Techniques for improving efficiency and scalability for the integration of information retrieval and databases

Wu, Hengzhi

Techniques for Improving Efficiency and Scalability for the Integration of Information Retrieval and Databases

Hengzhi Wu

University of London

Thesis submitted for the degree of Doctor of Philosophy

at Queen Mary, University of London

May 2010

Declaration of originality

I hereby declare that this thesis, and the research to which it refers, are the product of my own work, and that any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

The material contained in this thesis has not been submitted, either in whole or in part, for a degree or diploma or other qualification at the University of London or any other University. Some parts of this work have been previously published as:

[Wu and Roelleke, 2009] Wu, H. and Roelleke, T. (2009). Semi-subsumed events: A probabilistic semantics of the BM25 term frequency quantification. In *ICTIR*, pages 375–379.

[Roelleke et al., 2008] Roelleke, T., Wu, H., Wang, J., and Azzam, H. (2008). Modelling retrieval models in a probabilistic relational algebra with a new operator: the relational Bayes. *VLDB J.*, 17(1):5–37.

[Wu et al., 2008a] Wu, H., Kazai, G., and Roelleke, T. (2008a). Modelling anchor text retrieval in book search based on back-of-book index. In *SIRIG Workshop on Focused Retrieval*, pages 51–58.

[Wu et al., 2008b] Wu, H., Kazai, G., and Taylor, M. (2008b). Book search experiments: Investigating IR methods for the indexing and retrieval of books. In *ECIR*, pages 234–245.

[Roelleke et al., 2005] Roelleke, T., Ashoori, E., Wu, H., and Cai, Z. (2005). The QMUL team with probabilistic SQL at enterprise track. In *TREC*.

Abstract

This thesis is on the topic of integration of Information Retrieval (IR) and Databases (DB), with particular focuses on improving efficiency and scalability of integrated IR and DB technology (IR+DB). The main purpose of this study is to develop efficient and scalable techniques for supporting integrated IR and DB technology, which is a popular approach today for handling complex queries over text and structured data.

Our specific interest in this thesis is how to efficiently handle queries over large-scale text and structured data. The work is based on a technology that integrates probability theory and relational algebra, where retrievals for text and data are to be expressed in probabilistic logical programs such as probabilistic relational algebra or probabilistic Datalog. To support efficient processing of probabilistic logical programs, we proposed three optimization techniques that focus on aspects covered logical and physical layers, which include: scoring-driven query optimization using scoring expression, query processing with top-k incorporated pipeline, and indexing with relational inverted index.

Specifically, scoring expressions are proposed for expressing the scoring or probabilistic semantics of implied scoring functions of PRA expressions, so that efficient query execution plan can be generated by rule-based scoring-driven optimizer. Secondly, to balance efficiency and effectiveness so that to improve query response time, we studied methods for incorporating top-k algorithms into pipelined query execution engine for IR+DB systems. Thirdly, the proposed relational inverted index integrates IR-style inverted index and DB-style tuple-based index, which can be used to support efficient probability estimation and aggregation as well as conventional relational operations.

Experiments were carried out to investigate the performances of proposed techniques. Experimental results showed that the efficiency and scalability of an IR+DB prototype have been improved, while the system can handle queries efficiently on considerable large data sets for a number of IR tasks.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Thomas Roelleke, this thesis would not have been possible if without his valuable advice and support in so many areas. His research knowledge, technical understanding, common sense, patience, enthusiasm and friendship guided me and encouraged me throughout the time of my PhD study and research.

I owe my deepest gratitude to Dr. Gabriella Kazai for guiding me with initial experiments when she was a colleague of mine in Queen Mary, as well as for mentoring me in the Book Search project during my internship in Microsoft Research Cambridge. Her wide knowledge in information retrieval and expertise in evaluation have been of great value for me.

I am deeply grateful to Professor Mounia Lalmas for her valuable comments and pre-reviews to a number of my publications and paper submissions.

I warmly thank Dr. Tassos Tombros and Professor Shaogang Gong, who examined me for my PhD progression when we had an inspiring discussion on the methodology for writing a PhD thesis, such as how to make individual contributions to be cohered as a whole in a thesis.

My warm thanks to my current and former lab-mates in Queen Mary Information Retrieval Group: Jun Wang, Elham Ashoori, Frederik Forst, Zoltan Szlavik, Hany Azzam, Sirvan Yahyaei, Gabriella Kazai, Theodora Tsikrika, Zhigang Kong, You-Jin Chan, Shanu Sushmita, Simon Carter and Shanu Sushmita. In particular, special thanks to Jun Wang who have helped me with setting up experiments and understanding IR models.

I am grateful to my PhD examiners Dr. Andrew MacFarlane and Dr. Ralf Schenkel, who provided detailed and valuable suggestions and their specialist views for helping me to improve the presentation of my thesis.

Thanks to the Department of Computer Science for keeping me in offices, stationary, and for funding my summer school to Dublin and various conferences. Thanks to the system stuffs for maintaining machines and providing me experimental environments.

Thanks to various other friends and partners in crime: Jiayi Wu, Weizhi Luo, Jin Luo, Xiang Li.

Thanks to Microsoft Research Cambridge for funding my travel to SIGIR 2008 conference.

During this work I have collaborated with many colleagues for whom I have great regard, and I wish to extend my warmest thanks to all those who have helped me with my work in the Department of Computer Science of Queen Mary and Microsoft Research Cambridge.

Thanks finally for support from family: Mum, Dad, the Wus and the Leis.

The financial support of the Queen Mary University of London is grateful acknowledged.

Chapter 1

Introduction

1.1 Research Background of the Thesis

Information retrieval (IR) has being recognised as one of the most popular research topics in the field of information management. Originally born as a pure librarian management technology, IR has been developed as a main technology to retrieve information from free-text documents. Though in the modern information retrieval [Baeza-Yates and Ribeiro-Neto, 1999a], IR technologies have evolved to handle not only text but also other data formats such as multimedia data (e.g. image, video, speech etc.), nevertheless, text retrieval still remains as a main research area of IR.

In particular, with rapid growth of the Internet, the amount of text available on the Web explosively increased, which catalysed IR to develop competent techniques to handle massively large-scale data efficiently. Moreover, new research agendas for IR to handle complex queries have also been called out. For example, several challenges to IR research for enterprise search are discussed in [Hawking, 2004], in which versatility and customisability of IR engines became one of the main concerns to tackle the challenges led by complex search space.

On the other hand, as another major field that dedicates in information management systems, database (DB) community had been driven by very different discipline from IR in the past (e.g. see [Rijsbergen, 1979]): DB focused on data models, structured data (records), deduction, and artificial query languages (e.g. SQL); whereas IR focused on ranking models, unstructured data (free-text), induction, and natural query language (e.g. keywords). However, the situation is

radically different today. As it was pointed out in [Chaudhuri et al., 2005]:

> "Virtually all advanced applications need both structured data and text documents, and information fusion is a central issue. Seamless integration of structured data and text is at the top of the wish lists of many enterprises."

Actually, researchers had foreseen similar demands in early 1980s with regards to integrating structured data and text into information management systems, e.g. see [Schek and Pistor, 1982]. In recent years, demands on the integration of IR and DB technologies keep growing, which can be seen from both research agendas and industrial developments:

- For IR community, the search engine giant Google developed a DB-like structured storage called BigTable [Chang et al., 2006], while other similar systems including HBase/Hadoop[1], and SimpleDB[2] by Amazon.
- For DB community, the Lowell report (2003) [Abiteboul et al., 2003] and Claremont report (2008) [Agrawal et al., 2008] on database research self assessment continuously considered integration of DB and IR as highly interested topic; the call for special issue on integration of DB&IR by VLDB journal [Croft and Schek, 2008] emphasises DB's interest; and related publications to the topic frequently appeared in several top-tier DB conferences.

This thesis studies the techniques for integrating IR and DB technologies, with specific focus on improving efficiency and scalability for one of the approaches, named IR+DB. In the rest of this chapter, we will have a glance at the integration technologies, and will address the outline of the thesis.

1.1.1 Integration of Information Retrieval and Databases at a Glance

> "Search engines are structurally similar to database systems" [Zobel and Moffat, 2006]

Information Retrieval (IR) and Database (DB) are currently separated technologies with respect to information management, however, they have similar research goals in principle, which

[1] http://wiki.apache.org/hadoop/Hbase
[2] http://aws.amazon.com/simpledb/

is to answer queries of information need. On the one hand, researches in the two fields had been driven by the two communities towards relatively different directions: IR technology mainly handles keyword-based queries in natural languages over unstructured data, whereas DB technology mainly handles logical queries in artificial languages over structured data; IR retrieves relevant information based on similarity between queries and information items, whereas DB retrieves records based on matching between queries and data; IR benchmarks focus on measuring effectiveness, whereas DB benchmarks focus on measuring efficiency.

Nevertheless, the situation of considering IR and DB as totally apart areas have been changed, because modern information applications require information management systems to be capable to handle both text and structured data efficiently and effectively. As a result, integrating IR and DB technologies became a popular topic for research, because integrated IR and DB systems may take advantages from both fields.

For example, considering the following scenarios:

1. An intranet search engine in a large corporation. The engine supports keyword-based query and manipulates retrieval among corporation's internal web sites, resource databases, employee forum, mailing lists. Therefore, the searchable data sources including web pages, database records, discussion lists, and emails. The intranet search engine should be able to handle different data sources and merge ranked results. In addition, the engine should be sufficiently efficient for employees.

2. A book search Web service. The Web Service is powered by a search engine which incorporates with libraries, book sellers and readers/editors reviewing web sites. The search engine indexes text and metadata of books, web pages of various sources. The service accepts keyword-based query with additional indication of intentions, e.g. looking for reference or buy books, and returns a ranked list of books.

3. A price monitoring agent of online shops. The agent is based on an automatic crawler periodically crawling competitors' web sites, and it can compare the prices of commodities of competitors to the shop's own prices in database, and then ranks competitors' web sites based on their degree-of-competitive.

Based on different architectures, various approaches for the integration of IR and DB have been proposed; while IR+DB is one of the approaches that look interesting to us, because it

takes a thorough view of the problems with regards to text and structured data retrieval, so that it tends to absorb the most necessary and effective techniques from IR and DB, while it avoids the functionality of existing systems that could be overloaded for integrated IR and DB systems.

1.1.2 Motivation of Research

The motivation of our studies is to investigate suitable techniques to tackle the efficiency and scalability problems restricting IR+DB technology to be applicable for very large data sets.

On the one hand, both IR and DB have developed very successful techniques for efficiently handling very large and ever growing data sets; in other words, both mainstream IR engines and commercial DB systems have been developed to be efficient and scalable based on respective criteria.

On the other hand, because of different business or application models, the criteria of being efficient and scalable for IR and DB systems are quite different. For instance, a de facto standard for Web search engines in nowadays to be acceptable for casual users is to respond queries in sub-seconds; whereas for database management systems (DBMS) in banks, it would be sufficient enough for DBMS to process transactions in several hours.

Therefore, in terms of the applied areas of IR+DB systems, we consider more IR-oriented applications which do not involve traditional DB criteria such as ACID (Atomicity, Consistency, Isolation, Durability) for transaction, while these applications may benefit from relational operations (such as in DB) in combining multiple sources or evidences.

As a result, our motivation is to improve the efficiency and scalability of IR+DB systems supporting probabilistic relational algebra (PRA) [Fuhr and Roelleke, 1998] as a query language, especially to speed up the query processing in such systems for complex queries involving expensive (in terms of time consumption) PRA expressions.

1.2 Research Problems

This research hypothesis of this thesis is as follow:

Hypothesis. IR and DB integrated systems can be speeded up and scaled up by adapting and evolving existing techniques in information retrieval and databases.

This thesis answers the following three research questions.

1. How to optimize probabilistic relational algebra expressions so that to generate logical or physical query plans that can be processed efficiently?

2. How to incorporate top-*k* processing mechanisms into generic query execution engine of IR+DB systems or infrastructures.

3. How to adapt IR-style indexing methods into an IR+DB platform, so that to provide efficient accessibility to statistics that are needed for flexible scoring and ranking, and how to design and implement such index that is scalable for large-scale data?

1.3 Outline of the Proposed Techniques in the Thesis

In this thesis, we propose and investigate three main techniques for improving the efficiency and scalability of IR+DB system, which cover three main aspects (layers) with regards to logical query optimization, physical query processing, and storage and indexing method (see Figure 1.1). The three techniques aim to improve the processing performance with regards to efficiency and scalability of IR+DB system from two angles: on the one hand, we proposed a scoring-driven optimization method from a logical optimization point of view; on the other hand, we proposed a top-*k* incorporated pipeline and a relational inverted index from a physical optimization point of view. Though our starting point is to optimize the query evaluation for probabilistic relational algebra (PRA) expressions, but these techniques may also benefit a broader range of IR+DB systems.

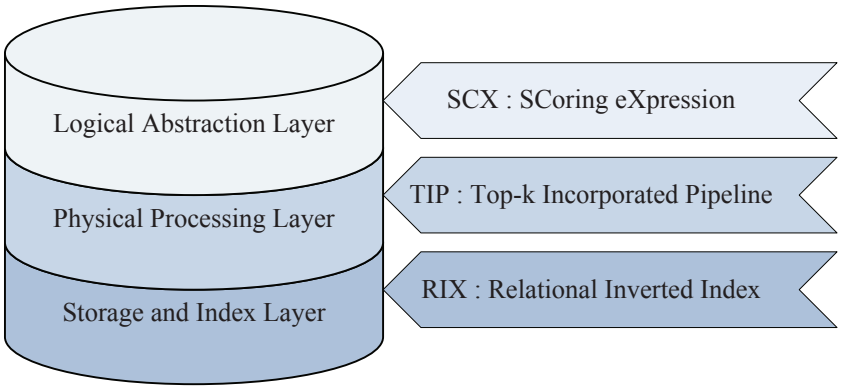


Figure 1.1: Proposed techniques in the layers of an IR+DB system

The features of the three techniques are outlined in the following subsections.

1.3.1 Scoring-Driven Optimization

A scoring-driven optimization is a new class of query optimization for PRA which can be applied in parallel with other two types of techniques: algebraic optimization and cost-driven optimization.

Algebraic optimization and cost-driven optimization are two main state-of-the-art query optimizations that widely applied in database systems, where the former rewrites relational algebra expressions (or relational expressions for short) into other transformations based on algebraic equivalence, whereas the latter tries to optimize the mapping from logical query plan to physical execution plan, which aims to choose the least expensive implementations based on pre-defined cost models.

Scoring-driven optimization can be categorised into logical optimization such as algebraic optimization, but it is substantially different from algebraic optimization by focusing on the aspect of scoring functions that is not considered by algebraic optimization. Because a traditional relational algebra (RA) does not involve scoring functions, so that an algebraic optimization for RA only need to consider relational equivalence. However, a probabilistic relational algebra (PRA, see e.g. [Fuhr and Rölleke, 1997, Roelleke et al., 2008]) incorporates scoring functions internally, so that logical optimization for PRA expressions is more complicated, because an optimizer needs to consider not only relational semantics but also scoring semantics.

The introduced scoring-driven optimization technique articulates scoring function for PRA expression using scoring expression (SCX): first of all, it interprets scoring semantics of PRA expressions while considering relational semantics as well; while optimizer finds the scoring function implied by a PRA expression based on extensional semantics is incompatible to intensional semantics (based on possible worlds model, see Section 2.4), the method adjusts articulating scoring expression and aligns it to intensional semantics; in addition, the technique helps to verify algebraic equivalent PRA expressions based on scoring equivalence (see Section 3.3.2).

In a word, scoring-driven optimization may becomes a new direction of research of logical optimization for IR+DB systems.

1.3.2 Top-*k* Incorporated Pipeline

Top-*k* processing is a query processing method aims to shorten query response time, which has been widely applied in both IR search engines and database systems. The basic idea is that to

only compute the results from the most likely relevant data and try to stop the process as soon as enough results have been produced. To employ top-*k* mechanisms in IR+DB, we investigate how to integrate top-*k* algorithms into a pipelined query execution engine for IR+DB systems, and we proposed a top-*k* incorporated pipeline (TIP) for the purpose.

In a pipelined query execution engine for IR+DB, complex queries can be viewed as assemblies of multiple common query blocks, while a query block is a form of a Select-Estimate-Aggregate (SEA) query (see Section 4.3.1), which means a list of ranked result is computed by common physical operations of selection, probability estimation and probability (or score) aggregation, so that top-*k* algorithms can be incorporated into (physical) probability estimators and probability aggregators.

Applicability of a well-known top-*k* algorithm named threshold algorithm (TA) and its variants had been investigated, where variants based on no random access (NRA) algorithm and combined algorithm (CA) were considered in an conceptual design of TIP (see Section 4.3.2). Moreover, NRA-style top-*k* limits allotting strategies were investigated for *tf*-*idf* model, where three different strategies based on uniform allotment and IDF were studied. In addition, in order to estimate the performances tradeoff with respect to efficiency versus effectiveness, an ideal measuring method was introduced to estimate the tradeoff points of top-*k* processing (see Section 4.3.3). Experiments were carried out for investigating the performances tradeoff while top-*k* mechanism applied.

In short, a generic top-*k* integrated query processing engine is highly intriguing for integrated IR and DB systems.

1.3.3 Relational Inverted Index

A relational inverted index (RIX) was proposed specifically for IR+DB systems. It employs special index structure that combines IR-style inverted index (i.e. inverted files) and DB-style TID-list index (i.e. tuple-identifier-based index).

By utilising inverted lists for indexing basic IR statistics such as term frequencies and document frequency, RIX provides similar facilities as those for conventional IR systems to IR+DB systems, so that efficient retrieval for frequently used statistics is enabled to IR+DB query engine; in other words, RIX supports efficient processing for queries applying popular IR ranking models and retrieval strategies.

Moreover, RIX integrates TID-lists index to support relational operations on IR+DB sys-

tems. Comparing to dedicated IR systems, to be flexible to combine multiple data (of information) sources is one of the main advantages of integrated IR and DB systems, and TID-lists index is widely used in traditional databases to speed up query processing of relational algebra expressions.

