acknowledgments

Chapter 8 is currently being prepared for submission for publication, which is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the paper.

Chapter 9

Experimental Results

In this chapter, we experimentally evaluate the effectiveness of context-sensitive ranking and query performance of proposed techniques. Specifically, we concentrate on three measure: ranking quality, query performance and view selection efficiency. The first metric validates the value of context-sensitive ranking. The last two metrics show the efficiency of our techniques.

9.1 Data Set & Experiment Setup

We use the PubMed data set to evaluate the effectiveness of context-sensitive ranking and the efficiency of the materialized view technique. PubMed maintains 18 million citations. Citations have both unstructured data such as title and abstract, and structured data such as publication time, citation type, and citation type. The value of using combinations of structured queries and keyword search for PubMed is validated by the fact that current PubMed search interface [4] already supports such combinations (though returned results have no relevance ranking). Moreover, there are dozens of third-party tools (such as GoPubMed [39], PubReMiner [1]) that use similar combinations to improve information retrieval and analytics.

In the experiments, we use the Lucene library [2] as the standard text search system. Lucene is a general-purpose text search system and reflects the state-of-the-art of keyword query evaluation. We only use Lucene for perfor-

mance evaluation, but not for ranking. The reason is that Lucene's ranking module provides limited interfaces for customized ranking, which is not suitable for our context-sensitive ranking model.

The algorithms and the framework are implemented under Java 6. All the experiments are performed on an Intel i7 860 PC, with 8G memory.

9.2 Ranking Quality

We evaluate the effectiveness of context-sensitive ranking using the TREC Genomics benchmark of 2007 [57], which consists of 162,048 full-text documents, a small fraction of the PubMed data set. The TREC Genomics also contains 34 topics in the form of biological questions, which were collected from bench biologists and represent actual information needs in biomedical research. For each query, relevant documents were tagged manually by biologists based on pooled results from team submissions as the gold standard.

9.2.1 Context: Combinations of MeSH Terms

In the first set of experiments, we use combinations of MeSH terms to specify contexts. A MeSH term is an atomic value in the Category field. Thus, a conjunction of MeSH terms is equivalent to conjunctive equality predicates on the Category field.

Given the TREC Genomics questions, conventional keyword queries are constructed by extracting one or more noun keywords from the questions. For example, for the question "*What symptoms are caused by human parvovirus infection*", a possible keyword query is $Q_k = \text{symptoms} \wedge \text{human} \wedge \text{parvovirus} \wedge \text{infection}$.

Then we rely on PubMed's Automatic Term Mapping (ATM) to construct appropriate contexts. Given a set of keywords, PubMed's ATM maps them to one or more MeSH terms. For the previous example, ATM maps the keywords to two MeSH terms: Humans and Parvovirus. Then $Q_s = \text{Humans} \wedge \text{Parvovirus}$ specifies the context that studies Humans and Parvovirus.

For the constructed context-sensitive queries, we exclude those queries whose result sets are too small (less than 20), or the corresponding relevant document sets in the gold standard are too small (less than 5), since ranking thereof is not so important. Altogether 30 queries qualify for the experiment. Table 9.1 lists the queries.

The main concern of ranking quality in practice is the number of relevant results in top few returned results, which are most likely to be examined by users. To study this aspect, we measure the rank precision among top ranked results, i.e., the number of relevant results in top K results. For the TREC Genomics benchmark, the relevance of a document to the query is based on whether the TREC Genomics gold standard includes the document. In the experiments, K is set to 20, as statistics from PubMed has shown that most users do not go beyond looking top 20 [4].

In additional to the precision, the reciprocal rank [96] is another popular measure for evaluating top ranked results. The reciprocal rank is the inverse of the position of the first relevant document in the ranked results. The higher the reciprocal rank of the query, the better the ranking is. In particular, if the first result is relevant, the reciprocal rank is $\frac{1}{1} = 1$.

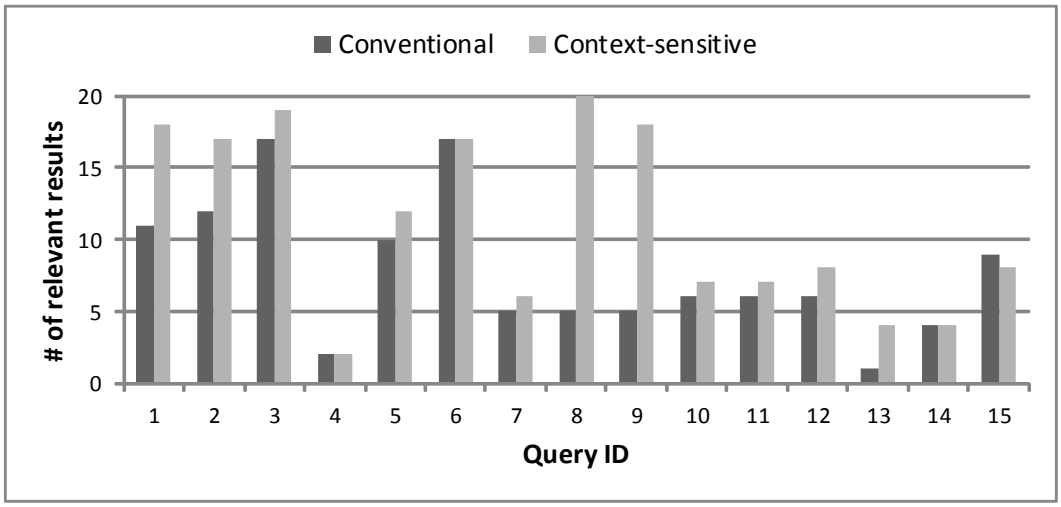
In the experiments, we use the TF-IDF model as shown in Formula 2.3.1. While more sophisticated ranking functions are in use nowadays, TF-IDF still remains at the core and provides a clean way to measure the effect of context sensitivity.