Because PRA combines probability theory and relational model, so that users can implement scoring functions by formulizing PRA expressions. On the other hand, it is desired by a PRA query engine for a versatile and efficient indexing method, which can provide flexible and scalable infrastructures to support and speed up the processing procedures of probability estimation, aggregation, and relational operations.

For RIX, we will discuss the index structures and construction methods; in addition, we will discuss indexing algorithms that aims to reduce the overhead of I/O operations; in particular, we will investigate several building strategies for different scales of source data in order to achieve a sub-optimal constructing performance; and moreover, we will introduce the accessing and retrieval methods.

1.4 Overview of the Thesis

The remainder of this thesis is organised as follows:

Chapter 2: Integration of Information Retrieval and Databases The chapter reviews the state-of-the-art technologies of information retrieval and databases, in particular, it pays specific attentions to the techniques that are related to the integration of IR and DB. It introduces backgrounds such as ranking models of IR, ranked databases, probabilistic databases and various integrating approaches of IR and DB.

Chapter 3: Scoring-Driven Query Optimization with Scoring Expression The chapter discusses a logical query optimization technique which is driven by scoring functions that are associated to probabilistic relational algebra (PRA) expressions. It addresses the specifications of scoring expression (SCX), and it discusses the details of applying SCX to conduct scoring-driven query optimization for PRA expressions.

Chapter 4: Query Processing with Top-k Incorporated Pipeline The chapter introduces top-k incorporated pipeline (TIP) which is to be employed in a physical query execution engine; in addition, it investigates the performances tradeoff while applying NRA-style top-k mechanisms for executing PRA queries for *tf-idf* model.

Chapter 5: Indexing with Relational Inverted Index The chapter presents a relational inverted index for IR+DB systems, in which it discusses the indexing structures, constructing algorithms, and retrieval modes of the index.

Chapter 6: Summary and Conclusion The chapter concludes this thesis, where it discusses some other potential techniques that are mature and popular but have not yet been considered and investigated in this thesis; furthermore, it also takes a prospective view on potential future work.

Appendix A: Getting Started with Birdie This appendix gives a quick start guide for the IR+DB prototype named Birdie in which the proposed techniques of this thesis were implemented. In addition, the appendix also presents in short the inside architecture of the prototype.

Appendix B: Full MagazineCorpus Table This appendix illustrates a table for a toy magazine corpus.

Chapter 2

Integration of Information Retrieval and Databases

2.1 Introduction

In this chapter, we introduce concepts and backgrounds of the integration of Information Retrieval (IR) and Databases (DB), in which we introduce state-of-the-art technologies and systems, and in particular, address some known issues with regard to efficiency and scalability.

Traditionally, IR and DB are distinguished technologies under the same taxonomy of information management. Though the two have very similar aim namely satisfying information need of query, they have been developed toward different directions:

- IR is expert on handling unstructured data such as text, special indexing techniques were proposed to support efficient search over text data; whereas DB is mastered on dealing with structured data record, various data models were developed to support complex queries, and DB indexes allow efficient query evaluation (i.e. query processing) using relational operations.

- IR developed dedicated techniques and search optimization for specific application domains respectively, while flexibility and customisability were usually less considered; whereas DB is interested in data models, high level abstraction and controllable query optimization, and flexibility and customisability are highlighted features to be emphasised.

- IR retrieves documents (traditionally), and it defines retrieval effectiveness based on relevance of retrieved document to query; whereas DB retrieves data records, and it defines

retrieval quality by if matches of data records against query were found. IR benchmarks mainly (and almost only) focus on effectiveness (e.g. TREC and INEX); whereas DB benchmarks are varied (e.g. TPC benchmarks), but in most benchmarks efficiency is an important aspect to be tested following a cost-performance model (e.g. Debit/Credit).

Today situations have changed. Since any advanced application today uses both text documents as well as databases, either community (IR or DB) has to confront a territory it was not familiar before. As a result, the demands on information systems which can search on both text and record effectively and efficiently provoked great enthusiasm on research and development of integrating IR and DB technologies.

The authors of [Chaudhuri et al., 2005] reviewed the issues on integrating IR&DB deeply and widely. They pointed out the shortages of processing queries on different types of data of IR and DB: in short, there is no query optimization for advanced queries in the IR world, and insufficient text support in the DB world. There have been existing built-from-scratch DB+IR systems such as QUIQ [Kabra et al., 2003] aim to handling structured data and text fields and supporting scoring for similarity search, but they were designed to be domain-oriented systems rather than universal infrastructure. Nevertheless, it is glad to see efforts on building generic integrated IR&DB platforms had been carried out.

Although motivations for integrating IR and DB might be varied, but benefits of integrated IR&DB technologies could be summarised as the follows:

- Flexibility and customisability: it offers expressive declarative query languages for implement ranking functions and retrieval strategies while developing IR related applications.
- Easy development and maintenance: from an engineering perspective, the advantages of using high level declarative languages include shortening development circles, easing source code maintenance, reducing difficulties and risks, and tolerant for changes.
- Handling complex queries: the underlying (probabilistic) relational model of database provide computing powers to handle queries involving complex relationships and conditions, which is usually missing in traditional IR systems.
- Seamlessly handling queries over multi-format data sources: unstructured data (e.g. text) and structured data (e.g. record) are integrated naturally, therefore, queries over multi-format data sources could be processed seamlessly on an integrated framework.

In the remainder of this chapter, we review the state of the art techniques on text retrieval and efforts of integrating scoring or ranking functions into databases. In particular, we recall essential concepts and theories behind probabilistic databases and take a detailed look at the query evaluation technique on probabilistic database.

2.2 A Brief Review of Information Retrieval

In short, the task of IR is to look for information items (e.g. documents) from a collection (e.g. a corpus) that relate to given queries, where if an information item in the collection is related to a query, then it is called relevant information for the query, otherwise is called irrelevant information. In the words, IR aims to retrieve information that are *about* queries, which is different from traditional DB task that aims to retrieve information (i.e. data records) that *match* queries.

In this section, we review basic IR procedures for traditional text retrieval with an example, and then revisit some dominant retrieval models that are applied by IR systems.

2.2.1 Basic Procedures of Information Retrieval

In general, there are five basic procedures in a typical IR system, which are:

- *preprocessing*: to prepare input for indexing. The procedure may include several sub-processes, for instance, parsing of documents, i.e. extract words from documents; casefold, i.e. change uppercase letters to lowercase; stopword removal, e.g. remove pronouns and prepositions; and stemming, i.e. to cut off suffixes of plural, the past tense, or the present continuous tense, etc..

- *indexing*: to construct either meta-index or full-text index. A meta-index contains meta data of indexed collection and provides accessing methods to original documents, whereas a full-text index is an additional representation of the original collection with extra statistics included.

- *query processing*: to retrieve lists of ranked relevant documents with regard to given queries. This procedure conducts the cored function of an IR system, i.e. for a given query, it retrieves a list of candidate relevant documents, and then ranks the candidates based on predefined ranking model(s). Note that extra sub-processes could be employed before conducting main retrieval process. For example, determining if a query is a list of terms (words) or a phrase, or deploying query expansion.

- *result preparation*: prepares final results for presentation. Usually, retrieval result (raw result) contains a list of unique identifiers of retrieved documents, which are very likely meaningless to end users of an IR system. Therefore, raw results have to be transformed to a form that is more meaningful to human users, for example, showing original document titles, including a summary for each document, or displaying thumbnail images of documents.

- *evaluation*: assesses the effectiveness of retrieved results. Typical evaluation methods include *precision* and *recall*, in which precision measures the percentage of relevant documents that are retrieved, whereas recall measures the percentage of retrieved documents that are relevant. Several benchmarks have been proposed for evaluating retrieval effectiveness, for instance, TREC[1] for text retrieval and INEX[2] for XML (structured documents) retrieval.

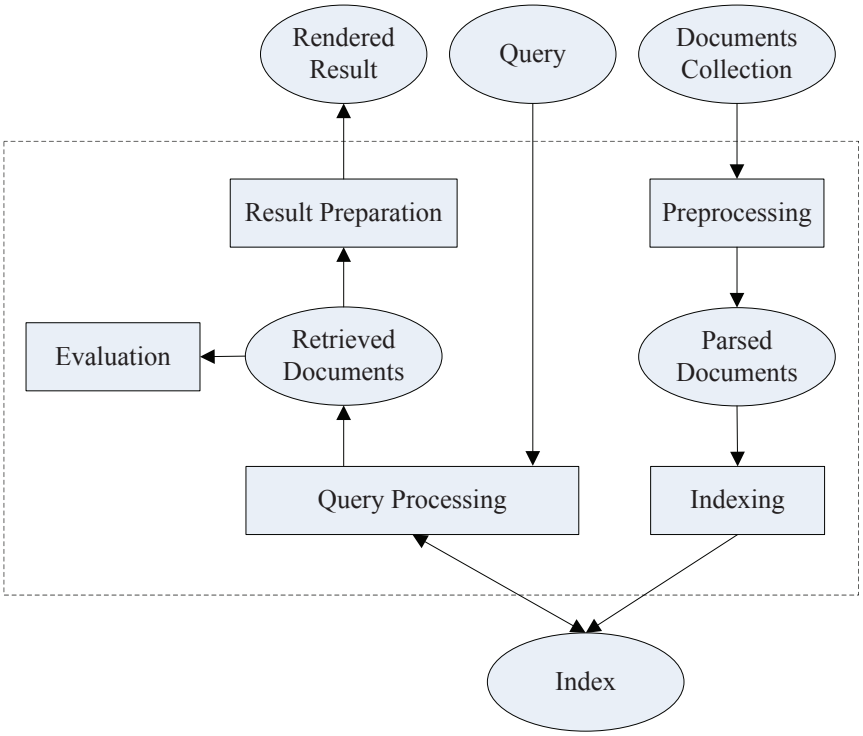


Figure 2.1: Basic procedures of Information Retrieval

In summary, the basic procedures could be assembled in an IR system as Figure 2.1.

[1] http://trec.nist.gov/
[2] http://inex.is.informatik.uni-duisburg.de/

Here we introduce an example of document collection for demonstration. A toy magazine corpus is given in Figure 2.2, in which contains two documents, one named 'fortune' and the other named 'time'. This toy corpus will be referred by most of later discussions where an example collection is needed.

FORTUNE
13 Test Drive
Hybrid wars heat up, as Honda pushes into the fray with the gas-electric Insight.
BY ALEX TAYLOR III

46 Bavaria's Next Top Model
With its new GT, BMW hopes to expand the definition of a luxury touring car.
But down the road it has to figure out what consumers want in a premium green automobile.
BY ALEX TAYLOR III

TIME
Hybrid vs. Hybrid: How the cars of the future compare
THE PRIUS - TOYOTA
The original hybrid uses both gas and electric engines to get the best fuel economy of any car in the U.S. today - and it costs less than the Volt's target price.
WHAT'S NEXT
Future versions will be plug-ins, but are unlikely to have the Volt's all-electric range.

THE VOLT - GENERAL MOTORS
The Volt is an extended-range electric vehicle: it's powered by electricity, with what amounts to a gasoline-fueled electric generator for longer drivers.
A QUESTION OF COST
Critics love the Volt technology - but they wonder if the car will be affordable.
BY BRYAN WALSH

Figure 2.2: A magazine corpus

Though structure is one of the inherent features of real-life documents, but in traditional text retrieval, documents are viewed as bag-of-words, where structures are usually ignored. Hence when the magazine corpus is to be preprocessed as a collection of non-structured documents, its documents are parsed into bag-of-words while structures information (if there are any) are discarded during the procedure. As aforementioned, several sub-processes could be employed besides parsing. For instance, terms are usually case-folded where upper case letters are replaced by lower case letters. In addition, stopwords such as "the" and "of" should be removed if they are not parts of phrases. Optionally, stemming could be applied to cut off semantic suffixes of words, e.g. change "cars" to "car". Moreover, hyphens of composed words could be removed as well, for example, breaking down "gas-electric" to "gas" and "electric". Furthermore, different abbreviations that refer to the same concept could be unified, for example, since both "U.S." and "U.S.A" refer to "United States of America" so that they would be replaced by "usa". In the end

of preprocessing, a possible output of bags-of-words may look like Figure 2.3.

#DocID=1
fortune 13 test drive hybrid war heat honda push fray gas electric insight alex taylor iii 46 bavarias next top model new gt bmw hope expand definition luxury touring car down road figure out consumer want premium green automobile alex taylor iii
#DocID=2
time hybrid hybrid car future compare prius toyota original hybrid use gas electric engine best fuel economy car usa today cost volt target price next future version plug in unlikely volt all electric range volt general motor volt extended range electric vehicle power electricity amount gasoline fuel electric generator longer driver question cost critic love volt technology wonder car affordable bryan walsh

Figure 2.3: A possible appearance of the magazine corpus after preprocessing as bags-of-words

On the other hand, structured document retrieval (e.g. XML retrieval) has also been well studied in IR. As Figure 2.2 shows, text are organised in groups such as paragraphs or sections or chapters, while some groups may include titles to indicate the contents of the groups, in which titles might be written in special formats such as highlighted or enlarged. In addition, references or bibliographies are common parts of documents, which appear in web pages as "anchor text + hyperlink". Figure 2.4 shows a possible appearance of the magazine corpus in XML format. Without doubts, structured documents are more informative than plain text documents, while the computational cost of structured document retrieval is also expected to be more expensive.

2.2.2 Retrieval Models

Retrieval model (or ranking model) is one of the foundations of IR and the most important building block of query processing. To review the dominant IR models, we distinguish mainly two classes: non-probabilistic and probabilistic models. On the non-probabilistic side, Vector Space Model (e.g. see [Baeza-Yates and Ribeiro-Neto, 1999b]), *tf-idf* and BM25 family (e.g. see [Robertson and Walker, 1994, Robertson et al., 2004]) are the dominant models; and on the probabilistic side, Binary Independent Retrieval (BIR) model and Language Modelling (LM) are the main candidates. Non-probabilistic models are mainly based on heuristics, whereas probabilistic models come with a theory background and some heuristics.

Probabilistic models date back to [Maron and Kuhns, 1960], which try to estimate the probability of a document being judged relevant to a particular query, this is denoted as the probability of relevance $P(d|q)$. Because there is no direct quantitative method to estimate the relevance probability, there are various methods to estimate the relevance probability. In late 1970s, [Robertson and Sparck Jones, 1976] pre-

```
<doc id="1" name="fortune">
 <title id="1">FORTUNE</title>
 <chapter id="1">
  <title id="1" font="bold">13 Test Drive</title>
  <section id="1">
   <para id="1">
    <link id="1" refDocId="2" refDocName="time">Hybrid wars</link>
    heat up, as Honda pushes into the fray with the gas-electric Insight.
   </para>
  </section>
  <author id="1" font="italic">BY ALEX TAYLOR III</author>
 </chapter>
 <chapter id="2">
  <title id="1" font="bold">46 Bavaria's Next Top Model</title>
  <section id="1">
   <para id="1">
    With its new GT, BMW hopes to expand the definition of a luxury touring car.
   </para>
   <para id="2">
    But down the road it has to figure out what consumers want in a premium green automobile.
   </para>
  </section>
  <author id="1" font="italic">BY ALEX TAYLOR III</author>
 </chapter>
</doc>
<doc id="2 name="time">
 <title id="1">TIME</title>
 <chapter id="1>
  <title id="1" font="bold">Hybrid vs. Hybrid: How the cars of the future compare</title>
  <section id="1">
   <title id="1" font="smallcaps">The Prius - Toyota</title>
   <para id="1">
    The original hybrid uses both gas and electric engines to get the best fuel economy of any car
    in the U.S. today - and it costs less than the Volt's target price.
   </para>
   <title id="2" font="smallcaps">What's Next</title>
   <para id="2">
    Future versions will be plug-ins, but are unlikely to have the Volt's all-electric range.
   </para>
  </section>
  <section id="2">
   <title id="1" font="bold">The Volt - General Motors</title>
   <para id="1">
    The Volt is an extended-range electric vehicle: it's powered by electricity, with what amounts
    to a gasoline-fueled electric generator for longer drivers.
   </para>
   <title id="2" font="smallcaps">A Question of Cost</title>
   <para id="2">
    Critics love the Volt technology - but they wonder if the car will be affordable.
   </para>
   <author id="1" font="italic">BY BRYAN WALSH</author>
  </section>
 </chapter>
</doc>
```

Figure 2.4: A possible appearance of the magazine corpus in XML format

sented BIRM. In the middle to end 1980s, [van Rijsbergen, 1986] initiated approaches to model IR as the probability $P(d \rightarrow q)$ of a non-classical implication between documents and queries. Early 1990s brought the inference network model [Turtle and Croft, 1990], Middle 1990s contributed the $P(d \rightarrow q)$ framework [Wong and Yao, 1995], and late 1990s to early 2000s brought LM (e.g. see [Ponte and Croft, 1998, Berger and Lafferty, 1999, Zhai and Lafferty, 2002, Lafferty and Zhai, 2003]) and Divergence from Randomness (DFR) [Amati and van Rijsbergen, 2002].

In probabilistic IR models, an important aspect is how to estimate the term weight with probability of relevance. Without relevant information we can estimate the term weight via *idf* (inverse document frequency), For examples, [Croft and Harper, 1979, Yu et al., 1982, Robertson, 1981] etc. have investigated *idf* heuristics against the probabilistic model.

More recently, [Hiemstra, 2000, Robertson, 2004, Roelleke and Wang, 2006] highlighted relationships between the three main classes of models: *tf-idf*, BIR, and LM. The work on relationships of models isolates the common components (probability estimations) in models that are the basic ingredients for modelling IR models.

2.2.2.1 Dominant Non-probabilistic Models

Following the convention in [Roelleke and Wang, 2008], we apply similar notions as they are shown in Table 2.1 to demonstrate the IR models.

$n_D(t)$	number of *documents* in a collection in which t occurs
N_D	number of *documents* in a collection
$n_L(t,x)$	number of *locations* in sequence x in which t occurs, if x is not given then it is the number of *locations* t occurs in a collection
$N_L(x)$	number of *locations* in sequence x in which t occurs, if x is not given then it is the number of *locations* in a collection

Table 2.1: Notions

Basic TF-IDF The basic *tf-idf* model defines as following:

$$RSV_{TF\text{-}IDF}(d,q) := \sum_{t \in d \cap q} tf(t,d) \cdot idf(t) \tag{2.1}$$

$$tf(t,d) := \frac{n_L(t,d)}{N_L(d)} \tag{2.2}$$

$$df(t) := \frac{n_D(t)}{N_D} \tag{2.3}$$

$$idf(t) \quad := \quad -\log \frac{n_D(t)}{N_D} \tag{2.4}$$

BM25 Family Traditional BM25 [Robertson and Walker, 1994] is a 2-Poisson based retrieval function. The relevance status value (RSV) score of a document calculated by BM25 is given as:

$$RSV_{BM25}(d,q) := \sum_{t \in d \cap q} \frac{n_L(t,d) \cdot (k_1+1)}{n_L(t,d) + k_1 \cdot \left((1-b) + b\frac{dl}{avdl}\right)} \cdot idf_{RSJ}(t) \tag{2.5}$$

$$idf_{RSJ}(t) = \log \frac{N_D - n_D(t) + 0.5}{n_D(t) + 0.5} \tag{2.6}$$

where $tf(t,d)$ is the within-document term count (raw frequency) of the term, k_1 and b are free parameters, dl is document length, $avdl$ is average document length across the collection, and $idf_{RSJ}(t)$ is the Robertson-Sparck-Jones idf weighting formula.

BM25F [Robertson et al., 2004] is an extension of the BM25, where a document can be modeled as having a number of fields, where different fields may be of different importance. For example, the title of a document may be one such field. A term occurring in the title field then can be given higher importance to if the term occurred in the body of the document.

In BM25F different weights w_i are assigned to the different fields (i.e., reflecting importance). Although the parameter k_1 may also be chosen specifically for the different fields, the study of [Robertson et al., 2004] has shown that field-specific b is more useful. The definition of BM25F is given by (the subscript f indicates field-specific variables):

$$RSV_{BM25F}(d,q) := \sum_{t \in f \cap q, f \subset d} \frac{\sum_f w_f \frac{n_L(t,f)}{B_f}}{k_1 + \sum_f w_f \frac{n_L(t,f)}{B_f}} \cdot idf_{RSJ}(t) \tag{2.7}$$

$$B_f = (1-b) + b \cdot \frac{fl}{avfl} \tag{2.8}$$

2.2.2.2 Dominant Probabilistic Models

Binary Independent Retrieval The BIR [Robertson and Sparck Jones, 1976] model is a theoretical pillar of probabilistic retrieval. The BIR defines the *RSV* as follows:

The BIR defines the *RSV* as follows:

$$RSV_{BIR}(d,q) := \sum_{t \in d \cap q} \left[\log \frac{P_D(t|q,r)}{P_D(\bar{t}|q,r)} - \log \frac{P_D(t|q,\bar{r})}{P_D(\bar{t}|q,\bar{r})} \right] \tag{2.9}$$

Language Modelling Language modelling linearly combines the probability $P_L(t)$ (probability that term t occurs in collections) and the probability $P_L(t|d)$ (probability that term t occurs in document d). These probabilities are estimated in the tuple space.

The *RSV* of LM is defined as follows:

$$P_L(t|d) := tf(t,d) \tag{2.10}$$

$$P_L(t) := \frac{n_L(t)}{N_L} \tag{2.11}$$

$$mixture = \lambda \cdot P_L(t|d) + (1-\lambda) \cdot P_L(t) \tag{2.12}$$

$$RSV_{LM}(d,q) := \sum_{t \in d \cap q} \log \left(1 + \frac{\lambda}{1-\lambda} \cdot \frac{P_L(t|d)}{P_L(t)} \right) \tag{2.13}$$

The mixture parameter λ is to be set: It can be term-dependent, query-dependent, or background-dependent.

2.3 Integrating Ranking into Relational Databases

As we have seen in the previous section, producing ranked result is an iconic characteristic of IR systems. On the other hand, though yielding ranked result was not a priority interest to relational databases at the first place, but this situation changed when applications for multimedia and decision-support emerged, in which internal support of processing top-k queries inside databases was deemed to be necessary and beneficial. Note that the researches on handling top-k queries within DB extensively impact later efforts of integrating ranking into RDBMS.

The theoretical studies on bring ranking into databases had been carried out since late 1980s, which aimed to establish a probabilistic framework for handling imprecise information and vague queries, for example, see [Cavallo and Pittarelli, 1987, Fuhr, 1990]. During mid 1990s, object-relational database (e.g. [Stonebraker and Moore, 1996]) systems became popular in databases research, while such systems (e.g. [Chaudhuri and Gravano, 1996, Fagin, 1996]) were used for managing multimedia data types such as text and images. In a typical multimedia application, a database usually sits at the back-end and manages storage and retrieval, whereas a middleware system was built upon database to provide actual functionality of the application. Since multimedia predicates often involve approximate matching, for example, measuring similarities of different shapes, colours or textures, which is logically similar to IR of estimating relevant documents to queries. As a result, such systems often need to answer fuzzy queries (e.g. [Fagin, 1996]) such

as “show me ten images in the database that look the most like an example”.

Around the same period, another applied area that caught many interest in DB community was decision-support and data warehousing system. Ranking and cardinality limits are commonly needed by business analysts (e.g. [Kimball and Strehlo, 1995]). In such systems, queries about, for example, the top $n\%$ of retailed commodities in terms of gross sales revenues, are often asked.

DB researchers were aware that databases at that time did not produce top-k results directly, which means databases always yielded full results and left the jobs of getting top-k answers to middleware. As a result, old databases wasted time on working out unwanted answers. Therefore, database's engine could stop, or in other words, early terminate, the processing when answering top-k queries if database could get notice of how many tuples are desired in the results.

The requirements provoked researches to implement new operators and algorithms (e.g. [Fagin, 1996, Carey and Kossmann, 1997, Carey and Kossmann, 1998]) into database's query engine, and to extend SQL (e.g. [Carey and Kossmann, 1997]) for supporting such extensions so that top-k queries could be expressed in declarative language. In particular, a family of top-k algorithms were proposed and caught a lot of attentions: Ronald Fagin introduced a *Fagin Algorithm* in [Fagin, 1996] in 1996, and then a well-known *Threshold Algorithm (TA)* (which is an extension of the *Fagin Algorithm*) was proposed respectively by Nepal and Ramakrishna [Nepal and Ramakrishna, 1999], Güntzer et al. [Güntzer et al., 2000], and Fagin et al. [Fagin et al., 2001]. More details about TA algorithm and related work about top-k processing will be addressed later in the Chapter 4.

The knowledge gained from fuzzy query processing on multimedia applications were soon spread and impact other applied domains of databases, in which ranking was found beneficial to solve some other problems of DB, such as the *empty answers* problem when queries are too selective, or the *many answers* problem while queries are not selective enough. In addition, ranking of query results is also helpful to applications where schema of databases are invisible to users (including application developers and casual end users), this might due to security restrictions while exposing database schema would lead to unauthorised information leak, or ordinary users should not be bothered to handle complicated schema. As a result, ranking of query results became desirable to keyword-based and schema-free search. Systems built for similar purposes could be found, e.g. in [Agrawal et al., 2002, Agrawal et al., 2003,

Hristidis and Papakonstantinou, 2002, Hristidis et al., 2003, Chaudhuri et al., 2004]. In their early works, though neither [Agrawal et al., 2002] nor [Hristidis and Papakonstantinou, 2002] discussed exactly how ranking is applied, however, the subsequent works of both studies adopt IR-like scoring functions. For instance, [Agrawal et al., 2003] introduced an *idf*-like ranking strategy called *QF Similarity*, which collaborates data frequencies with workload characteristic, while [Hristidis et al., 2003] employed BM25-style scoring strategy for ranking.

It is worth noting that along with the research of ranking for databases, DB researchers also made valuable contributions to rank-aware query optimization. Since new operators (including logical and physical) such as rank-join had been introduced with the emergence of top-k algorithms like TA, it had been noticed that query optimization techniques needed to be evolved for the new candidates. In consequence, various algebraic optimization methods as well as query scheduling had been proposed. For example, [Donjerkovic and Ramakrishnan, 1999] introduced probabilistic optimization of top-k queries with histogram, and [Ilyas et al., 2003] studied optimization for supporting top-k join queries, while [Ilyas et al., 2004] discussed rank-aware query optimization, and [Li et al., 2005] proposed a query algebra called RankSQL along with its algebraic optimization for relational top-k queries. On the studies of query scheduling, [Mutsuzaki et al., 2007] introduced top-k query evaluation with probabilistic guarantees which utilises probabilistic estimation of the lower-bound of candidate scores, while [Bast et al., 2006] IO-Top-k discussed the scheduling of IO access on candidate joined lists. More related works about query optimization will be discussed later in the Chapter 3.

In a word, despite the aforementioned researches aimed at studying generic ranking methods that are suitable for RDBMS, but no doubt their works are important progresses towards integrating DB and IR technologies.

2.4 Probabilistic Databases

Traditional relational databases base on a relational data model made by Codd [Codd, 1970], in which data are considered to be certain that they are either in or not in the databases. In other words, the relational data model supports a binary Boolean logic where the existence of data within databases could be only one of true or false. On the other hand, in order to handle uncertain data and vague or "fuzzy" information in relational databases, efforts had been carried out to extend the traditional relational data model.

Researches on probabilistic databases could date back as early as 1980s. As one of the earliest efforts, [Cavallo and Pittarelli, 1987] proposed a probabilistic data model based on intensional semantics, in which traditional relational model could be viewed as a generalisation of the probabilistic model, and reconstruction of relational algebra by considering dependent probabilities was discussed. In addition, [Bosc et al., 1988] proposed to extend relational and object oriented data models using fuzzy set and possibility theory. Furthermore, the notion of quality of databases and its estimation using a probabilistic approach was discussed in [Motro, 1988]. Moreover, [Fuhr, 1990] studied a probabilistic learning model for vague queries and imprecise information in databases, and [Fuhr, 1993, Fuhr and Rölleke, 1997] introduced a probabilistic relational algebra (PRA) extended traditional (deterministic) relational algebra by incorporating probabilities aggregation with relational operations. Note that the query evaluation of aforementioned works are based on intensional semantics of the possible worlds model of knowledge.

In recent, researches on the query evaluation based on extensional semantics have been carried out extensively. For instance, [Dalvi and Suciu, 2004, Re et al., 2007] introduced a *safe-plan* evaluation method for conjunctive query plans, which basically pushes probabilistic projection into join[3]. Alternatively, [Benjelloun et al., 2006a] introduced an evaluation technique employing auxiliary tables, which is so-called lineage that traces the elemental relations (contain only independent events) involved in complex relations (of complex events).

Various prototypes of probabilistic databases have been developed, for example, HySpirit (see e.g. [Fuhr and Roelleke, 1998, Fuhr et al., 1998, Rölleke et al., 2001]), MystiQ (see e.g. [Dalvi and Suciu, 2005, Boulos et al., 2005]), Trio (see e.g. [Benjelloun et al., 2006b, Mutsuzaki et al., 2007]), and MayBMS (see e.g. [Antova et al., 2007a, Antova et al., 2007b, Antova et al., 2007c]).

2.4.1 Possible Worlds Model

First of all, we informally describe the underlying model of probabilistic databases. In short, probability assignment in a probabilistic database complies with the *possible worlds* model (see e.g. [Cavallo and Pittarelli, 1987, Fagin and Halpern, 1994]), in which a relation R, i.e. including table and view, is considered to be a world s, and a tuple τ is viewed as an event e; while a probability p is assigned to τ when τ is in relation R, which indicates the possibility of event e

[3]As we will discuss in the later sections, the results obtained from this method is a way to reflect complex event probabilities, but the results do not present the actual complex events indicated by the intensional semantics.

occurs in world s. Furthermore, the probability of event e could be in the whole world Ω is the summation of probabilities of e in all possible worlds.

For example, assume we have a Car database which contains relations in the Table 2.2, where "eco-car" means "ecological car" or "environmental friendly car", and "HV" stands for "hybrid vehicle" [4], while "ICE" for "internal combustion engine", and "EM" for "electric motor".

CarCategory			
ID	$P(e)$	*Class*	*CarType*
a_1	0.7	Eco-car	HV

(a) CarCategory

CarPropulsionSystem			
ID	$P(e)$	*EngineType*	*VehicleType*
b_1	0.6	ICE	HV
b_2	0.8	EM	HV

(b) CarPropulsionSystem

Table 2.2: A Car database containing probabilistic relations

The *CarCategory* table is for car classification, while the *CarPropulsionSystem* table tells what type of engine could be used for propelling a certain type of vehicle. The schema of the tables are given as follows:

```
CarCartegoy(Class, CarType)
CarPropulsionSystem(EngineType, VehicleType)
```

In addition, a probability $P(e)$ is given to each tuple identified by tuple ID. For instance, $P(a_1) = 0.7$, $P(b_1) = 0.6$ and $P(b_2) = 0.8$, which might be explained as, e.g. 70% eco-cars are hybrid cars, and 60% hybrid cars have internal combustion engine, while 80% HVs installed electric motor.

The implied possible worlds of the Car database are illustrated in Figure 2.5, where it shows which worlds that each event may drop, while a corresponding possible worlds database is given in Figure 2.6. While assuming a world w_i consists of only *independent* events e_j, which is called elemental events, and the probability of the world is the conjunction of probabilities of all possible events. For instance, in the world w_2 where events a_1, b_1 and $\overline{b_2}$ occurs, i.e. $w_2 = a_1 \cap b_1 \cap \overline{b_2}$, so that the probability of the world is computed as $P(w_2) = 0.7 \cdot 0.6 \cdot (1 - 0.8) = 0.084$.

[4] E.g. see [Chan, 2002], "A hybrid vehicle is a vehicle that uses two or more distinct power sources to move the vehicle". Fuels for HVs include gasoline/diesel, gaseous fuels, biofeuls, synthetic fuels, hydrogen, in which different fuels could be mixed in ways that to be consumed by internal combustion engines (ICE), or the fuels are used to generate electricity, i.e. through electric generators or fuel-cell (an electrochemical conversion device), to power electric motors (EM). Other power sources for HV may include, for example, solar or compressed air, which could be either transformed into electricity for EMs or directly used by special propulsion systems. In particular, the "Hybrid Electric Vehicle" (HEV) (e.g. see [Gao et al., 2005]) is a type of HVs which is characterised by having both EM and ICE, in which the EM and ICE collaborate in certain powertrain to propel the car.

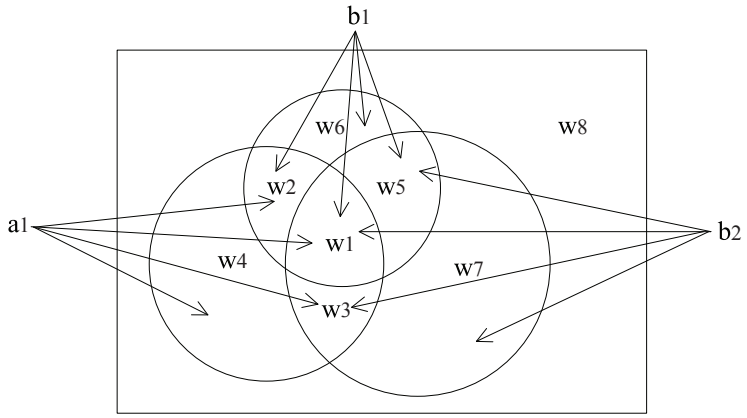


Figure 2.5: Possible worlds

Moreover, if an event is derived from two or more elemental events, for example, let event $e = a_1 \cap (b_1 \cup b_2)$, and let $P(e)$ be the probability of e in all worlds. Because $a_1 \cap (b_1 \cup b_2) = a_1 \cap \overline{(\overline{b_1} \cap \overline{b_2})}$, therefore $P(e) = P(a_1) \cdot (1 - P(\overline{b_1}) \cdot P(\overline{b_2})) = 0.7 \cdot (1 - 0.4 \cdot 0.2) = 0.644$. On the other hand, $P(e)$ could also be obtained by summing the probabilities of all worlds where e might occur. In this case, for e to occur then a_1 must occur and at least one of b_1 and b_2 must occur. In other words, the worlds that satisfy e could only be w_1, w_2 and w_3, hence we get $P(e) = P(w_1) + P(w_2) + P(w_3) = 0.644$, which is the same result as directly computing from the probabilities of elemental events.

possible world instance	*probability of world instance*
$w_1 = \{a_1, b_1, b_2\}$	$P(w_1) = 0.7 \cdot 0.6 \cdot 0.8 = 0.336$
$w_2 = \{a_1, b_1, \overline{b_2}\}$	$P(w_2) = 0.7 \cdot 0.6 \cdot (1-0.8) = 0.084$
$w_3 = \{a_1, \overline{b_1}, b_2\}$	$P(w_3) = 0.7 \cdot (1-0.6) \cdot 0.8 = 0.224$
$w_4 = \{a_1, \overline{b_1}, \overline{b_2}\}$	$P(w_4) = 0.7 \cdot (1-0.6) \cdot (1-0.8) = 0.056$
$w_5 = \{\overline{a_1}, b_1, b_2\}$	$P(w_5) = (1-0.7) \cdot 0.6 \cdot 0.8 = 0.144$
$w_6 = \{\overline{a_1}, b_1, \overline{b_2}\}$	$P(w_6) = (1-0.7) \cdot 0.6 \cdot (1-0.8) = 0.036$
$w_7 = \{\overline{a_1}, \overline{b_1}, b_2\}$	$P(w_7) = (1-0.7) \cdot (1-0.6) \cdot 0.8 = 0.096$
$w_8 = \{\overline{a_1}, \overline{b_1}, \overline{b_2}\}$	$P(w_8) = (1-0.7) \cdot (1-0.6) \cdot (1-0.8) = 0.024$
$\mathcal{W} = \bigcup_{i=1}^{8} w_i$	$\sum_{i=1}^{8} P(w_i) = 1$

Figure 2.6: An example of a probabilistic database based on possible worlds (intensional semantics)

Now we define probabilistic database formally.