Given a context-sensitive query $Q_c = Q_k|Q_s$, we compare the context-sensitive ranking and the conventional ranking. The conventional ranking of Q_t is equivalent to the ranking of the conventional query $Q_t(\mathcal{D}) = Q_k(\mathcal{D}) \cap Q_s(\mathcal{D})$, where Q_s is treated as a boolean filter and does not contribute to ranking scores. The measures of the precision and the reciprocal rank are shown in Figures 9.1 and 9.2, where the x-axis denotes the query ID. In Figure 9.1(a) and 9.1(b), the y-axis denotes the number of relevant results in top 20 results. In Figure 9.2(a) and 9.2(b), the y-axis denotes the reciprocal rank, whose maximum value is 1.

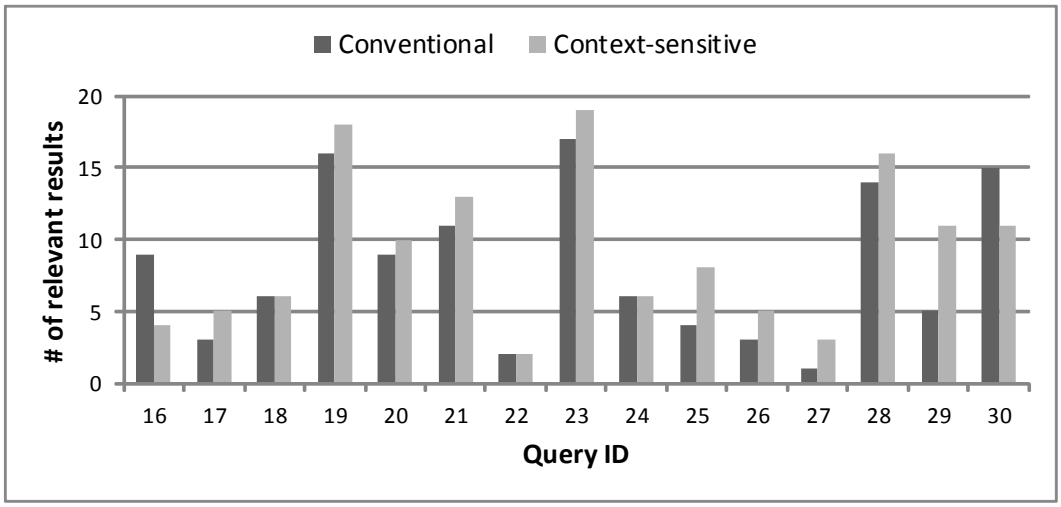
Figure 9.1(a) and 9.1(b) show that context-sensitive ranking delivers bet-

Table 9.1: MeSH Queries

QID	Keyword queries	MeSH predicates
1	serum, lupus	Proteins
2	proteins, serum, lupus	Proteins
3	cell, receptor, binding, vasoactive	D002477
4	glycan, modification, molecular	D011134
5	insect, segmentation	D007313
6	drosophila, neuroblast	D004330
7	axon, guidance	D001369
8	axon, guidance	D017173
9	axon, guidance	D017173 $\wedge$ D001369
10	actin, polymerization, muscle	D000199
11	actin, polymerization, muscle	D009132
12	actin, polymerization, muscle	D009132 $\wedge$ D000199
13	puberty, humans	D005796
14	human, homologs, rats	D011506
15	alcohol, preference	D000431
16	alcohol, preference	D005796
17	centrosomal, brain	D013568
18	activation, pmrd	D011506
19	apc, colon, cancer	D003110
20	actins, assembly, apc	D011506
21	signal, recognition, particle	D012261
22	host, solubility, proteins	D011506
23	SYMPTOMS, human, parvovirus, infection	D006801
24	Sarcoma, Ewing, PATHWAYS	D012512
25	inhibit, HIV	D006678
26	drugs, inhibit, HIV	D006678
27	membrane, fusion, HIV	D006678
28	nfkappab, signaling, pathway	D055614
29	genes, inos	D015398
30	genes, signaling, inos	D015398

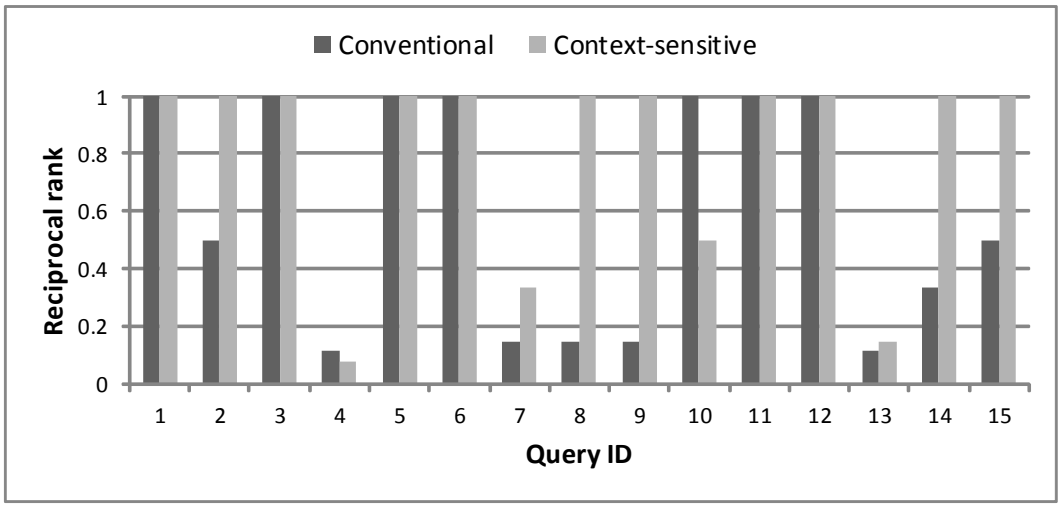


(a)