Definition 2.4.1. *Probabilistic Database*. A probabilistic database $\mathcal{D}$ is a set of relations that $\mathcal{D} = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$. A relation $\mathcal{R}$ is a quadruple that $\mathcal{R} = (\mathbf{X}, \Delta, \mathbf{f}, \mathbf{p})$, where X is the schema of $\mathcal{R}$ which is a non-empty set of distinct symbols called attributes, for a relation $\mathcal{R}$ having schema X

is denoted as $\mathcal{R}_{\mathrm{X}}$; and Δ is a set of tuples $\{\tau_1, \ldots, \tau_n\}$ called domain, where $\forall \tau \in \Delta, \tau \mapsto \mathrm{X}$; while f is a function $\mathrm{f}(\mathcal{R}'_{\mathrm{Y}}) \rightarrow \mathcal{R}_{\mathrm{X}}$ that yields relation $\mathcal{R}$; and p is an event expression that computes the event probability $P(e = \tau | \tau \in \Delta)$ where $P(e) \rightarrow [0, 1]$.

Here a relation in Definition 2.4.1 is similar to a traditional viewpoint in non-probabilistic databases where tuples are considered as mappings from attributes' names to values in the domains of the attributes; on the other hand, it extends conventional definition by introducing event expression p, which computes event probabilities and assigns the probabilities to tuples; in other words, a probabilistic weighting is attached to each of the mappings from names to values.

Definition 2.4.2. *Elemental Relation.* An elemental relation in a probabilistic database is a relation that contains only independent events; furthermore, elemental relations are independent of each others.

Definition 2.4.3. *Complex Relation.* A complex relation in a probabilistic database is a relation that is derived from one or several other elemental or complex relations through certain combination of operations of Cartesian product, projection and selection.

Given the above definitions, as a result, traditional (deterministic) relational databases can be viewed as a specialisation of probabilistic database, where the event expression of any relation always computes $P(e) = 1$ and assigns probability one to tuples. Next, the possible worlds model is defined as follow:

Definition 2.4.4. *Possible Worlds.* Let e be a probabilistic event, given an event space $\mathcal{E} = \{e_1, \ldots, e_n\}$ in which all events are independent, i.e. $\forall e_i, e_j \in \mathcal{E}$ where $i \neq j$, there is $P(e_j | e_i) = P(e_j) \cdot P(e_i)$; the possible worlds $\mathcal{W}$ is a set of instances of worlds that $\mathcal{W} = \{w_1, \ldots, w_m | m = 2^n\}$, in which a world instance w consists of conjunctive events that $w = \bigcap_{i=1}^{n} e_i$, and the probability of a world instance $P(w) = \prod_{i=1}^{n} e_i$, where $e_i \mapsto e_k \underline{\vee} \overline{e_k}$, $k = i$ and $e_k \in \mathcal{E}$. The world instances are disjoint events of each others, the summation of the probabilities of all world instances is 1, i.e. $\sum_{i=1}^{m} P(w_i) = 1$.

Furthermore, the relationship between probabilistic database and possible worlds model is given by Theorem 2.4.1.

Theorem 2.4.1. *Let $\mathcal{D}$ be a probabilistic database and $\mathcal{W}$ be the implied possible worlds of $\mathcal{D}$, $\forall \mathcal{R} \in \mathcal{D}$, the event probability $P(e = \tau | \tau \in \mathcal{R})$ is equivalent to the summation of probabilities of*

a set of implied world instances $\{w\} \subset \mathcal{W}$, *i.e.* $P(e = \tau | \tau \in \mathcal{R}) = \sum_{i=1}^{k} P(w_i)$, *where* $\forall w \in \{w\}$ $\tau \vdash w$.

Proof. Assume $\mathcal{E} = \{e_1, \ldots, e_n\}$ is an event space and $\mathcal{W} = \{w_1, \ldots, w_m\}$ is the implied possible worlds of $\mathcal{E}$ where $m = 2^n$. For a complex event $e' = (e_i \cap e_j) \cup (e_i \cap e_k) = e_i \cap (e_j \cup e_k)$, the implied world instances are $\{w\} = \{\{e_i, e_j, e_k\}, \{e_i, e_j, \overline{e_k}\}, \{e_i, \overline{e_j}, e_k\}\}$, while

$$\begin{aligned}
\bigcup_{\forall w \in \{w\}} w &\Leftrightarrow e_i e_j e_k \cup e_i e_j \overline{e_k} \cup e_i \overline{e_j} e_k \\
&\Leftrightarrow e_i \cap (e_j e_k \cup e_j \overline{e_k} \cup \overline{e_j} e_k) \\
&\Leftrightarrow e_i \cap (e_j \cap (e_k \cup \overline{e_k}) \cup \overline{e_j} e_k) \\
&\Leftrightarrow e_i \cap (e_j \cup \overline{e_j} e_k) \\
&\Leftrightarrow e_i \cap (e_j \cup e_k) \\
&\Leftrightarrow e'
\end{aligned}$$

The above equivalence always holds for any conjunctive complex event where $\{e\} \subseteq \mathcal{E}$. Thus the statement of Theorem 2.4.1 is necessary and sufficient, and therefore Theorem 2.4.1 is sound. Proved. □

2.4.2 Probabilistic Relational Algebra

In [Cavallo and Pittarelli, 1987], the probabilistic database is defined only based on intensional semantics (i.e. the possible worlds model), in other words, there are not complex relations within the database. Different from [Cavallo and Pittarelli, 1987], [Fuhr, 1990, Fuhr, 1993] defined probabilistic database based on extensional semantics, in which the tuples of tables (i.e. materialised relations) are viewed as independent events; whereas the query evaluation on such database is based on intensional semantics, which complies to the possible worlds model. In a subsequent work of [Fuhr, 1993], a probabilistic relational algebra (PRA) was proposed in [Fuhr and Rölleke, 1997], which integrates *probability aggregations* with relational algebra, where five basic operators $(\sigma, \Pi, \bowtie, \cup, -)$ for selection, projection, join, union, and difference respectively were redefined.

Based on [Fuhr and Rölleke, 1997] and the probabilistic database defined in Definition 2.4.1, the formal definitions of the five basic operators of PRA are given as follows.

Definition 2.4.5. Selection. Given a relation $\mathcal{R}'_X \in \mathcal{D}$ and a set of predicates $\Theta = \{=, \neq, <, \leq, >, \geq, \approx\}$, the *selection* operator σ is defined as:

$$\begin{aligned}&\sigma_{\mu:X\Theta x}(\mathcal{R}'_X) \rightarrow \mathcal{R}_X \\ &\Rightarrow \text{if } \exists\tau \in \mathcal{R}'_X \text{ that } \mu(\tau) \text{ is true, then } \sigma_{\mu:X\Theta x}(\tau) \mapsto \tau \text{ where } \tau \in \mathcal{R}_X\end{aligned} \tag{2.14}$$

and the event probability is given by:

$$P(e = \tau | \tau \in \mathcal{R}_X) \quad := \quad P(e = \tau | \tau \in \mathcal{R}'_X) \tag{2.15}$$

For instance, taking an example from the Car database in Table 2.2, a PRA expression as $\sigma_{EngineType='ICE'}(CarPropulsionSystem)$ yields:

$\sigma_{EngineType='ICE'}(CarPropulsionSystem)$		
$P(e)$	*EngineType*	*VehicleType*
0.6	ICE	HV

Definition 2.4.6. Projection. Given a relation $\mathcal{R}'_X \in \mathcal{D}$ and $\overline{X}$ is the algebraic closure of X, the *projection* operator Π is defined as:

$$\begin{aligned}&\Pi_{\overline{X}}(\mathcal{R}'_X) \rightarrow \mathcal{R}_{\overline{X}} \\ &\Rightarrow \text{if } \exists\{\tau'\} \subseteq \mathcal{R}'_X \text{ and } \forall\tau' \in \{\tau'\} \text{ that } [\tau'_{\overline{X}}] = \tau, \text{ then } \Pi_{\overline{X}}(\{\tau'\}) \mapsto \tau \text{ where } \tau \in \mathcal{R}_{\overline{X}}\end{aligned} \tag{2.16}$$

and the event probability is given by:

$$\begin{aligned}P(e = \tau | \tau \in \mathcal{R}_{\overline{X}}) \quad &:= \quad \bigcup_{\forall\tau':[\tau'_{\overline{X}}]=\tau} P(e = \tau' | \tau' \in \mathcal{R}'_X) \\ &\Rightarrow \quad 1 - \prod_{\forall\tau':[\tau'_{\overline{X}}]=\tau} (1 - P(e = \tau' | \tau' \in \mathcal{R}'_X))\end{aligned} \tag{2.17}$$

Note that the projection also does duplicate removal for results, in other words, the tuples in the results must be distinct. Similarly, an example PRA expression of projection as $\Pi_{VehicleType}(CarPropulsionSystem)$ yields:

$\Pi_{VehicleType}(CarPropulsionSystem)$	
$P(e)$	*VehicleType*
0.92	HV

Definition 2.4.7. Natural Join. Given two relations $\mathcal{A}_X, \mathcal{B}_Y \in \mathcal{D}$ and a set of predicates $\Theta = \{=, \neq, <, \leq, >, \geq, \approx\}$, while $\overline{X}$ and $\overline{Y}$ are the algebraic closures of X and Y respectively, the *natural join* operator $\bowtie$ is defined as:

$$
\begin{aligned}
&\mathcal{A}_{\mathrm{X}} \bowtie_{\mu:\overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}} \mathcal{B}_{\mathrm{Y}} \rightarrow \mathcal{R}_{\mathrm{Z}}, \text{ where } \mathrm{Z} = \mathrm{X} \cup \mathrm{Y} = (\mathrm{X} + \mathrm{Y}) \\
&\Rightarrow \mathcal{A}_{\mathrm{X}} \times \sigma_{\mu:\overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}}(\mathcal{B}_{\mathrm{Y}}) \rightarrow \mathcal{R}_{\mathrm{Z}} \\
&\Rightarrow \text{if } \exists \tau_a \in \mathcal{A}_{\mathrm{X}}\ \exists \{\tau_b\} \subseteq \mathcal{B}_{\mathrm{Y}} \text{ and } \forall \tau_b \in \{\tau_b\} \text{ that } \mu(\tau_a, \tau_b) = [\tau_{a,\overline{\mathrm{X}}}]\Theta[\tau_{b,\overline{\mathrm{Y}}}] \text{ is true,} \\
&\quad \text{then } \tau_a \times \{\tau_b\} \mapsto \tau \text{ where } \tau \in \mathcal{R}_{\mathrm{Z}}
\end{aligned}
\tag{2.18}
$$

Here the schema of join result is defined by a recursive regular expression $\mathrm{Z} = \mathrm{X} \cup \mathrm{Y}$ (denoted as $\mathrm{X} + \mathrm{Y}$), where concatenated attributes $\mathrm{X} + \mathrm{Y}$ must occur at least once. In addition, the event probability is given by:

$$
\begin{aligned}
P(e = \tau | \tau \in \mathcal{R}_{\mathrm{Z}}) \quad &:= \quad \bigcup_{\exists \tau_a \exists \{\tau_b\}:\mu(\tau_a,\tau_b)} \Big(P(e = \tau_a | \tau_a \in \mathcal{A}_{\mathrm{X}}) \bigcap P(e = \tau_b | \tau_b \in \mathcal{B}_{\mathrm{Y}}) \Big) \\
&\Rightarrow \quad P(e = \tau_a | \tau_a \in \mathcal{A}_{\mathrm{X}}) \bigcap \left(\bigcup_{\exists \tau_a \exists \{\tau_b\}:\mu(\tau_a,\tau_b)} P(e = \tau_b | \tau_b \in \mathcal{B}_{\mathrm{Y}}) \right) \\
&\Rightarrow \quad P(e = \tau_a | \tau_a \in \mathcal{A}_{\mathrm{X}}) \cdot \left(1 - \prod_{\exists \tau_a \exists \{\tau_b\}:\mu(\tau_a,\tau_b)} (1 - P(e = \tau_b | \tau_b \in \mathcal{B}_{\mathrm{Y}})) \right)
\end{aligned}
\tag{2.19}
$$

The join operator of PRA performs an algebraic manipulation that often causes confusions. Because join's definition is based on intensional semantics, according to formula 2.18 in which the output of the product between τ_a and set $\{\tau_b\}$ would be mapped to a distinct tuple τ that represents an *complex event*, which is invisible from an extensional semantics point of view, where only elemental relations that contain independent events could be presented. However, we may "peek" at the results by employing a traditional RA projection Π' that only does duplicate removal (i.e. does not do aggregation) after join. By using this method, the event probabilities of tuples obtained by join can be displayed under the extensional semantics, but be aware that a result set displayed in such way does not represent the *actual* complex events under the intensional semantics. For example, to obtain the actual complex events with join we can write a PRA expression $Actual = CarCategory \bowtie_{CarType=VehicleType} CarPropulsionSystem$, and to "peek" at the computed event probability we can write $\Pi'_{Class}(Actual)$. Both results are illustrated in the following tables.

Definition 2.4.8. Union. Given two relations $\mathcal{A}_{\mathrm{X}}, \mathcal{B}_{\mathrm{Y}} \in \mathcal{D}$ where $[X] = [Y]$, the *union* operator $\cup$ is defined as:

$CarCategory \bowtie_{CarType=VehicleType} CarPropulsionSystem$	
$P(e)$	$(Class, CarType, EngineType, VehicleType)$
0.644	((Eco-car, HV) ∩ (ICE, HV)) ∪ ((Eco-car, HV) ∩ (EM, HV))

(a) Actual

$\Pi'_{Class}(Actual)$	
$P(e)$	$Class$
0.644	Eco-car

(b) Peek

$$\begin{aligned}
&\mathcal{A}_X \cup \mathcal{B}_Y \rightarrow \mathcal{R}_X \\
&\Rightarrow \text{if } \exists\{\tau_a\} \subseteq \mathcal{A}_X \; \exists\{\tau_b\} \subseteq \mathcal{B}_Y \text{ that } [\{\tau_a\}] \neq [\{\tau_b\}], \text{ then } \{\tau_a\} \cup \{\tau_b\} \rightarrow \{\tau\} = \{\{\tau_a\}, \{\tau_b\}\}, \\
&\quad \text{else if } \exists\{\tau_a\} \subseteq \mathcal{A}_X \; \exists\{\tau_b\} \subseteq \mathcal{B}_Y \text{ that } [\{\tau_a\}] = [\{\tau_b\}], \text{ then } \{\tau_a\} \cup \{\tau_b\} \rightarrow \{\tau'\} = \{\tau_a\}, \\
&\quad \text{where } \{\tau\} \cap \{\tau'\} = \emptyset \text{ and } \{\tau\} \cup \{\tau'\} = \mathcal{R}_X
\end{aligned} \tag{2.20}$$

and the event probability is given by:

$$\begin{aligned}
P(e = \tau | \tau \in \mathcal{R}_X) &:= P(e = \tau | \tau \in \mathcal{A}_X) \bigcup P(e = \tau | \tau \in \mathcal{B}_Y) \\
&\Rightarrow 1 - (1 - P(e = \tau | \tau \in \mathcal{A}_X)) \cdot (1 - P(e = \tau | \tau \in \mathcal{B}_Y)) \\
&\Rightarrow \begin{cases} P(e = \tau | \tau \in \mathcal{A}_X) & \text{if } \tau \in \mathcal{A}_X \cap \overline{\mathcal{B}_Y} \\ P(e = \tau | \tau \in \mathcal{B}_Y) & \text{if } \tau \in \overline{\mathcal{A}_X} \cap \mathcal{B}_Y \\ 1 - (1 - P(e = \tau | \tau \in \mathcal{A}_X)) \cdot (1 - P(e = \tau | \tau \in \mathcal{B}_Y)) & \text{if } \tau \in \mathcal{A}_X \cap \mathcal{B}_Y \end{cases}
\end{aligned} \tag{2.21}$$

The union operation should be applied when we need to combine relations (from the same database or other databases) of additional knowledge. For example, assuming there is another table *CarClassification* which has a schema that is equivalent to *CarCategory*:

$CarClassification$		
$P(e)$	$SuperClass$	$SubClass$
0.4	Eco-car	HV
0.6	Eco-car	EV

To combine the two tables, we use union $CarCategory \cup CarClassification$, and the expression yields:

$CarCategory \cup CarClassification$		
$P(e)$	$Class$	$CarType$
0.82	Eco-car	HV
0.6	Eco-car	EV

Definition 2.4.9. Difference. Given two relations $\mathcal{A}_X, \mathcal{B}_Y \in \mathcal{D}$ where $[X] = [Y]$, the *difference* operator $-$ is defined as:

$$\begin{aligned}
&\mathcal{A}_{\mathrm{X}} - \mathcal{B}_{\mathrm{Y}} \rightarrow \mathcal{R}_{\mathrm{X}} \\
&\Rightarrow \text{if } \exists\{\tau_a\} \subseteq \mathcal{A}_{\mathrm{X}}\ \exists\{\tau_b\} \subseteq \mathcal{B}_{\mathrm{Y}} \text{ that } [\{\tau_a\}] = [\{\tau_b\}], \text{ then } \mathcal{A}_{\mathrm{X}} - \mathcal{B}_{\mathrm{Y}} \rightarrow \{\tau\} = \mathcal{R}_{\mathrm{X}}, \\
&\quad \text{where } \{\tau\} \subseteq \mathcal{A}_{\mathrm{X}} \text{ and } \{\tau\} \cap \{\tau_a\} = \{\tau'\} \text{ and } \forall \tau' \in \{\tau'\}\ P(e = \tau' | \tau' \in \mathcal{R}_{\mathrm{X}}) > 0
\end{aligned} \tag{2.22}$$

and the event probability is given by:

$$\begin{aligned}
P(e = \tau | \tau \in \mathcal{R}_{\mathrm{X}}) \quad &:= \quad P(e = \tau | \tau \in \mathcal{A}_{\mathrm{X}}) \bigcap \overline{P(e = \tau | \tau \in \mathcal{B}_{\mathrm{Y}})} \\
&\Rightarrow \quad P(e = \tau | \tau \in \mathcal{A}_{\mathrm{X}}) \cdot (1 - P(e = \tau | \tau \in \mathcal{B}_{\mathrm{Y}})) \\
&\Rightarrow \quad \begin{cases} P(e = \tau | \tau \in \mathcal{A}_{\mathrm{X}}) & \text{if } \tau \in \mathcal{A}_{\mathrm{X}} \cap \overline{\mathcal{B}_{\mathrm{Y}}} \\ P(e = \tau | \tau \in \mathcal{A}_{\mathrm{X}}) \cdot (1 - P(e = \tau | \tau \in \mathcal{B}_{\mathrm{Y}})) & \text{if } \tau \in \mathcal{A}_{\mathrm{X}} \cap \mathcal{B}_{\mathrm{Y}} \end{cases}
\end{aligned} \tag{2.23}$$

In contrast to union, the difference operator of PRA removes or degrades knowledge from a relation with respect to other relations. The operator differs from a traditional (deterministic) relational algebra in a way that it does not eliminate a tuple from the first relation (i.e. the left operand of $-$) unless the tuple's event probability is downgraded to zero, whereas a traditional RA would delete the tuple if it finds a match from the second relation. For instance, a PRA expression $CarClassification - CarCategory$ yields:

$CarClassification - CarCategory$		
$P(e)$	*SuperClass*	*SubClass*
0.12	Eco-car	HV
0.6	Eco-car	EV

2.4.3 Query Evaluation Techniques for Conjunctive Queries

Here we compare varied query evaluation techniques on probabilistic databases. In general, an evaluation method could be categorised by whether it is based on intensional semantics or based on extensional semantics.

[Fuhr and Rölleke, 1997] discussed an evaluation technique based on intensional semantics, where it shows for the PRA expressions that do not involve conjunctive queries could be evaluated by means of *simple evaluation*, which is equivalent to extensional semantics and has the same computational complexity as traditional RA evaluation; whereas for conjunctive queries, PRA performs equivalently to deduct the implied possible worlds of the probabilistic database, which guarantees correct probabilities could be obtained but it has to be more expensive. The way to construct possible worlds has been demonstrated in the previous section 2.4.1.

On the other hand, [Dalvi and Suciu, 2004] proposed a method that is fully based on extensional semantics. In order to compute correct probabilities for conjunctive queries, it rewrites query plan by projecting out dependent attributes of joined relations, in which a *Safe-Plan* algorithm is performed respectively to the first (i.e. left-hand-side) and second (i.e. right-hand-side) relations of join operands.

The *Safe-Plan* method introduced in [Dalvi and Suciu, 2004] could be interpreted[5] as follows. Let $\mathcal{R}_{\mathrm{X}}$ and $\mathcal{S}_{\mathrm{Y}}$ be two relations, for conjunctive queries in a form of $\Pi_{\overline{\mathrm{X}+\mathrm{Y}}}(\mathcal{R}_{\mathrm{X}} \bowtie_{\overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}} \mathcal{S}_{\mathrm{Y}})$. Let $\mathrm{Z} = \overline{\mathrm{X}+\mathrm{Y}}$, then to find a safe plan for the type of conjunctive queries is to find an equivalent query plan $\Pi_{\mathrm{Z}}(\mathcal{R}'_{\overline{\mathrm{X}}} \bowtie_{\overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}} \mathcal{S}'_{\overline{\mathrm{Y}}})$ that satisfies:

$$\begin{aligned} \mathrm{Z}, \mathcal{R}'_{\overline{\mathrm{X}}}, \overline{\mathrm{X}}\Theta\overline{\mathrm{Y}} &\rightarrow \{A_1, \ldots, A_n\} \\ \mathrm{Z}, \mathcal{S}'_{\overline{\mathrm{Y}}}, \overline{\mathrm{X}}\Theta\overline{\mathrm{Y}} &\rightarrow \{A_1, \ldots, A_m\} \\ [\{A_1, \ldots, A_n\}] &= [\{A_1, \ldots, A_m\}] \end{aligned} \tag{2.24}$$

To explain, now assume $\mathrm{X} = (A, B, C)$ and $\mathrm{Y} = (G, H)$, then the following inferences can be obtained:

$$\begin{aligned} \mathcal{R}.A, \mathcal{R}.B, \mathcal{R}.C &\rightarrow \mathcal{R}_{\mathrm{X}} \\ \mathcal{S}.G, \mathcal{S}.H &\rightarrow \mathcal{S}_{\mathrm{Y}} \\ \mathcal{R}_{\mathrm{X}} &\rightarrow \mathcal{R}.A, \mathcal{R}.B, \mathcal{R}.C \\ \mathcal{S}_{\mathrm{Y}} &\rightarrow \mathcal{S}.G, \mathcal{S}.H \end{aligned}$$

Given a conjunctive query plan $\Pi_A(\mathcal{R}_{\mathrm{X}} \bowtie_{C=H} \mathcal{S}_{\mathrm{Y}})$, we can examine if the plan is a safe plan according to the formula 2.24, and we obtain:

$$\begin{aligned} \Pi_A(\mathcal{R}_{\mathrm{X}} \bowtie_{C=H} \mathcal{S}_{\mathrm{Y}}) &\rightarrow (\mathcal{R}.A, \mathcal{R}_{\mathrm{X}}, C = H) \cup (\mathcal{R}.A, \mathcal{S}_{\mathrm{Y}}, C = H) \\ \mathcal{R}.A, \mathcal{R}_{\mathrm{X}}, C = H &\rightarrow \mathcal{R}.A, \mathcal{R}.B, \mathcal{R}.C, \mathcal{S}.H \\ \mathcal{R}.A, \mathcal{S}_{\mathrm{Y}}, C = H &\rightarrow \mathcal{R}.A, \mathcal{R}.C, \mathcal{S}.G, \mathcal{S}.H \\ [\{\mathcal{R}.A, \mathcal{R}.B, \mathcal{R}.C, \mathcal{S}.H\}] &\neq [\{\mathcal{R}.A, \mathcal{R}.C, \mathcal{S}.G, \mathcal{S}.H\}] \end{aligned}$$

Since there are no equivalent sets of attributes could be deducted from the two operands of joins, hence the original query plan is not safe. Because the attribute of the outer projection belongs to $\mathcal{R}_{\mathrm{X}}$, so that try to project *out* independent attributes (in other words, project *on* de-

[5]The original *Safe-Plan* algorithm is to look for a split of attributes of original query that forms safe sub-plans, and then the algorithm is performed recursively to transform the entire plan into safe plan. Alternatively, we noticed the method could be also explained as to deduct equivalent attribute sets of join operands, where independent attributes are directly projected out, which might seem to be more intuitive to demonstrate the dependencies of attributes during query plan rewriting.

pendent attribute H) from $\mathcal{S}_{\mathrm{Y}}$, i.e. let ${\mathcal{S}'}_{\overline{\mathrm{Y}}} = \Pi_H(\mathcal{S}_{\mathrm{Y}})$, and a substituted plan could be obtained as $\Pi_A(\mathcal{R}_{\mathrm{X}} \bowtie_{C=H} {\mathcal{S}'}_{\overline{\mathrm{Y}}})$. Now let us test it again to see if a safe plan has been achieved:

$$\begin{array}{rcl}
\Pi_A(\mathcal{R}_{\mathrm{X}} \bowtie_{C=H} {\mathcal{S}'}_{\overline{\mathrm{Y}}}) & \rightarrow & (\mathcal{R}.A, \mathcal{R}_{\mathrm{X}}, C=H) \cup (\mathcal{R}.A, {\mathcal{S}'}_{\overline{\mathrm{Y}}}, C=H) \\
\mathcal{R}.A, \mathcal{R}_{\mathrm{X}}, C=H & \rightarrow & \mathcal{R}.A, \mathcal{R}.B, \mathcal{R}.C, \mathcal{S}'.H \\
\mathcal{R}.A, {\mathcal{S}'}_{\overline{\mathrm{Y}}}, C=H & \rightarrow & \mathcal{R}.A, \mathcal{R}.C, \mathcal{S}'.H \\
[\{\mathcal{R}.A, \mathcal{R}.B, \mathcal{R}.C, \mathcal{S}'.H\}] & \neq & [\{\mathcal{R}.A, \mathcal{R}.C, \mathcal{S}'.H\}]
\end{array}$$

Again, the obtained attribute sets are not equivalent, which is because there is still an independent attribute $\mathcal{R}.B$ from $\mathcal{R}_{\mathrm{X}}$. So let us project out the independent attributes of both relations, and let ${\mathcal{R}'}_{\overline{\mathrm{X}}} = \Pi_{A,C}(\mathcal{R}_{\mathrm{X}})$, ${\mathcal{S}'}_{\overline{\mathrm{Y}}} = \Pi_H(\mathcal{S}_{\mathrm{Y}})$, then we got $\Pi_A({\mathcal{R}'}_{\overline{\mathrm{X}}} \bowtie_{C=H} {\mathcal{S}'}_{\overline{\mathrm{Y}}})$, and re-examine the plan:

$$\begin{array}{rcl}
\Pi_A({\mathcal{R}'}_{\overline{\mathrm{X}}} \bowtie_{C=H} {\mathcal{S}'}_{\overline{\mathrm{Y}}}) & \rightarrow & (\mathcal{R}'.A, {\mathcal{R}'}_{\overline{\mathrm{X}}}, C=H) \cup (\mathcal{R}'.A, {\mathcal{S}'}_{\overline{\mathrm{Y}}}, C=H) \\
\mathcal{R}'.A, {\mathcal{R}'}_{\overline{\mathrm{X}}}, C=H & \rightarrow & \mathcal{R}'.A, \mathcal{R}'.C, \mathcal{S}'.H \\
\mathcal{R}'.A, {\mathcal{S}'}_{\overline{\mathrm{Y}}}, C=H & \rightarrow & \mathcal{R}'.A, \mathcal{R}'.C, \mathcal{S}'.H \\
[\{\mathcal{R}'.A, \mathcal{R}'.C, \mathcal{S}'.H\}] & = & [\{\mathcal{R}'.A, \mathcal{R}'.C, \mathcal{S}'.H\}]
\end{array}$$

Finally, the obtained attribute sets become equivalent, and the original plan is to be replaced by the rewritten safe plan.

In fact, the key of the *Safe-Plan* is to look for dependent attributes in conjunctive queries, in which independent attributes should be projected out, so that tuples (i.e. probabilistic events) could be aggregated before to be involved into complex correlation such as join. Note that the dependencies of attributes are indicated by join predicates, i.e. the operands of Θ are indeed dependent attributes, which could be easily found out so that early aggregations could be performed.

Moreover, another alternative evaluation technique that employing data lineage has been introduced in [Benjelloun et al., 2006a]. This method applies auxiliary tables called lineage of data, which is used for tracing the origins (elemental relations) of complex relations. With regard to evaluating conjunctive queries, it complies to the intensional semantics but computes probabilities in a smart way, which has been demonstrated in Definition 2.4.7 of join in the equation 2.19 at the last implication.

2.5 Integrated IR and DB Technologies

Intuitively, an integrated IR and DB technology is desirable because modern applications require capabilities to handle both text and structured data, and neither state-of-the-art search engines nor traditional databases were developed towards this goal. On the one hand, most (if there is a few exceptions) IR search engines are developed as problem-oriented or application-specific systems which implement (hard-coded) certain ranking models inside. Since very few of them consider flexibility and high level query optimization as necessary features, hence it is unlikely to adapt such systems for different applications without significant engineering efforts that could be as much as implementing brand new systems. On the other hand, databases have been evolved as generic information management systems that provide customisable and scalable solutions for wide range of applications, however, conventional RDBMS lacks built-in text engine that could handle text retrieval efficiently, which makes IR applications built upon DBs unlikely to be scalable. Similar observation and analysis have been discussed in [Chaudhuri et al., 2005].

In general, an integrated IR and DB technology or system is found attractive in the following aspects:

- Relatively short development circles for diverse IR applications requiring text and structured data handling capability.
- Flexible to combine multi-origin data sources for probabilistic inference including induction and deduction.
- Customisability for scoring and ranking functions and supporting high level query optimization methods.

Above all, we shortly review the state of the art of integrated DB and IR technologies.

2.5.1 State-of-the-Art

One of the most fundamental efforts can be found in [Grossman et al., 1997, Grossman and Frieder, 2004]. In this work, they utilise the classical relational model to achieve high level integration of structured data and text using strictly unchanged standard SQL, to perform keyword searches; specifically, they implement relevance ranking models for document retrieval such as Boolean retrieval, proximity searches and vector space model. To be noticed, the work discussed, on the one hand, several benefits of implementing IR applications

upon DB engines, for instance, shortening development circles and combining multiple sources (as we have mentioned), and save costs and risks for developing brand new information systems; on the other hand, it was aware that a high levelled integrated DB and IR approach inevitably suffers efficiency drawback compared to specially built IR systems, however, they argued that this disadvantage could be overcome by parallel processing that was widely available in commercial databases.

Similarly, [Grabs et al., 2001, Grabs et al., 2004] introduce the PowerDB-IR system that maps IR strategies to SQL for document and structured data retrievals, in which it implements a TF-IDF-based model in SQL. Moreover, it investigated parallel processing for ad hoc query and online update based on database clusters with 2^n, where $n = 1, 2, 4, 8, 16$, nodes (machines) respectively. All systems mentioned above are based on mapping scoring or ranking functions of IR strategies to SQL

Instead of developing IR models directly in SQL, an alternative approach is to utilise databases as a storage layer and then to implement IR functionality as *middleware*, while such approach is especially preferable for the retrieval of semistructured data such as XML. For instance, a good example for this category is the TopX [Theobald et al., 2005b, Theobald et al., 2005c] search engine, which employs the threshold algorithm (TA) [Fagin et al., 2001, Güntzer et al., 2000, Nepal and Ramakrishna, 1999] for top-k processing specifically focuses on XML data, while more details about TopX will be discussed later in Section 4. In this approach, data are stored in database's tables that act as inverted indexes[6], while the middleware may fetch data from indexes by issuing simple Select statements of SQL with top-k constraint, in which sophisticated and judicious index accessing schedules are normally expected. The middleware differs from the approach of directly mapping IR on SQL in a sense that the scoring functions of IR models, or more precisely, the aggregations of computing scores for ranked results, are (partially or totally) external of database systems. Therefore, the middleware developers are more responsible for the system performance with regards to efficiency. Other systems fall into this category include, for example, [Hiemstra, 2002, Weigel et al., 2004, Weigel et al., 2005a, Weigel et al., 2005b].

Another theoretically possible but impractical approach is to deploy IR engine as backbone and let text retrieval methods to be called by built-upon database, ADTs and APIs

[6]Note that some commercial databases even support index-as-table data organisation using vanilla B+-tree index, for instance, the index organized table (IOT) in Oracle database.

should be available to access the underlying IR functionality. Such architecture was mentioned by [Chaudhuri et al., 2005] and referred as IR-via-ADTs, while it has been considered to be unpopular because it is rather complicated and inflexible but brings little benefit for integrated IR and DB applications.

So far as we can see, with respect to integration approaches, neither IR-on-SQL nor middleware provides an IR and DB integrated solution that could offer some benefits without compromising other aspects. To be specific, IR-on-SQL offers great flexibility but loses too much on efficiency, even if parallel processing is available, there is still a big gap to fill while comparing to dedicated IR systems; on the other hand, middleware approach solves efficiency problem to some extent, but it actually leaves less flexibility to application developers, because the scoring functions are usually coded within the middleware, which is in the end similar to dedicated IR systems.

In order to develop an integrating technology that could adapt the most of advantages of IR and DB without having to compromise useful features, logical layers have been developed to connect IR concepts to data models that could be adapted to the relational model which is relied on by databases, so that seamless integration of IR and DB may become practical. For instance, a Matrix framework for IR was introduced in [Roelleke et al., 2006], where a wide range of IR methods, such as IR concepts, frequencies or statistics, retrieval models, evaluation metrics, and so on, could be represented in a few standard matrix operations; in addition, a carefully chosen notation was proposed in the framework for allowing consistent meanings of frequencies in event spaces, which is readily applicable as building blocks for IR applications in common matrix operation libraries. In particular, an integrated DB&IR system, named parameterised search system (PSS) [Cornacchia and de Vries, 2006, Cornacchia and de Vries, 2007, Cornacchia et al., 2008], built upon an array database, the MonetDB [Boncz et al., 2006], has been developed by utilising the matrix framework [Roelleke et al., 2006].

Finally, an ultimate DB and IR integrated system may be built from scratch, which means much more efforts have to be paid, but on the other hand, a judiciously designed retrieval system that is specific for text and structured data is more likely to achieve the best of all desirable features such as flexibility, versatility, efficiency and scalability. Such system was named DB+IR in [Chaudhuri et al., 2005], and a good example is the QUIQ system [Kabra et al., 2003] for customer support applications. However, QUIQ was questioned by [Chaudhuri et al., 2005] that it

was not designed to be a generic infrastructure, so whether the system could be easily extended for wider applied domains is the main concern.

Different from QUIQ, the HySpirit [Fuhr and Rölleke, 1997, Roelleke and Fuhr, 1996] platform has been being built as a universal framework for text and structured data since mid 1990s, it bases on a probabilistic relational algebra (PRA) that integrates probabilistic theory and relational algebra, and several other high level abstraction layers such as probabilistic SQL (PSQL) [Roelleke et al., 2008], probabilistic Datalog (PD) [Fuhr, 2000, Wu et al., 2008a], and probabilistic object-oriented logic (POOL) [Roelleke, 1999, Fuhr et al., 1998] had been built upon the PRA platform. In additional, a new logical operator, the relational Bayes, has been proposed in [Roelleke et al., 2008] to support the modelling of IR-style probability estimations in PRA, where relations in a database could be then freely transformed between probabilistic and non-probabilistic, as a result, the modelling of IR strategies in PRA is to be more elegant than standard SQL.