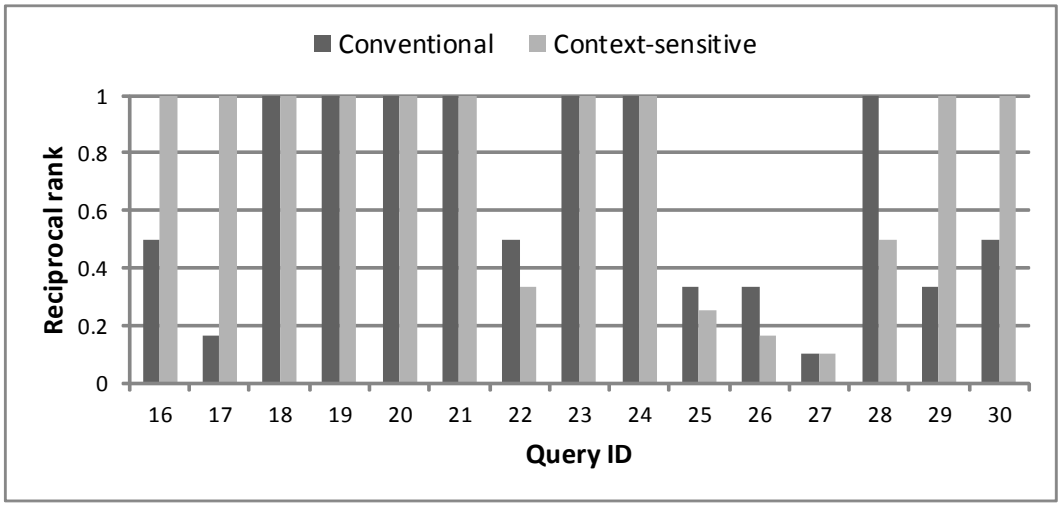


(b)

Figure 9.1: Precision in top 20 retrieved results



(a)



(b)

Figure 9.2: Reciprocal rank of top 20 results

ter ranking in 21 out of 30 queries, with occasional large improvements over conventional ranking (e.g., Q8 and Q9), while in the few occasions conventional ranking is superior (Q15, Q16, Q30) and the gap is not large. Statistically, the mean precisions of conventional ranking and context-sensitive ranking over 30 queries are 7.9 and 10.2 respectively; the mean reciprocal ranks over 30 queries are 0.62 and 0.78 respectively.

It is worth pointing out that some queries shown in Figure 9.1 and 9.2 do not benefit from context-sensitive ranking. Our observation is that ranking effectiveness depends on how well a context specification fits the original TREC query. In the experiments, the contexts are mechanically generated by PubMed's ATM mapping. We expect that context-sensitive ranking can deliver more improvements over conventional rankings for real-life queries, as their contexts are constructed by domain expects.

9.2.2 Context: More than MeSH terms

In the second set of experiments, we consider structured queries beyond combinations of MeSH terms. We also consider predicates on other structured fields of citations, such as publication year and category. Queries are constructed similar to prior experiments. That is: keyword queries are extracted from biological questions. Structured queries are constructed based on PubMed query construction tool [4]. A sample of the query workload is shown in Table 9.2.

The numbers of ranking precision and reciprocal ranking of 27 queries are shown in Table 9.3, where CSR is a shorthand for context-sensitive ranking, and CON is a shorthand for conventional ranking schemes. CMP indicates the increase/decrease of the two metrics, denoted by $\uparrow$ or $\downarrow$.

As we can see from Table 9.3, context-sensitive ranking delivers better ranking quality for most queries with structured predicates. A few decreasing cases are underlined in the table. Even for these queries, the drop of ranking quality is only indicated by one metric. On average, the precision in top 20 results is 30% better; and the reciprocal ranking is much improved.

Table 9.2: A sample of queries with structured predicates

QID	Structured Query Q_s	Keyword Query Q_k
200	Year $\geqslant$ 1998	"serum, lupus"
203	Year $\geqslant$ 1950	"receptor, vip"
205	Category = 'disease' $\wedge$ Year = 2001	"coronary, disease"
215	Year $\geqslant$ 2002 $\wedge$ Category = 'actins'	actin, polymerization
221	Year $\geqslant$ 2000 $\wedge$ Year $\leqslant$ 2003	cd44
226	Year $\leqslant$ 2008	protein, signal recognition, particle
235	Category = 'signal' $\wedge$ Category = 'transduction'	inos

9.3 View Selection

In this section, we consider how efficient the view selection algorithm is. For easy of exposition, we only consider combinations of MeSH terms as structured queries. Total number of MeSH terms are more than 65,000. In other words, the Category field yields more than 65,000 pivot columns. They post major challenges on view selection. Other columns can be negligible compared with MeSH columns.