To summarise, the state-of-the-art approaches for integrating DB and IR for supporting applications that need to handle retrievals of both text and structured data can be classified as the follows, in which a similar categorisation has also been used in [Chaudhuri et al., 2005]:

- IR-on-DB (IR-on-SQL): The IR scoring functions or ranking models to be mapped into standard SQL.
- Middleware: Databases are used as storage and indexes, whereas IR scoring functions or ranking models are coded in middleware system.
- IR-via-ADTs: Logically DB and IR are separated and parallel engines, where the DB engine utilises the IR engine through ADTs for text retrieval capabilities.
- RISC, or DB+IR/IR+DB: The idea of RISC is that IR layer is on top of a relational *storage* engine, where special designed declarative (query) languages or algebra provides the expressiveness for modelling IR strategies. If built-from-scratch DB+IR/IR+DB systems actually adapting techniques such as those in relational storage engine, then these could be viewed as in the same class as RISC.

Despite that different approaches have been proposed to integrate DB and IR, but there are several common features could be recognised in all approaches, in which these features might be supported in different degrees. These features are:

- High-levelled abstraction query languages are provided, e.g. declarative languages such as (probabilistic) SQL or expressive algebra such as PRA, for modelling retrieval strategies and scoring/ranking functions over sets or lists of tuples.

- Sophisticated designed query processing methods, e.g. top-k mechanism, for handling very large amount of data efficiently; and employing scalable solutions for handling continuously growing data, e.g. parallel machines or distributed computing.

- Rule-based query execution optimizations are provided, e.g. manipulations based on algebraic equivalence or cost-driven optimizations, for allowing application developers to tune the query execution engine for efficiently processing queries of certain applications.

However, so far there are no existing benchmarking methodologies for integrated DB and IR systems that measure all listing aspects as above. Despite of various benchmarks have been proposed respectively by IR community and DB community, while each benchmark has specific focus on either effectiveness or efficiency, none of them have quantitatively studied the correlated effects such as effectiveness versus efficiency.

The most used benchmark in IR for text retrieval is TREC[7], which provides testing collections in various scales from hundreds of megabytes to hundreds of gigabytes; in addition, it provides sets of queries and varied retrieval tasks, e.g. ad hoc or filtering, for evaluation; in recent years, it also proposed a number of tracks for specific application domains, e.g. web search or enterprise search. For evaluating retrievals over semi-structural documents, INEX[8] benchmark is used, which provides a number of XML collections as well as query sets, tasks, and assessment metrics.

On the DB side, the TPC-H[9] benchmark is made for decision support , which evaluates the efficiency of databases of handling complex SQL queries. In addition, a TEXTURE [Ercegovac et al., 2005] benchmark was proposed for measuring the efficiency of database queries over text fields.

Nevertheless, comprehensive benchmarks for integrated DB and IR systems are desirable, which may eventually evolve from combining and adapting mature DB and IR benchmarking methods, and this is believed to be an interesting area for future studies.

[7] http://trec.nist.gov/
[8] http://inex.is.informatik.uni-duisburg.de/
[9] http://www.tpc.org/tpch/

2.5.2 Modelling IR Strategies in Declarative Languages

In this section, we introduce modelling IR concepts and retrieval strategies in declarative languages, and thereby demonstrate how IR methods could be implemented in a DB and IR integrated paradigm.

As an example, let *MagColl* be a relation storing the toy magazine corpus in Figure 2.2, which schema is as given as:

```
MagColl(Term, DocId)
```

The common frequencies that will be discussed in the following section are illustrated in Table 2.3.

MagColl	
Term	*DocId*
hybrid	1
car	1
honda	1
bmw	2
green	2
car	2
hybrid	3
hybrid	3
car	3
prius	3
volt	3

(a) Original table

P_C_t	
score	*Term*
0.273	hybrid
0.273	car
0.091	honda
0.091	bmw
0.091	green
0.091	prius
0.091	volt

(b) $P_C(t)$ or $tf(t)$

P_C_t_d		
score	*Term*	*DocId*
0.33	hybrid	1
0.33	car	1
0.33	honda	1
0.33	bmw	2
0.33	green	2
0.33	car	2
0.4	hybrid	3
0.2	car	3
0.2	prius	3
0.2	volt	3

(c) $P_C(t|d)$ or $tf(t,d)$

df_t	
score	*Term*
0.67	hybrid
1.0	car
0.33	honda
0.33	bmw
0.33	green
0.33	prius
0.33	volt

(d) $df(t)$

Table 2.3: Example table of a toy magazine corpus for document retrieval

2.5.2.1 The Basics

First of all, the basic idea is to store and manage data, which includes unstructured, semi-structured and structured data such as text, XML and record, within a paradigm that bases on relational model.

In practice, a relation in database sense is either a set or a multiset. A multiset (e.g. see [Blizard, 1989]) can be formally defined as a pair (S,m) where S is some set and $m : S \rightarrow \mathbb{N}$ is a function from S to the set $\mathbb{N} = \{1,2,3,\ldots\}$ of positive natural numbers. The set S is the *underlying set of elements* of (distinct) tuples. For each tuple s in S the multiplicity of s is the number $m(s)$. For example, let relation $\mathcal{R}$ to be a multiset $\{a,a,b\}$ in which a and b are tuples, so that $\mathcal{R}$ is defined as $(\{a,b\},\{(a,2),(b,1)\})$. Therefore, the concept of set is a specialisation of the concept of multiset, where a set can be defined as $(\{t_1,t_2,\ldots,t_n\},\{(t_1,1),(t_2,1),\ldots,(t_n,1)\})$.

However, for the convenience of discussion, while referring multiset we stipulate that there is at least one tuple in the underlying set of elements satisfies multiplicity $m(s_i) > 1$.

In order to reflect the characteristic given by multiset and set, we use an α adornment technique[10], where α is either m or s, to indicate the essence of relation. Respectively, $\mathcal{R}^m$ means relation $\mathcal{R}$ is a multiset containing duplicate tuples, whereas $\mathcal{R}^s$ stands for relation $\mathcal{R}$ is a set containing only distinct tuples. Now we can define the cardinality of relations, which is needed for connecting common IR concepts with relational model and it is given in Table 2.4.

Notation	*Description*
$\lvert\mathcal{R}^\alpha\rvert$ or $\#\mathcal{R}^\alpha$	the cardinality of relation $\mathcal{R}$, where adornment α specifies whether $\mathcal{R}$ is a set (with adornment s) or a multiset (with adornment m) of tuples
$\lvert\mathcal{R}(A_1,\ldots,A_k)^\alpha\rvert$ or $\#\mathcal{R}(A_1,\ldots,A_k)^\alpha$	the cardinality of result obtained from projecting on attributes $A_1,\ldots,A_k$ of $\mathcal{R}$, where adornment α specifies whether the result is a set (with adornment s) or a multiset (with adornment m) of tuples

Table 2.4: Notation of cardinality

With respect to Table 2.4, two points are worth clarifying. First, if adornment α is not specified for $\mathcal{R}$, then in default $\mathcal{R}$ is considered to be a multiset, i.e. $\mathcal{R} = \mathcal{R}^m$. Second, be aware of the position of α: if it is adorning $\mathcal{R}$, then it is immediately after $\mathcal{R}$, e.g. $|\mathcal{R}^\alpha|$; else if it is adorning the result of projection, then it follows the closed parenthesis of attributes list, e.g. $|\mathcal{R}(A_1,\ldots,A_k)^\alpha|$.

Let us define some common IR frequencies with the above notation. By considering conventional IR notation and the notation proposed in [Roelleke et al., 2006], three widely used frequencies, i.e. *within-collection term/location frequency*, *within-document term/location frequency*, and *document frequency*, are defined as follows.

In [Roelleke et al., 2006], term frequency is also referred as *location frequency* (*lf*), which means the number of *locations* that a term occurs in a given space, for instance, within a collection or within a document. Here conventional name of *tf* is used but it is exchangeable with *lf*. First, the term frequency based on collection is defined as follow.

Definition 2.5.1. Within-Collection Term Frequency. Denoted as $tf(t)$ and its definition is given by:

$$tf(t) := \frac{|\mathcal{R}(t)^m|}{|\mathcal{R}^m|} \tag{2.25}$$

[10]Similar adorning technique had been applied to magic-set transformations, e.g. see [Ullman, 1988, Mumick et al., 1990a], which are for evaluation of Datalog rules, whereas here it is only used for the purpose of disambiguation.

If $\mathcal{R}$ is viewed as a sampling space of an event space, where the total number of events is $|\mathcal{R}^m|$, while for a random event t which occurs $|\mathcal{R}(t)^m|$ times, hence $tf(t)$ can be explained as probability $P_{\mathcal{R}}(t)$, where the subscription of P indicates the sampling space. If we specify the relation as a collection, then we can use $\mathcal{C}$ in the subscription as $P_{\mathcal{C}}(t)$:

$$\begin{aligned} \frac{|\mathcal{R}(t)^m|}{|\mathcal{R}^m|} &= P_{\mathcal{R}}(t) \\ &= P_{\mathcal{C}}(t) \end{aligned} \tag{2.26}$$

Second, the term frequency based on document is defined as follow.

Definition 2.5.2. Within-Document Term Frequency. Denoted as $tf(t,d)$ and its definition is given by:

$$tf(t,d) := \frac{|\mathcal{R}(t,d)^m|}{|\mathcal{R}(d)^m|} \tag{2.27}$$

Note that $tf(t,d)$ can be explained as a conditional probability of t given d. Similarly, considering $\mathcal{R}$ as sampling space, in which $tf(t,d)$ can be viewed as a division of conjunctive probability $P_{\mathcal{R}}(t,d)$ and unconditional probability $P_{\mathcal{R}}(d)$. Similarly, if we specify the relation as a collection, we use $\mathcal{C}$ in the subscription as $P_{\mathcal{C}}(t|d)$:

$$\begin{aligned} \frac{|\mathcal{R}(t,d)^m|}{|\mathcal{R}(d)^m|} &= \frac{|\mathcal{R}(t,d)^m|/|\mathcal{R}^m|}{|\mathcal{R}(d)^m|/|\mathcal{R}^m|} \\ &= \frac{P_{\mathcal{R}}(t,d)}{P_{\mathcal{R}}(d)} \\ &= P_{\mathcal{R}}(t|d) \\ &= P_{\mathcal{C}}(t|d) \end{aligned} \tag{2.28}$$

In [Roelleke et al., 2006], document frequency is defined upon document space. Given $\mathcal{R}$, the document space is the underlying set of elements, i.e. distinct tuples, of $\mathcal{R}$. Hence, the definition of df is given as follow:

Definition 2.5.3. Document Frequency. Denoted as $df(t)$ and its definition is given by:

$$df(t) := \frac{|\mathcal{R}(t,d)^s|}{|\mathcal{R}(d)^s|} \tag{2.29}$$

From an IR point of view, $df(t)$ is computed on a different event space from the tfs with respect to probability estimation: if we call the event space for estimating tfs as *tuple space*, i.e. events are either 1-tuples (i.e. within-collection tf as $\langle term \rangle$) or 2-tuples (i.e. within-document tf as $\langle term, doc \rangle$), then the event space for estimating df is called *document space* (see [Roelleke et al., 2006]), which means the space consists of distinct documents. In fact, we may extend the concept from document to wider subjects without losing generalisation, and call such event space as *subject space*.

2.5.2.2 An Extended PRA for Modelling IR

Previously, PRA [Fuhr and Rölleke, 1997] has been discussed in Section 2.4.2 within probabilistic databases paradigm, however, at least two constraints exist in this early version of PRA:

1. Probabilistic events are restricted to be independent only, which hardly satisfies most of real-life IR applications.

2. It only discusses probability aggregations with basic relational operators, whereas how initial probabilities could be obtained, i.e. probability estimation, has not been mentioned.

In order to extend the expressiveness of PRA for representing wider concepts of IR and supporting internal *probability estimation*, and extended PRA was introduced in [Roelleke et al., 2008]. One of the main contributions of [Roelleke et al., 2008] is that a new operator, the relation Bayes, was proposed for enabling probability estimation inside PRA. With the Bayes operator, conventional non-probabilistic relation can be "estimated" based on specified context or semantics, so that a non-probabilistic relation can be converted to probabilistic relation by assigning initial probabilities to its tuples. Unlike previous works such as [Fuhr, 1990, Fuhr and Rölleke, 1997, Dalvi and Suciu, 2004, Benjelloun et al., 2006a] that rely on *external* estimator for obtaining initial probabilities, the Bayes operator incorporates such function within a probabilistic database framework, which pushes the integration of DB and IR one step forward.

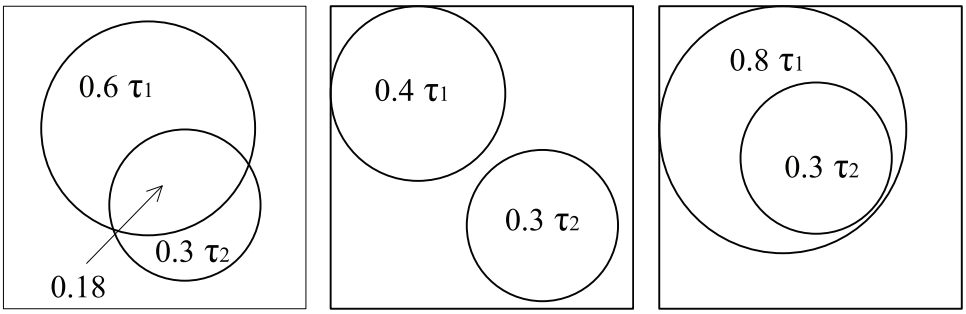
Here we briefly review the extended version of PRA discussed in [Roelleke et al., 2008]. First of all, the syntax of PRA is given in Figure 2.7.

For probability aggregation, three basic assumptions for indicating the relationship of each two tuples are consider, which are independent, disjoint, and subsumed. The relationships of two tuples under certain assumption is illustrated in Figure 2.8.

prae := Selection | Projection | Join | Union | Subtraction | Bayes | Relation
Selection := 'SELECT' '[' EMPTY | praCondition ']' '(' prae ')' Projection := 'PROJECT' assumption '[' EMPTY | varList ']' '(' prae ')' Join := 'JOIN' probAssumption '[' EMPTY | praCondition ']' '(' prae ',' prae ')' Union := 'UNITE' assumption '(' prae ',' prae ')' Subtraction := 'SUBTRACT' probAssumption '(' prae ',' prae ')' Bayes := 'BAYES' probAssumption '[' EMPTY | varList ']' '(' prae ')' Relation := NAME
assumption := 'DISTINCT' | 'ALL' | probAssumption probAssumption := 'DISJOINT' | 'INDEPENDENT' | 'SUBSUMED' | 'SUM_LOG' | 'MAX_LOG' predicate := '=' | '!=' | '<' | '<=' | '>' | '>=' var := '$'NAME value := STRING | NUMBER varList := var [',' varList] praCondition := var predicate (var | value) [',' praCondition]

Figure 2.7: Syntax of extended PRA

In general, different aggregation functions are applied to PRA operators while aggregate probabilities, which depend on the given assumptions in PRA expressions. The aggregation functions for certain assumptions are demonstrated in Figure 2.9.



(a) Independent (b) Disjoint (c) Subsumed

Figure 2.8: Assumptions: independent, disjoint, and subsumed

The aggregation functions and assumptions are complements to the previous definitions of basic PRA operators in Section 2.4.2.

For probability estimation, a relational Bayes operator is applied. Informally, Bayes estimates tuple probability in the following two ways depending on if the input relation is non-probabilistic or probabilistic:

- If the input relation is non-probabilistic, Bayes counts the number of tuples grouped by *evidence* attribute A, and then assigns to each tuple a probability of $1.0/|\mathcal{R}(A)^{\alpha}|$;

$$
\begin{aligned}
P(\tau_i \vee \tau_j) &:= \begin{cases} P(\tau_i) + P(\tau_j) - P(\tau_i) \cdot P(\tau_j) & \text{if independent} \\ P(\tau_i) + P(\tau_j) & \text{if disjoint} \\ \max(\{P(\tau_i), P(\tau_j)\}) & \text{if subsumed} \end{cases} \\
P(\tau_i \wedge \tau_j) &:= \begin{cases} P(\tau_i) \cdot P(\tau_j) & \text{if independent} \\ 0 & \text{if disjoint} \\ \min(\{P(\tau_i), P(\tau_j)\}) & \text{if subsumed} \end{cases} \\
P(\tau_i \wedge \neg \tau_j) &:= \begin{cases} P(\tau_i) \cdot (1 - P(\tau_j)) & \text{if independent} \\ P(\tau_i) & \text{if disjoint} \\ P(\tau_i) - P(\tau_j) & \text{if subsumed and } P(\tau_i) > P(\tau_j) \\ 0 & \text{if subsumed and } P(\tau_i) \leq P(\tau_j) \end{cases}
\end{aligned}
$$

Figure 2.9: Assumptions and probability aggregations: independent, disjoint, and subsumed

- If the input relation is probabilistic, then Bayes sums the probabilities of tuples grouped by *evidence* attribute A, and the assigns to each tuple a probability of $1.0/\sum_{i=0}^{n} P(\tau_i)$, where $P(\tau)$ is tuple probability, and $n = |\mathcal{R}(\tau)^{\alpha}|$ which is the number of tuples for a specified value of attribute A.

2.5.2.3 Examples of Modelling Probability Estimations

Here, examples of modelling probability estimations for some common IR frequencies (see Section 2.5.2.1) with declarative languages are given. In terms of modelling, it means to write queries representing scoring and ranking functions in a declarative language, e.g. SQL queries or PRA queries, to yield weighted and ranked results. On the one hand, we demonstrate the modelling of within-collection *tf*, within-document *tf*, and *df* in standard SQL and PRA. On the other hand, the examples also show that modelling similar IR concepts or retrieval strategies in PRA could be more elegant than modelling in standard SQL.

Modelling with standard SQL Modelling probability estimations in standard SQL involves composing complex queries. According to [Grossman et al., 1997], complex SQL queries are queries containing two or more query blocks. For instance, if a $SELECT \ldots FROM \ldots WHERE$ statement contains embedded $SELECT \ldots FROM \ldots WHERE$ statement(s) then it is a complex query, whereas a single $SELECT \ldots FROM \ldots WHERE$ statement involving multiple joins is not a complex query.

The following SQL statement simulates the Bayes operation of PRA while has not been given evidence key (attribute):

```
SELECT Term, MagColl.DocId, eventSpace.weight
FROM MagColl, (SELECT 1.0/COUNT(*) AS weight
  FROM MagColl) AS eventSpace;
```

The following SQL statement simulates the Bayes operation while has been given *DocId* as evidence key (attribute):

```
SELECT Term, MagColl.DocId, eventSpace.weight
FROM MagColl, (SELECT DocId, 1.0/COUNT(*) AS weight
  FROM MagColl GROUP BY DocId) AS eventSpace
WHERE MagColl.DocId = eventSpace.DocId;
```

An SQL statement for getting within-collection *tf* is given as follow:

```
SELECT Term, SUM(eventSpace.weight) AS P_C_t
FROM MagColl, (SELECT 1.0/COUNT(*) AS weight
  FROM MagColl) AS eventSpace
  GROUP BY Term;
```

An SQL statement for getting within-document *tf* is given as follow:

```
SELECT Term, MagColl.DocId,
  SUM(eventSpace.weight) AS P_C_t_d
  FROM MagColl,
     (SELECT DocId, 1.0/COUNT(*) AS weight
     FROM MagColl GROUP BY DocId) AS eventSpace
WHERE MagColl.DocId = eventSpace.DocId
  GROUP BY Term, MagColl.DocId;
```

An SQL statement for getting *df* is given as follow:

```
SELECT docSpace.term, docSpace.weight/eventSpace.weight AS df_t
FROM (SELECT DISTINCT Term, COUNT(DISTINCT DocId) AS weight
  FROM MagColl GROUP BY Term) AS docSpace,
  (SELECT COUNT(DISTINCT DocId) * 1.0 AS weight
    FROM MagColl) As eventSpace
    GROUP BY docSpace.Term, docSpace.weight, eventSpace.weight;
```

Modelling with PRA The following PRA expression demonstrates the Bayes operation while has not been given evidence key (attribute):

```
weight = BAYES DISJOINT [](MagColl);
```

The following PRA expression demonstrates the Bayes operation while has been given *DocId* as evidence key (attribute):

```
weight = BAYES DISJOINT [$DocId](MagColl);
```

The following PRA expression yields results with within-collection *tf*:

```
P_C_t = PROJECT DISJOINT [$Term](
      BAYES DISJOINT [](MagColl));
```

The following PRA expression yields results with within-document *tf*:

```
P_C_t_d = PROJECT DISJOINT [$Term,$DocId](
        BAYES DISJOINT [$DocId](MagColl));
```

The following PRA expressions yield results with *df*:

```
dColl = PROJECT DISTINCT(MagColl);

docSpace = BAYES [](
            PROJECT DISTINCT [$DocId](MagColl));

termDoc = PROJECT [dColl.$Term, dColl.$DocId](
           JOIN [dColl.$DocId = docSpace.$DocId](dColl, docSpace));

df_t = PROJECT DISJOINT [$Term](termDoc);
```

2.6 Summary

In this chapter, we reviewed the state-of-the-arts of information retrieval and database technologies, in which we focused on some specific areas that are related to the integration of IR and DB technologies.

To summarise, we covered four main fields in the previous discussions: First, we reviewed several interesting topics in traditional IR research, which include the general architecture of IR systems, conventional IR tasks, and popular IR models. Second, we reviewed previous studies aiming to integrate ranking into relational databases. Third, we discussed the details of probabilistic databases, where we addressed the theory and techniques behind probabilistic databases, which include the underlying possible worlds model (of conventional probabilistic database), a probabilistic relational algebra, and query evaluation techniques for conjunctive queries based on extensional semantics. Final, we introduced related research specific to the integration of IR and DB technologies, where we reviewed various integrated solutions and architectures, while in particular, we focused on one of the approaches called IR+DB and discussed some important background and features; for example, an extended version of PRA, and modelling IR strategies in declarative languages.

Chapter 3

SCX: Scoring-Driven Query Optimization with Scoring Expression

3.1 Introduction

Rule-based query optimization is one the most important techniques in databases technology. At the moment, most of the approaches could be mainly categorised into two broad folders: algebra optimization, cost-driven optimization. The first one bases on the laws of algebraic equivalence and conducts through algebraic manipulations, in which an algebra expression might be rewritten into another one or several transformation(s) that yield(s) equivalent result. While not changing logical expressions, the second approach attempts to choose the least expensive implementations in hand for logical operators, this could be achieved by enumerating the candidates (i.e. physical operators) space through dynamic programming, in which, necessarily, certain predefined cost models for estimating the costs of considered implementations are deployed. In addition, there is a less general, comparing to the first two, yet could be very effective method that optimizes queries based on semantics. Although semantic optimization is usually employed in ad hoc manners, but along with sophisticated techniques, for instance, ontology or abstract semantic graphs, it is possible to design a rule-based semantics-driven optimizer for databases and IR+DB systems.

In this chapter, we propose a scoring-driven optimization technique for probabilistic relational algebra (PRA). In particular, we introduce *scoring expression (SCX)*, which is employed for articulating the semantics of scoring functions that are implied in PRA expressions.

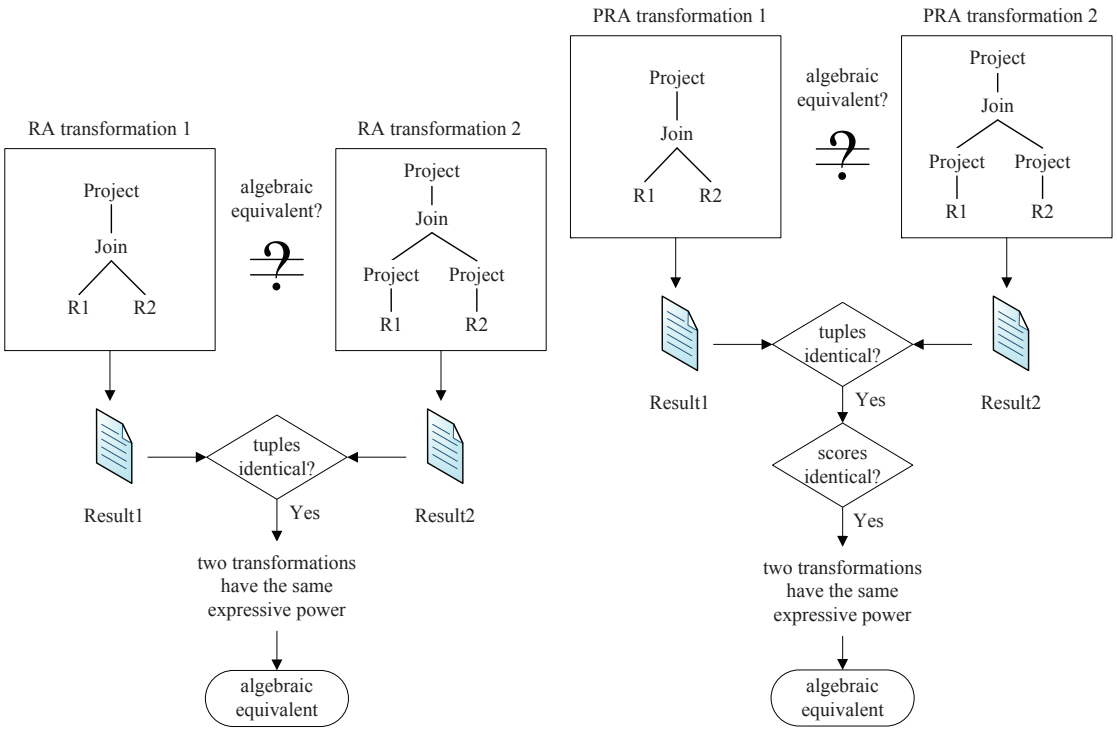
In short, the idea behind scoring-driven optimization is to analyse the semantics of PRA expressions that are represented in SCX, so that supplying additional semantic information to PRA execution engine about how scores or probabilities could be or should be computed. For instance, PRA expressions for estimating varied kind of probabilities (see Section 2.5) could be therefore linked to ordinary aggregations such as counting and summing, which in a way provides possibilities to PRA execution engine to utilise statistic-materialised indexes such as a relational inverted index (RIX) that will be discussed later (see Chapter 5). For another instance, SCX could be manipulated and turned into different transformations based on rules and probabilistic assumptions, and it provides an additional choice to [Dalvi and Suciu, 2004] and [Benjelloun et al., 2006a] for query evaluation on probabilistic database based on extensional semantics. In fact, SCX fills the gap between intensional evaluation and extensional evaluation by visualising how probabilities are aggregated.

Observably, a scoring-driven optimization is based on semantic equivalence of scoring functions, whereas an algebraic optimization is based on algebraic equivalence. However, the actual purpose of scoring-driven optimization is to achieve algebraic equivalence. Here we discuss informally the various equivalence and the difference of algebraic equivalence on traditional (non-probabilistic) relational algebra (RA) and PRA.

An algebraic equivalence requires different transformations of RA expressions yields the same set of results. Whereas for PRA it is even stricter to achieve algebraic equivalence amongst transformations, because it needs PRA expressions not only comply to the requirements for traditional RA, but also the results of different transformations should satisfy scoring equivalence, i.e. the computed scores for tuples in the results of PRA transformations should be identical. However, the classic law of algebraic manipulations is for traditional RA, which does not contain any semantics of scoring methods that are included in PRA, which poses uncountable difficulties for a rule-based optimizer to find equivalent PRA transformations. In addition, note that traditional equivalence of RA expressions does not imply same order of results, whereas order might need to be considered while investigating equivalence in a ranking paradigm such as PRA. In other words, ordering could pose additional complication for proving equivalence of PRA expressions. The differences of algebraic equivalence between traditional RA and PRA are illustrated in Figure 3.1[1]. As a result, we were motivated to develop a method that is able to express the semantics

[1]It is worth clarifying that Figure 3.1 and Figure 3.2 are illustrations rather than procedure flow charts.

of the underlying scoring functions of PRA expressions, and it leads us to the proposal of scoring expression.



(a) RA equivalence (b) PRA equivalence

Figure 3.1: Algebraic equivalence of traditional RA and PRA expressions

In order to decide whether two PRA expressions are algebraic equivalent, one has to scrutinise the scoring functions of the PRA expressions, if and only if the implied scoring functions produce same scores for all corresponding tuples in respective results, i.e. to be scoring equivalent, one of the requirements of algebraic equivalence for PRA expressions could be satisfied. By utilising SCX, scoring equivalence of PRA transformations could be verified if and only if their interpreted scoring expressions are semantically equivalent (see Figure 3.2). In other words, the SCX provides a necessary and comprehensive mechanism for the verification of algebraic equivalence of PRA expressions.

Outline of Chapter First, traditional optimization techniques for databases are reviewed in Section 3.2; second, the syntax and semantics of scoring expression (SCX) are discussed in Section 3.3; third, the methods of scoring-driven optimization are addressed in Section 3.4; moreover, experiments and experimental results are presented in Section 3.5; finally, the chapter is summarised in Section 3.6

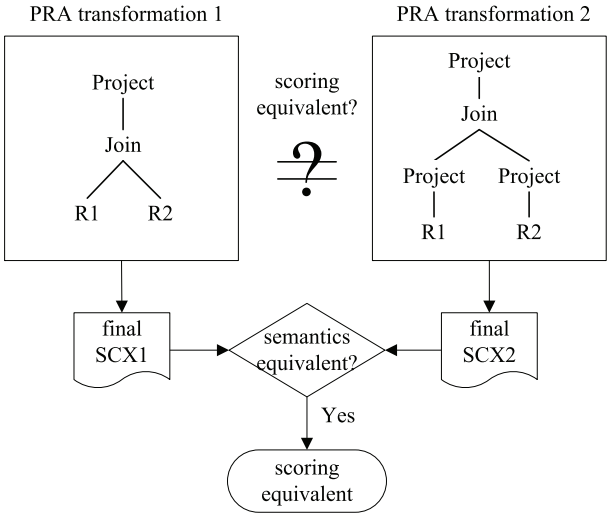


Figure 3.2: Scoring equivalence of PRA expressions

3.2 Query Optimizations for Databases

Query optimizations in databases can be categorised into two folds: logical optimization and physical optimization. In general, logical optimization focuses on manipulating algebra expressions, where expensive expressions are replaced by equivalent transformations (e.g. [Yan and Larson, 1995, Cherniack and Zdonik, 1998]). In particular, magic-sets (e.g. [Mumick et al., 1990b, Beeri and Ramakrishnan, 1991]) is a special RA reformulating method that adds constraint to algebra expressions, so that limits intermediate results.

On the other hand, physical optimization focuses on mapping the logical algebra to a physical algebra, and follows a divide-and-conquer strategy that generates a least-cost execution plan using dynamic programming (e.g. [Cole and Graefe, 1994]). Physical implementations include, for example, join algorithms (e.g. [Li et al., 2002], materialised views (e.g. [Goldstein and Larson, 2001]), index selection (e.g. [Agrawal et al., 2000]), and top-k processing (e.g. [Fagin et al., 2003b]). In particular, rank-aware optimizations [Ilyas et al., 2004, Li et al., 2005] are proposed to handling special rank-joins (e.g. [Natsev et al., 2001]).

In [Dalvi and Suciu, 2004], an efficient query evaluation technique on probabilistic databases was proposed. Their method is to replace expensive intensional query plan by less expensive extensional plan. However, an extensional plan may lead to incorrect scores where scores bigger than one are computed, which breaks the correctness of probabilistic semantics. Their solution is to search and exploit the "safe" extensional plans only. It is worth mentioning that this work is the most close method to our scoring-driven approach.

There has been extensive work in query processing and optimization since the early 70s. It is hard to capture the breadth and depth of this large body of work in a short section. On the other hand, it is one of the fundamentals of databases world, hence it is a compulsory topic in almost every database text book, e.g. see [Ullman, 1989].

The synonym of query processing in DB is query evaluation. In [Graefe, 1993], the author reviews wide range of state-of-the-art query evaluation techniques for large databases, such as sorting and hashing, disk access, aggregation, various join algorithms, query execution, parallel algorithms, complex query plan, and so on.

Among several specific research topics of evaluation, studying efficient join algorithms caught many interests (e.g. see [Shapiro, 1986, Mishra and Eich, 1992]), because join is one of the expensive operations (and maybe is the 'most expensive' one). As we known, the nested-loop-join is a 'naive' physical implementation of join, it concatenates the tuples from two relations within a nested loop, and then produces a new relation. More advanced algorithms have been proposed based on hashing (hash join, e.g. see [Mokbel et al., 2004]) and sorting (merge join, e.g. see [Graefe, 1994, Li et al., 2002, Dittrich et al., 2002]). The reason that join is expensive is because it takes two relations as input and produces a new relations. In addition, in the case of there are aggregations in the branches of a join then extra difficulties are added to the manipulation. Another operation that also produces intermediate result and involves aggregations is union, thus, interests were caught as well (e.g. see [Galil and Italiano, 1991]).

Apart from the physical operations that related to the relational operators, there is one other physical operations that do not have a relational counterpart but generated interest as well, which is the sort operation. Sorting algorithms are categorised to memory sort and external sort. The data of memory sort are all held in memory while sorting, however, if data size is larger than memory size, the external sort should be applied instead. [Estivill-Castro and Wood, 1992] introduces various sorting algorithms that include memory sort and external sort. In [Graefe, 2006], the author reviews the well-known sorting algorithms that have been implemented in databases.

The two key components of query evaluation of a SQL database are the query optimizer and the query execution engine. The query execution engine implements a set of physical operators, and the query optimizer is responsible for generating the input (with low cost) for the execution engine. An overview of query optimization in relational systems is given by [Chaudhuri, 1998]. In particular, cost model [Chaudhuri and Shim, 1995,

Chaudhuri and Shim, 1996] is applied to assign an estimated cost to any partial or complete plan. Other important techniques for implementing query optimizer include dynamic programming [Graefe and Ward, 1989, Cole and Graefe, 1994] and use of interesting orders [Chaudhuri, 1998]. Alone the line, research have been carried on the algebraic optimization, i.e. to study different transformations of algebra expressions in order to find out the efficient but equivalent expressions of queries. A study of equivalence of the Project-Select-Join queries is given by [Yang and Larson, 1987]. [Yan and Larson, 1995] studies the transformations of aggregations. Note that [Chaudhuri, 1998] points out that "*transformations do not necessarily reduce cost and therefore must be applied in a cost-based manner by the enumeration algorithm to ensure a positive benefit*".

Another technology that worth paying attention is the processing of views. A view is a relation that represented by a query, i.e. the intermediate result of an algebra expression, it is also called intensional relation [Ullman, 1988]. The evaluation of some views could be extremely expensive, because they are obtained by evaluating complex queries while we cannot build index to facilitate the processing. During past decades, the materialised view was proposed (e.g. see [Goldstein and Larson, 2001]), so that view can be processed as ordinary tables.

3.2.1 Algebraic Manipulation

For a canonical relational algebra (RA) that bases on sets, an RA expression could have one or more transformations as long as they comply to the laws of algebraic manipulations (e.g. see [Ullman, 1989]). The basic idea behind the algebra optimization is that since these RA transformations have the same expressive power, in other words, the result sets yielded by the transformations should be identical, hence, there might exist one of the transformations that is superior than the others in terms of executive efficiency while it is under certain circumstances.

Different extensions had been made to the canonical RA and equivalence of these extensions had been studied. For instance, including predicates which leads to multisets; including aggregate functions into relational calculus and RA (see [Klug, 1982]); integrating probabilistic theory and set theory leads to PRA (see [Fuhr and Rölleke, 1997, Roelleke, 1994]); introducing rank predicate leads to RankSQL (see [Li et al., 2005]). Nonetheless there are various extensions of RA, the basic laws of algebraic manipulations are applicable to both canonical RA and its variants or extensions.

Here we introduce the laws of algebraic manipulations for algebra optimization, where com-

patible notations to Section 2.4 are used.