We set T_C to 0.17% of the PubMed data set. This parameter is picked by the diagnostic tool for the target of 200 ms. PubMed has 18 million citations. Thus, the absolute value of the mining threshold is $T_C \cdot |\mathcal{D}| = 30,600$. In other words, only contexts whose sizes are greater than $30,600$ are covered by views. Query performance under this setting will be shown in Section 9.4. The maximum view size T_V is set to $2^{12} = 4096$ tuples. Note that this is the number of non-empty tuples. The actual number of pivot columns in a view can be much higher than 12.

For a materialized view V_K, while pivot columns (i.e., K) determines the number of tuples in V_K, the storage of V_K is also dependent on parameter columns, e.g., $\mathtt{len}(\mathcal{D}), \mathtt{tf}\big(\mathtt{d}(c), w_i\big)$, which are specified by a specific ranking

Table 9.3: Ranking effectiveness comparison.

QID	Precision			Reciprocal		
	CSR	CON	CMP	CSR	CON	CMP
200	13	9	44% ↑	1/1	1/3	↑
201	4	4	-	1/3	1/1	↑
202	6	0	∞ ↑	1/3	1/29	↑
203	17	13	31% ↑	1/1	1/3	↑
204	16	15	7%↑	1/2	1/1	↓
205	4	2	100% ↑	1/1	1/10	↑
206	10	10	-	1/1	1/1	↑
208	8	8	-	1/1	1/2	↑
210	1	0	∞ ↑	1/6	1/24	↑
212	8	5	60% ↑	1/3	1/4	↑
213	20	20	-	1/1	1/1	-
214	9	5	80% ↑	1/1	1/6	↑
215	6	1	500% ↑	1/2	1/13	↑
216	2	1	100% ↑	1/1	1/6	↑
217	1	0	∞ ↑	1/9	1/22	↑
218	13	12	8% ↑	1/1	1/1	-
221	10	6	67%↑	1/1	1/5	↑
223	1	1	-	1/6	1/8	↑
226	10	7	43%↑	1/2	1/3	↑
227	9	8	13% ↑	1/1	1/3	↑
228	1	1	-	1/2	1/4	↑
229	16	13	23% ↑	1/4	1/1	↓
230	7	6	17% ↑	1/1	1/3	↑
232	2	0	∞ ↑	1/2	1/27	↑
233	2	4	50% ↓	1/3	1/3	-
234	8	7	14% ↑	1/1	1/4	↑
235	10	7	43% ↑	1/5	1/2	↓
avg	**7.93**	**6.11**	**30%↑**	**1/2.4**	**1/7.0**	↑

function. In the experiments, we use the TF-IDF formula which demands document count $\mathtt{df}\big(w_i, Q_s(\mathcal{D})\big)$ of every query term which can be any keyword in the document set. Storing $\mathtt{df}\big(w_i, Q_s(\mathcal{D})\big)$ for all the keywords in the document set would result in tens of thousands of parameter columns in V_K.

In our system, V_K only stores the $\mathtt{df}\big(w_i, Q_s(\mathcal{D})\big)$ column if $|L_{w_i}| \geqslant T_C$. In other words, document counts of keywords with low frequencies are computed at query time. Consider the query $Q = w_1 \wedge w_2 | m_1 \wedge m_2$ and the materialized view $V_K, K = \{m_1, m_2, m_3\}$. Assume $|L_{w_2}| < T_C$. Then document count of w_2, which is evaluated as $|L_{w_2} \cap L_{m_1} \cap L_{m_2}|$, cannot be computed from V_K. However, since $|L_{w_2}| < T_C$, the support of $\{w_2, m_1, m_2\}$ must be less than T_C, and $L_{w_2} \cap L_{m_1} \cap L_{m_2}$ can be evaluated efficiently at query time. Notice that the evaluation of $L_{w_2} \cap L_{m_1} \cap L_{m_2}$ can start from the most selective keyword and leverage the optimization of skip pointers. The intersection $L_{m_1} \cap L_{m_2}$ is not enforced in the query plan, because collection cardinality $|L_{m_1} \cap L_{m_2}|$ and other statistics can be evaluated from V_K directly.

There are 910 keywords in the document set whose frequencies are greater than T_C. Therefore, every materialized view contains 912 parameter columns (the other two columns are context length and context cardinality). Given that the maximal number of the tuples in a materialized view is 4096, the maximal storage of a single view is 14.3 MB.

The total storage of the materialized views is 4.38 GB. For comparison, the original data set of PubMed takes 70 GB, and the Lucene index takes 5.72 GB. The average storage of a single view is 3.71 MB, which means that most views have fewer tuples than 4096. The cost of using a materialized view to compute statistics is very small. The total time of selecting and materializing views is 7.45 hours.

We use the same context size threshold (i.e., $30,600$) to select views as statistics. Each view tuple materializes one context. Similar to aggregation views, each tuple populates 912 columns to store different statistics. Total space consumed is 1.73 GB. Total materialization time is 20.1 hours.

The above experimental numbers reveal the trade-offs between space and

offline computations of the two forms of views. When using ARM to select views, we accurately enumerate queries whose contexts are greater than the context size threshold, and therefore spend longer time on execution and only materialize when we have to. When using aggregation views, graph-decomposition-based selection introduces approximation, which improves selection/materialization efficiency, but scarifies space efficiency. We may materialize more intermediate results than necessary, which can cover some non-expensive queries (all expensive queries are covered under the guarantee).

9.4 Query Performance

Next we evaluate the performance of context-sensitive queries. The complete PubMed data set is used in the experiments. The straightforward evaluation, which was described in Chapter 3, is implemented as follows: for each collection-specific statistic, a conventional keyword query that materializes the corresponding document set is constructed and sent to Lucene. After Lucene returns the document set, an aggregation is performed upon it. Consider the example query $Q = w_1 \wedge w_2 | m_1 \wedge m_2$ in Figure 3.1. Four collection-specific statistics are required for the TF-IDF function: document count for w_1, w_2, collection cardinality and collection length. Hence, three conventional queries are evaluated by Lucene: $Q_t^1 = m_1 \wedge m_2$, $Q_t^2 = w_1 \wedge m_1 \wedge m_2$ and $Q_t^3 = w_2 \wedge m_1 \wedge m_2$, upon which the required statistics can be computed. Basically, we simulate the execution plan of a context-sensitive query in Lucene by issuing multiple conventional keyword queries.

With the materialized view technique, before sending keyword queries to Lucene, collection-specific statistics are matched over the views first. If a view is usable for a collection-specific statistic, no Lucene evaluation is needed. It is possible that there are more than one views that are usable for a collection-specific statistic. In such cases, the view with the minimal size is picked.

Two categories of queries are tested in the experiments:

- **Large contexts**: queries whose context sizes are greater than T_C. They are

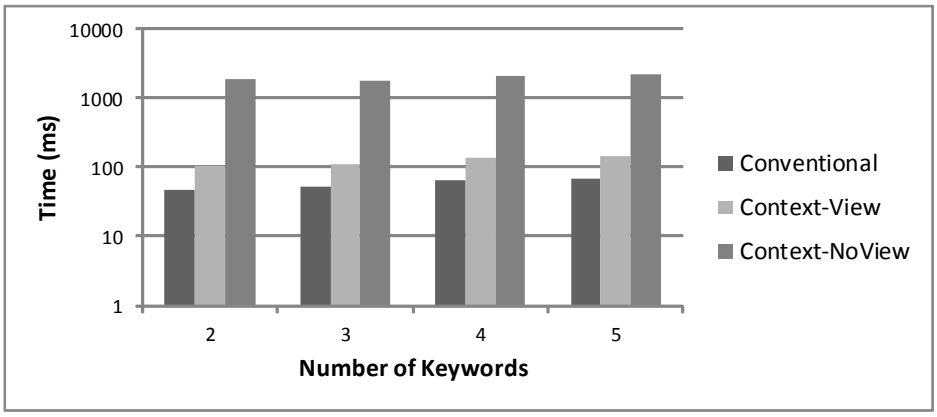


Figure 9.3: Execution time for the large-context queries

evaluated using some materialized view(s).

- **Small contexts**: queries whose context sizes are smaller than T_C. They are evaluated without any views.

The large-context queries demonstrate the effectiveness of the materialized view technique. The small-context queries show how bad the query performance could be when the context size is below T_C and the evaluation uses the straightforward approach. For each category, context-sensitive queries are randomly generated in the following way: keywords in Q_k are randomly selected from keywords in the citations' titles. Given the generated keywords, PubMed's ATM is used to map them to MeSH terms. We vary the number of keywords from 2 to 5. For each experiment, fifty queries are generated. The values shown in the following figures are the average of the fifty queries.

For a large-context query $Q_c = Q_k|Q_s$, three numbers are compared: (1) the execution time of the conventional query $Q_t(\mathcal{D}) = Q_k(\mathcal{D}) \cup Q_s(\mathcal{D})$, which returns the same result set as Q, but different ranking orders. (2) the execution time of Q with materialized views. (3) the execution time of Q without materialized views. The numbers are reported in Figure 9.3.

Figure 9.3 shows that the materialized view technique improves query efficiency significantly. Query performance of context-sensitive ranking with materialized views is about 2 times slower than the conventional queries, which

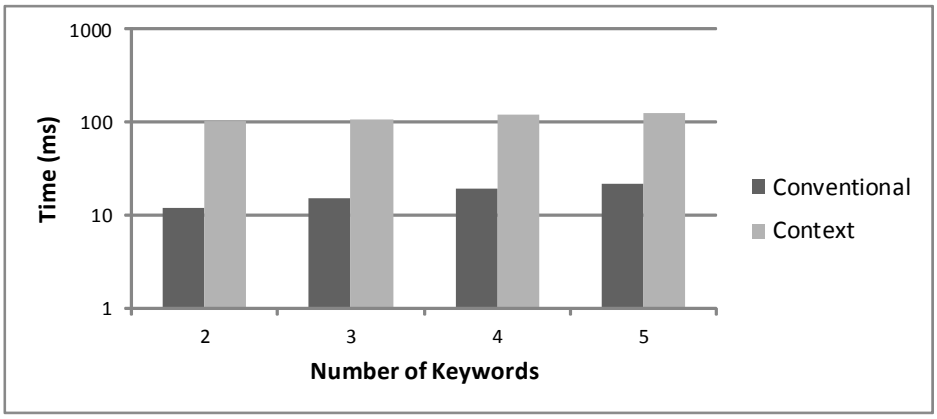


Figure 9.4: Execution time for the small-context queries

is much better than the straightforward approach. The performance drop is mainly attributed to the partial coverage of document counts for keywords in the materialized views: for a keyword w_i whose frequency is less than T_C, $df(w_i, \mathcal{D}_P)$ is computed at query time. Overall, the absolute execution time stays around 100 ms.

For a small-context query $Q_c = Q_k|P$, only two numbers are compared: (1) the execution time of the conventional query $Q_t = Q_k \cup P$, and (2) the execution time of Q_c. Note that since the Q_c's context size is smaller than T_C, no views can be used. The results are shown in Figure 9.4.

As expected, the performance decreases are much larger than the large-context queries, as every collection-specific statistic must be computed at query time. However, the absolute execution time of context-sensitive queries stays around 100 ms. Figure 9.3 and 9.4 validate our original goal for query performance: while context-sensitive ranking may sacrifice query performance to some degree, the execution time of worst-case queries should be bounded.

The experiments in Section 9.2 has shown that ranking quality is directly related to the contexts. As a special case, when the context size is too small, the statistics are much less unreliable. For example, one of the most important problems for language models is smoothing, a technique to estimate the keywords' probabilities. When the context size is too small, smoothing becomes harder. The derived language models may not achieve satisfactory ranking performance.

This means that the materialized view technique is even more important in practice: real-life queries that can benefit greatly from context-sensitive ranking are most likely to be answered by materialized views.

9.5 Acknowledgments

Parts of Chapter 9 were published in ACM SIGMOD International Conference on Management of Data, 2011, entitled “Context-sensitive Ranking for Document Retrieval”. This is joint work with Yannis Papakonstantinou. The other parts are currently being prepared for submission for publication. This is joint work with Yannis Papakonstantinou. The dissertation author is the primary investigator of the two papers.

Chapter 10

Conclusion and Future Work

10.1 Concluding Remarks

While information retrieval (IR) systems and databases evolved separately in the history, modern applications often involve both structured data and unstructured text. Such an interplay has inspired many research works from both IR and database communities. Result ranking is a problem at the core of all the existing effort. From a database's perspective, result ranking provides an effective way for users to browse relevant results first and potentially avoid examining a large number of results. From an IR's perspective, structured data have more rigid formats and semantics that can be understood by computers, thereby providing many opportunities to improve relevance ranking.

Though structured data sound promising for improving relevance ranking, their interactions with relevance ranking are not obvious on the surface. Conventional IR models and heuristics are based on a bag or a sequence of words, leaving no space for structured data. A common way to integrate structured data into ranking is to design application-specific ranking heuristics for each type of structured data, and then manipulate ranking formulas to reflect these heuristics.

Context-sensitive ranking proposed in this dissertation provides an elegant integration of structured data and relevance ranking. The essence of the ranking scheme is that structured queries regulate keyword statistics, which

eventually influence result ranking. We do not design new ranking heuristics or ranking formulas. All merits of the state-of-the-art of IR ranking models are automatically inherited.

While originally designed for content relevance, context-sensitive ranking is applicable to a much wider scope. Classification of ranking statistics (namely query-specific, document-specific and collection-specific) basically can be applied to any ranking functions of any entities. A structured query over entities will regulate the ranking of entities, as long as it changes collection-specific statistics. For instance, a location-based service ranks restaurants in a given area and recommends them to users. A restaurant ranking function, though not the same as content relevance ranking, still uses various statistics, each falling in one of the three categories. A structured query specifying a location will define collection-specific statistics over a set of restaurants near the location and regulate restaurant ranking through location-sensitive statistics. For instance, document frequency (DF) now evolves to restaurant frequency, which characterizes how discriminative a term is among restaurants around the area.

Context-sensitive ranking poses new challenges on query evaluation. The fundamental reason is that result ranking requires context-sensitive statistics, which seriously limit query optimization. Specifically, since the context must be materialized to compute statistics, query performance is mainly determined by the context size.

Our innovative reduction from statistics to aggregation queries greatly simplifies the problem and shreds new light on the connection between ranking and data analytics. Treating statistics as aggregation queries naturally leads to the view solution. Two problems arise under the umbrella of materialized views: view usability (what queries can be answered by views) and view selection (what views to materialize). We study the two problems respectively in the dissertation. Since our goal is not to study materialized views for generic queries, we concentrate on two forms of views: views as statistics caching and aggregation views as intermediate results. For view usability, we only consider single-block views without joins. The problem of view usability reduces to syntactic matching

between views and queries. For view selection, we depart from the standard problem setting in conventional databases and set the selection goal to guarantee the performance of worst-case queries.

10.2 Further Work

This dissertation builds a cornerstone for relevance ranking in the presence of structured data. One missing piece of the ranking puzzle is *structured text*. Conventionally, unstructured data often refers to text, which is typically modeled as a bag or a sequence of words. However, text data can also have embedded structures. For instance, a research paper is organized by sections, each of which may have several sub-sections. In each sub-section, text is further organized by paragraphs, sentences, and annotated semantic blocks such as definition and theorem. These structures are different from structured data and are embedded within text. We cannot decouple the two and model them separately. From the ranking perspective, structured text provides more semantic information than a sequence of words, and should provide better ranking effectiveness. Unfortunately, very little progress has been made for the ranking of structured text. While there are some works along this direction, no ranking models are widely accepted. A promising direction is to revisit ranking heuristics for structured text. Ranking heuristics are fundamental ingredients of IR. Our work revisits ranking heuristics in our new setting—context-sensitive heuristics specified by structured queries. It is natural to adopt the same philosophy for structured text. This line of works will have high impacts on industry. Structured text is widely observed in enterprise applications. For example, re-consider the email example. In addition to email headers which are considered as structured data, email bodies can be organized by conversations. Replies within a conversation form a nesting structure. How to design ranking schemes that are aware of this nesting structure is evidently important for almost every email system.

Another direction that is worth exploration is web search. Though web pages used to be less structured, many web pages also contain structured data,

such as last modified time, location and language. Furthermore, web pages are in the HTML format and have latent structures describing structured data, e.g., two-dimension web tables. Web mining [66] techniques can be used to extract and associate structured data with web pages. With progress of information extraction [83], structured data, such as names or locations, can be even extracted from text. All these structured data evidently can be used to improve search experience and ranking quality of search engines. Though web search is mainly based on keyword queries with no explicit structured predicates, recent works [84, 68] have been working on parsing keyword queries into structured forms. Combined with such query parsing techniques, context-sensitive ranking can be applied in web search, without changing existing sophisticated ranking formulas. Our techniques on improving query efficiency can be used as well. The challenge of this line of work, however, is that structured data associated web pages may not be accurate, because of careless web page publishers or extraction errors. Query semantics and ranking mechanism must be able to tolerate such errors. Current query semantics and ranking schemes may not be sufficient, because a structured query has a precise semantics and all documents that do not satisfy it are removed from results.

Bibliography

[1] http://bioinfo.amc.uva.nl/human-genetics/pubreminer/.

[2] http://lucene.apache.org/.

[3] http://www.json.org/.

[4] http://www.ncbi.nlm.nih.gov/pubmed/.

[5] http://www.w3.org/xml/.

[6] Rakesh Agrawal, Ralf Rantzau, and Evimaria Terzi. Context-sensitive ranking. In *SIGMOD*, 2006.

[7] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.

[8] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[9] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[10] Shurug Al-Khalifa, Cong Yu, and H. V. Jagadish. Querying structured text in an xml database. In *SIGMOD*, 2003.

[11] Sihem Amer-Yahia, Emiran Curtmola, and Alin Deutsch. Flexible and efficient xml search with complex full-text predicates. In *SIGMOD*, 2006.

[12] Avi Arampatzis and Jaap Kamps. A study of query length. In *SIGIR*, 2008.

[13] Ricardo A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *CPM*, pages 400–408, 2004.

[14] Ricardo A. Baeza-Yates and Alejandro Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *SPIRE*, pages 13–24, 2005.

[15] Nilesh Bansal, Sudipto Guha, and Nick Koudas. Ad-hoc aggregations of ranked lists in the presence of hierarchies. In *SIGMOD Conference*, pages 67–78, 2008.

[16] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. Materialized views selection in a multidimensional database. In *VLDB*, 1997.

[17] Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu. Faster adaptive set intersections for text searching. In *WEA*, pages 146–157, 2006.

[18] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[19] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[20] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[21] David M. Blei and John D. Lafferty. Correlated topic models. In *In Proceedings of the 23rd International Conference on Machine Learning*, 2006.

[22] Chavdar Botev and Jayavel Shanmugasundaram. Context-sensitive keyword search and ranking for xml. In *WebDB*, pages 115–120, 2005.

[23] Andreas Broschart and Ralf Schenkel. Proximity-aware scoring for xml retrieval. In *SIGIR*, 2008.

[24] J. Brutlag. Speed matters for google web search. http://code.google.com/speed/files/delayexp.pdf. 2009.

[25] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Inf. Process. Lett.*, 42(3), 1992.

[26] David Carmel, Yoëlle S. Maarek, Matan Mandelbrod, Yosi Mass, and Aya Soffer. Searching xml documents via xml fragments. In *SIGIR*, 2003.

[27] Soumen Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *WWW*, 2007.

[28] Chee Yong Chan and Yannis E. Ioannidis. Hierarchical prefix cubes for range-sum queries. In *VLDB*, 1999.

[29] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.

[30] Surajit Chaudhuri, Kenneth Ward Church, Arnd Christian König, and Liying Sui. Heavy-tailed distributions and multi-keyword queries. In *SIGIR*, 2007.

[31] Liang Jeff Chen and Yannis Papakonstantinou. Supporting top-k keyword search in xml databases. In *ICDE*, pages 689–700, 2010.

[32] Charles L. A. Clarke. Controlling overlap in content-oriented xml retrieval. In *SIGIR*, 2005.

[33] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, 2003.

[34] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Equivalences among aggregate queries with negation. In *PODS*, 2001.

[35] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *PODS*, pages 155–166, 1999.

[36] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering top-k queries using views. In *VLDB*, pages 451–462, 2006.

[37] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *SODA*, pages 743–752, 2000.

[38] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, pages 91–104, 2001.

[39] Andreas Doms and Michael Schroeder. Gopubmed: exploring pubmed with the gene ontology. *Nucleic Acids Research*, 33(Web-Server-Issue), 2005.

[40] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[41] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[42] Hui Fang, Tao Tao, and ChengXiang Zhai. A formal study of information retrieval heuristics. In *SIGIR*, 2004.

[43] Uriel Feige, Mohammad Taghi Hajiaghayi, and James R. Lee. Improved approximation algorithms for minimum-weight vertex separators. In *STOC*, 2005.

[44] Norbert Fuhr and Kai Großjohann. XIRQL: a query language for information retrieval in XML documents. In *SIGIR*, 2001.

[45] Torsten Grabs and Hans-Jörg Schek. Powerdb-xml: Scalable xml processing with a database cluster. In *Intelligent Search on XML Data*, 2003.

[46] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, 1996.

[47] Stéphane Grumbach, Maurizio Rafanelli, and Leonardo Tininini. Querying aggregate data. In *PODS*, pages 174–184, 1999.

[48] Stéphane Grumbach and Leonardo Tininini. On the content of materialized aggregate views. In *PODS*, pages 47–57, 2000.

[49] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD*, 2003.

[50] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[51] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112, 1997.

[52] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, pages 453–470, 1999.

[53] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.

[54] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, 8(1), 2004.

[55] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

[56] Taher H. Haveliwala. Topic-sensitive pagerank. In *WWW*, 2002.

[57] William R. Hersh and Ellen M. Voorhees. Trec genomics special issue overview. *Inf. Retr.*, 12(1), 2009.

[58] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range queries in olap data cubes. In *SIGMOD*, 1997.

[59] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[60] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD Conference*, pages 259–270, 2001.

[61] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[62] Heasoo Hwang, Andrey Balmin, Berthold Reinwald, and Erik Nijkamp. Binrank: Scaling dynamic authority-based search using materialized subgraphs. In *ICDE*, 2009.

[63] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[64] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *WWW*, 2003.

[65] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD*, 2004.

[66] Raymond Kosala and Hendrik Blockeel. Web mining research: A survey. *SIGKDD Explorations*, 2(1):1–15, 2000.

[67] Georgia Koutrika and Yannis E. Ioannidis. Personalization of queries in database systems. In *ICDE*, 2004.

[68] Xiao Li, Ye-Yi Wang, and Alex Acero. Extracting structured information from user queries with semi-supervised conditional random fields. In *SIGIR*, pages 572–579, 2009.

[69] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.

[70] G. Linden. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.

[71] Fang Liu, Clement T. Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *SIGMOD Conference*, pages 563–574, 2006.

[72] Xiaoyong Liu and W. Bruce Croft. Cluster-based retrieval using language models. In *SIGIR*, 2004.

[73] Ziyang Liu and Yi Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, pages 329–340, 2007.

[74] Ziyang Liu and Young Chen. Reasoning and identifying relevant matches for xml keyword search. *PVLDB*, 1(1):921–932, 2008.

[75] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD Conference*, pages 115–126, 2007.

[76] Zhongming Ma, Gautam Pant, and Olivia R. Liu Sheng. Interest-based personalized search. *ACM Trans. Inf. Syst.*, 25(1).

[77] Nikos Mamoulis, Kit Hung Cheng, Man Lung Yiu, and David W. Cheung. Efficient aggregation of ranked inputs. In *ICDE*, page 72, 2006.

[78] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, 2008.

[79] Werner Nutt, Yehoshua Sagiv, and Sara Shurin. Deciding equivalences among aggregate queries. In *PODS*, pages 214–223, 1998.

[80] Feng Qiu and Junghoo Cho. Automatic identification of user interest for personalized search. In *WWW*, 2006.

[81] Stephen E. Robertson, Hugo Zaragoza, and Michael J. Taylor. Simple bm25 extension to multiple weighted fields. In *CIKM*, 2004.

[82] Michal Rosen-Zvi, Thomas Griffiths, Mark Steyvers, and Padhraic Smyth. The author-topic model for authors and documents. In *UAI*, 2004.

[83] Sunita Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.

[84] Nikos Sarkas, Stelios Paparizos, and Panayiotis Tsaparas. Structured annotations of web queries. In *SIGMOD Conference*, pages 771–782, 2010.

[85] Feng Shao, Lin Guo, Chavdar Botev, Anand Bhaskar, Muthiah M. Muthiah Chettiar, Fan Yang 0002, and Jayavel Shanmugasundaram. Efficient keyword search over virtual xml views. In *VLDB*, pages 1057–1068, 2007.

[86] Xuehua Shen, Bin Tan, and ChengXiang Zhai. Context-sensitive information retrieval using implicit feedback. In *SIGIR*, 2005.

[87] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, 1998.

[88] Ahu Sieg, Bamshad Mobasher, and Robin D. Burke. Web search personalization with ontological user profiles. In *CIKM*, 2007.

[89] Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4), 2001.

[90] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *VLDB*, 1996.

[91] Chong Sun, Chee Yong Chan, and Amit K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.

[92] Jaime Teevan, Susan T. Dumais, and Eric Horvitz. Personalizing search via automated analysis of interests and activities. In *SIGIR*, 2005.

[93] Anja Theobald and Gerhard Weikum. The index-based xxl search engine for querying xml data with relevance ranking. In *EDBT*, 2002.

[94] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. An efficient and versatile query engine for topx search. In *VLDB*, 2005.

[95] Jeffrey Scott Vitter and Min Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *SIGMOD Conference*, 1999.

[96] Ellen M. Voorhees. Overview of the trec-9 question answering track. In *TREC*, 2000.

[97] Ryen W. White and Dan Morris. Investigating the querying and browsing behavior of advanced search engine users. In *SIGIR*, 2007.

[98] Jens E. Wolff, Holger Flörke, and Armin B. Cremers. Searching and browsing collections of structural information. In *IEEE Advances in Digital Libraries*, 2000.

[99] Dong Xin, Jiawei Han, and Kevin Chen-Chuan Chang. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *SIGMOD Conference*, pages 103–114, 2007.

[100] Shengliang Xu, Shenghua Bao, Ben Fei, Zhong Su, and Yong Yu. Exploring folksonomy for personalized search. In *SIGIR*, 2008.

[101] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD Conference*, pages 537–538, 2005.

[102] Yu Xu and Yannis Papakonstantinou. Efficient lca based keyword search in xml data. In *EDBT*, pages 535–546, 2008.

[103] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.

[104] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex sql queries using automatic summary tables. In *SIGMOD Conference*, pages 105–116, 2000.

[105] Mohammed Javeed Zaki. Scalable algorithms for association mining. *IEEE Trans. Knowl. Data Eng.*, 12(3), 2000.