Law 3.2.1. Commutative laws for joins and products. For relations $\mathcal{A}_{\mathrm{X}}$ and $\mathcal{B}_{\mathrm{Y}}$, while giving predicate function $\mu : \overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}$, then

$$\begin{aligned} \mathcal{A}_{\mathrm{X}} \bowtie_{\mu:\overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}} \mathcal{B}_{\mathrm{Y}} &\equiv \mathcal{B}_{\mathrm{Y}} \bowtie_{\mu:\overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}} \mathcal{A}_{\mathrm{X}} \\ \mathcal{A}_{\mathrm{X}} \times \mathcal{B}_{\mathrm{Y}} &\equiv \mathcal{B}_{\mathrm{Y}} \times \mathcal{A}_{\mathrm{X}} \end{aligned}$$

Law 3.2.2. Associative laws for joins and products. For relations $\mathcal{A}_{\mathrm{X}}$, $\mathcal{B}_{\mathrm{Y}}$ and $\mathcal{C}_{\mathrm{Z}}$, while giving predicate functions $\mu_1 : \overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}$ and $\mu_2 : \overline{\mathrm{Y}}\Theta\overline{\mathrm{Z}}$, then

$$\begin{aligned} (\mathcal{A}_{\mathrm{X}} \bowtie_{\mu_1:\overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}} \mathcal{B}_{\mathrm{Y}}) \bowtie_{\mu_2:\overline{\mathrm{Y}}\Theta\overline{\mathrm{Z}}} \mathcal{C}_{\mathrm{Z}} &\equiv \mathcal{A}_{\mathrm{X}} \bowtie_{\mu_1:\overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}} (\mathcal{B}_{\mathrm{Y}} \bowtie_{\mu_2:\overline{\mathrm{Y}}\Theta\overline{\mathrm{Z}}} \mathcal{C}_{\mathrm{Z}}) \\ (\mathcal{A}_{\mathrm{X}} \times \mathcal{B}_{\mathrm{Y}}) \times \mathcal{C}_{\mathrm{Z}} &\equiv \mathcal{A}_{\mathrm{X}} \times (\mathcal{B}_{\mathrm{Y}} \times \mathcal{C}_{\mathrm{Z}}) \end{aligned}$$

Law 3.2.3. Cascade of projections. For a relation $\mathcal{R}_{\mathrm{X}}$, projected attributes $\overline{\mathrm{X}}_1$ and $\overline{\mathrm{X}}_2$, where $\overline{\mathrm{X}}_1 \subseteq \overline{\mathrm{X}}_2$, then

$$\Pi_{\overline{\mathrm{X}}_1}\left(\Pi_{\overline{\mathrm{X}}_2}(\mathcal{R}_{\mathrm{X}})\right) \equiv \Pi_{\overline{\mathrm{X}}_1}(\mathcal{R}_{\mathrm{X}})$$

Law 3.2.4. Cascade of selections. For a relation $\mathcal{R}_{\mathrm{X}}$, predicate functions $\mu_1 : X\Theta y$ and $\mu_2 : X\Theta z$, then

$$\sigma_{\mu_1:X\Theta y}\left(\sigma_{\mu_2:X\Theta z}(\mathcal{R}_{\mathrm{X}})\right) \equiv \sigma_{\mu_1:X\Theta y \,\wedge\, \mu_2:X\Theta z}(\mathcal{R}_{\mathrm{X}})$$

Since $\mu_1 \wedge \mu_2 = \mu_2 \wedge \mu_1$, it could be concluded immediately that selections can be commuted, which is

$$\sigma_{\mu_1:X\Theta y}\left(\sigma_{\mu_2:X\Theta z}(\mathcal{R}_{\mathrm{X}})\right) \equiv \sigma_{\mu_2:X\Theta z}\left(\sigma_{\mu_1:X\Theta y}(\mathcal{R}_{\mathrm{X}})\right)$$

Law 3.2.5. Commuting selections and projections. In a condition that if predicate function $\mu : X\Theta x$ involves only projected attributes $\overline{\mathrm{X}}$, then

$$\Pi_{\overline{\mathrm{X}}}\left(\sigma_{\mu:X\Theta x}(\mathcal{R}_{\mathrm{X}})\right) \equiv \sigma_{\mu:X\Theta x}\left(\Pi_{\overline{\mathrm{X}}}(\mathcal{R}_{\mathrm{X}})\right)$$

Law 3.2.6. Commuting selection with Cartesian product.

$$\sigma_{\mu:X\Theta x}(\mathcal{A}_{\mathrm{X}} \times \mathcal{B}_{\mathrm{Y}}) \equiv \sigma_{\mu:X\Theta x}(\mathcal{A}_{\mathrm{X}}) \times \mathcal{B}_{\mathrm{Y}}$$

$$\sigma_{\mu_1:X\Theta x \wedge \mu_2:Y\Theta y}(\mathcal{A}_X \times \mathcal{B}_Y) \equiv \sigma_{\mu_1:X\Theta x}(\mathcal{A}_X) \times \sigma_{\mu_2:Y\Theta y}(\mathcal{B}_Y)$$
$$\sigma_{\mu_1:X\Theta x \wedge \mu_2:Y\Theta y}(\mathcal{A}_X \times \mathcal{B}_Y) \equiv \sigma_{\mu_2:Y\Theta y}\left(\sigma_{\mu_1:X\Theta x}(\mathcal{A}_X) \times \mathcal{B}_Y\right)$$

Law 3.2.7. Commuting selection with a union.

$$\sigma_{\mu:X\Theta x}(\mathcal{A}_X \cup \mathcal{B}_X) \equiv \sigma_{\mu:X\Theta x}(\mathcal{A}_X) \cup \sigma_{\mu:X\Theta x}(\mathcal{B}_X)$$

Law 3.2.8. Commuting selection with a set difference.

$$\sigma_{\mu:X\Theta x}(\mathcal{A}_X - \mathcal{B}_X) \equiv \sigma_{\mu:X\Theta x}(\mathcal{A}_X) - \sigma_{\mu:X\Theta x}(\mathcal{B}_X)$$

Law 3.2.9. Commuting selection with natural join – special case.

$$\sigma_{\mu:(X\cup Y)\Theta z}(\mathcal{A}_X \bowtie \mathcal{B}_X) \equiv \sigma_{\mu:X\Theta z}(\mathcal{A}_X) \bowtie \sigma_{\mu:Y\Theta z}(\mathcal{B}_Y)$$

Law 3.2.10. Commuting a projection with a Cartesian product.

$$\Pi_{\overline{X}\cup\overline{Y}}(\mathcal{A}_X \times \mathcal{B}_Y) \equiv \Pi_{\overline{X}}(\mathcal{A}_X) \times \Pi_{\overline{Y}}(\mathcal{B}_Y)$$

Law 3.2.11. Commuting a projection with a union.

$$\Pi_{\overline{X}}(\mathcal{A}_X \cup \mathcal{B}_X) \equiv \Pi_{\overline{X}}(\mathcal{A}_X) \cup \Pi_{\overline{X}}(\mathcal{B}_X)$$

3.3 Scoring Expression

3.3.1 Discovering the Scoring Semantics of PRA Expressions

We discussed the definitions of PRA operators in previous sections (see Section 2.4.2 and Section 2.5.2.2), attentive readers may have noticed that there are two kinds of semantics have been addressed in the definitions, which are relational semantics and scoring semantics.

With respect to relational semantics, PRA is equivalent to conventional non-probabilistic relational algebra; in addition, PRA incorporates scoring semantics, or to be precise, probabilistic semantics, within its operators. Therefore, this is why traditional algebraic manipulation cannot be applied to optimize PRA expressions without extra considerations with regards to their scoring semantics. In fact, the laws of algebraic manipulations (see Section 3.2.1) could be no longer applicable to a relational algebra while considering sorts of scoring semantics: there are few

algebraic equivalences can be found. Several earlier studies have encountered similar problems, for instance, in the SALT algebra proposed by [Chaudhuri et al., 2005].

Mapping scoring functions to relational algebra already poses difficulties to query optimization while the mapping between relational operators and scoring functions is a bijection, and if the mappings are one-to-many mappings such as the ones in PRA, where which scoring function is chosen to be mapped to an operator also depends on an applied (probabilistic) assumption, then it is impossible for a query optimizer based on algebraic equivalences can be practical.

To tackle the problem of very few algebraic equivalence can be found for scoring-based relational algebra such as PRA, scoring expression (SCX) was proposed to articulate scoring semantics of PRA expressions. In addition, a scoring-driven optimization technique shows how SCX can be transformed to alternative scoring expressions, or so-called *interpreted* SCX. The contributions of this technique, which explicitly presents the scoring semantics of PRA expressions, are two folds:

- It helps with detecting algebraic equivalence of PRA expressions. Since it is a tool for describing scoring functions which can be understandable by both human user and machine, on the one hand, it is necessary for human user (e.g. application developer) to understand the scoring semantics in order to discover equivalent PRA expressions; and on the other hand, it is also necessary for a rule-based query optimizer to analyse the scoring semantics in order to trigger pre-set manipulations.

- It helps with implementing other optimizations such as specialised operations and index selections. In other words, it is necessary for developing RISC-like physical operators for common scoring functions. For example, *tf* is a common frequency that is applied in several popular ranking models, e.g. BM25 and LM (see Section 2.2.2), hence using a dedicated operator for *tf* rather than executing a complex SQL query (e.g. see 2.5.2.3) on-the-fly is more practical while considering efficiency and scalability.

3.3.2 Equivalence of PRA Expressions

Algebraic equivalence is the cornerstone of optimization for relational expressions, as it is pointed out in [Ullman, 1989]:

> "Before we can 'optimize' expressions we must understand clearly when two expressions of relational algebra are equivalent."

Similarly, we must understand clearly when two expressions of probabilistic relational algebra [Fuhr and Rölleke, 1997, Roelleke et al., 2008] (or PRAE for short) are equivalent before we can optimize them. So far we have discussed the definition of relations in probabilistic database (see Section 2.4.1), which employs a viewpoint that is also shared in conventional databases, where a relation is a set of mappings from a set of attribute names to values (i.e. names for columns); whereas it is different from traditional relation in a sense that a probabilistic weighting is attached to each of these mappings (tuples), which is so-called tuple probability (or tuple score in a broad sense).

Before we discuss formally what criteria are necessary for PRAEs to be equivalent, let us first recall the definition of canonically algebraic equivalence of non-probabilistic relational expressions: A relational expression taking relations $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k$ as operands defines a function whose domain is k-tuples of relations $(r_1, r_2, \ldots, r_k)$, each r_i being a relation of the arity appropriate to $\mathcal{R}_i$. The outcome of the function is a relation which is the result of evaluating the expression by substituting each r_i for $\mathcal{R}_i$. Two relational expressions RE_1 and RE_2 are *equivalent*, written $RE_1 \equiv RE_2$, if they represent the same mappings, which means when substituting the same relations for identical names in the two expressions, we obtain the same result.

In order to define equivalence for PRAE, we also need to consider how event probabilities are computed by an expression. Moreover, since it is discussed in [Roelleke et al., 2008], probability estimations could be performed to transform non-probabilistic relations into probabilistic relations, which means the intermediate results of probability estimations could be scored with some sort of weightings which could be not yet probabilities. On the other hand, from and IR point of view, it would be appropriate to discuss algebraic equivalence under a broad sense of weightings rather than restricting ourselves to probabilistic weightings only. Therefore, we loosen the previous definition on probabilistic relation, and define scored relation as follow:

Definition 3.3.1. *Weighted Relation*. Let a scored relation $\mathcal{R}$ be a 4-tuple of $(\mathbf{X}, \Delta, \mathbf{f}, \mathbf{s})$, where X is a schema that is a set of attributes $A_1, A_2, \ldots, A_k$, so a relation $\mathcal{R}$ having schema X is denoted as $\mathcal{R}_\mathrm{X}$; and Δ is a set of domains $\{D_1, D_2, \ldots, D_k\}$, in which a domain contains a set of values $\{v\}$; the *Cartesian product* (or just *product*) of all domains, written $D_1 \times D_2 \times \cdots \times D_k$, is the set of all k-tuples $(v_1, v_2, \ldots, v_k)$, denoted as τ, such that v_1 is in D_1, v_2 is in D_2, and so on; and f is a function that performs mappings from attribute names X to values τ, i.e. $\mathrm{f}(\mathrm{X}') \rightarrow \tau'$, which yields any subset of the Cartesian product of one or more domains; and $\mathbf{s}$ is a scoring function

that computes weightings to assign to every mappings produced by f.

With this definition of scored relation, now we are able to discuss the definition of equivalence of PRA expressions. There are two considerations for two PRAEs to be equal: firstly, they must yield the same result with respect to the values of tuples (or just tuples for short), which is the same consideration as equivalence for conventional relational expressions; and secondly, the weightings computed by either PRAE must be the same. Since both criteria are important to verify equivalence of PRAEs, we call the first criterion *relational equivalence* and the second *weighting equivalence*.

Since relational equivalence has been established for decades, we do not repeat its discussions, and move on to the issues with regard to weighting equivalence: what does it mean exactly *the same weighting*? Firstly, we assume that "weightings are equivalent" as long as "the scores representing for weights are equal", in other words, *scoring equivalence*. So let us consider the following examples in Figure 3.3.

	Term	*DocId*
0.9	prius	doc1
0.8	volt	doc1

	Term	*DocId*
0.45b	prius	doc1
0.4b	volt	doc1

Figure 3.3: Example 1 of soft scoring equivalence

Let b be a parameter, so are the above two relations scoring equivalent? A cautious answer is "perhaps", because if $b = 2$ then they are equal with respect to scores. Moreover, let us consider another example in Figure 3.4.

	Term	*DocId*
0.9	prius	doc1
0.85	volt	doc1

	Term	*DocId*
0.89999	prius	doc1
0.84999	volt	doc1

Figure 3.4: Example 2 of soft scoring equivalence

So are they scoring equivalent? Again, it depends on the allowed precision of scores and whether rounding would be taken into account. Actually, we can go even further and have another example in Figure 3.5.

	Term	*DocId*
0.9	prius	doc1
0.8	volt	doc1

	Term	*DocId*
0.7	prius	doc1
0.5	volt	doc1

Figure 3.5: Example of strict ranking equivalence

This time one might say immediately: "No, they are absolutely not scoring equivalent!". However, if our interest is not on the exact scores but the relative importance of the tuples, then those two results actually tell the same, so they could be *about* the same at all. If we can tolerate differences on exact tuple scores but put more emphasis on the *order* of tuples based on weightings, then we can relax the concept of scoring equivalence to *ranking equivalence*.

However, we have not yet covered all the issues, because one have to expect some "odd" weightings to be yielded by IR strategies, for instance, can someone tell if the lists of results in Figure 3.6 and Figure 3.7 could be ranked the same?

	Term	*DocId*
1.0	prius	doc1
0.99999	volt	doc1

	Term	*DocId*
1.0	volt	doc1
1.0	prius	doc1

Figure 3.6: Example 1 of soft ranking equivalence

	Term	*DocId*
0.9	prius	doc1
0.8	volt	doc1

	Term	*DocId*
0.9	prius	doc1
0.8	volt	doc1
0.0	range-rover	doc1

Figure 3.7: Example 2 of soft ranking equivalence

As what have been demonstrated, it is necessary to specify the circumstances when weightings can be considered to be equivalent. In fact, we can have the following four situations:

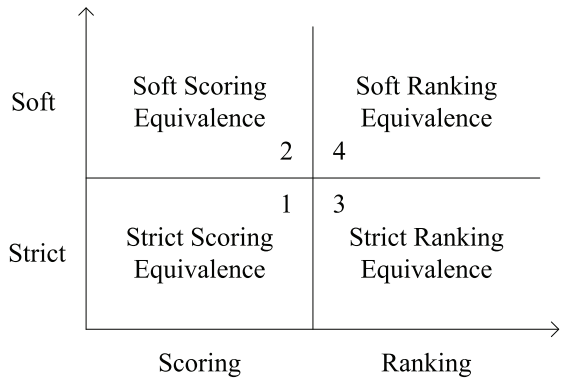


Figure 3.8: Coordinate of equivalences

Based on these four circumstances, we define equivalence with respect to the weightings yielded by PRAEs.

Definition 3.3.2. *Strict Scoring Equivalence*. Two PRA expressions are strictly scoring equivalent if they yield identical tuple scores for the same list of tuples.

Definition 3.3.3. *Soft Scoring Equivalence*. Two PRA expressions are softly scoring equivalent if the tuple scores yielded by them for the same list of tuples satisfy either one of the following conditions: 1) a common factor exists between the two lists of scores; that is, once the scores in one list multiply the common factor then two lists of scores become identical; or 2) the two lists of scores would be identical under a common agreement about the precision of numbers and approximation methods.

Definition 3.3.4. *Strict Ranking Equivalence*. Two PRA expressions are strictly ranking equivalent if they yield the same list of tuples in identical order.

Definition 3.3.5. *Soft Ranking Equivalence*. Two PRA expressions are soft ranking equivalent if they yield the same list of tuples in identical order when approximation is allowed.

With these definitions of scored relation and various scoring or ranking equivalence, now we are able to define the algebraic equivalence for PRA expressions:

Definition 3.3.6. *Algebraic Equivalence of PRA expressions*. Two PRA expressions are strictly algebraic equivalent, written $PRAE_1 \equiv PRAE_2$, if they present the same mappings with identical weightings; that is, when the same relations are substituted for identical names in the two expressions, we get the same result whose tuples are weighted by identical scores.

3.3.3 Ideas and Principles of Design

Previous discussions (see Sections 3.1 and 3.3.1) had motivated us to apply or develop an appropriate method for representing the scoring semantics of PRA expressions.

First of all, we wanted to see if there are existing methods or declarative languages that may provide the expressiveness. Possible candidates include, for instance, the SRAM language introduced in [Cornacchia and de Vries, 2007] which follows comprehension syntax [Buneman et al., 1994], and the SALT algebra in [Chaudhuri et al., 2005] which aims for expressing scoring or ranking functions over lists and text. What was found out is that all of these languages satisfy the requirement of modelling scoring functions. However, although they are inspiring, but they are still impossible to be applied to PRA for the aforementioned purpose of *representing* scoring semantics, this is because:

1. All of these (declarative) languages are relationally complete algebra which are about the same level as PRA in terms of abstraction and expressiveness. It is possible to transform

a PRA expression into one of these languages so that to obtain an indication of its finally implied scoring function, but it is impractical to interpret the scoring semantics of every single steps (or sub-expressions) of a PRA expression with any of these languages.

2. None of these languages distinguish the types of scores. For instance, how to determine if a score is an ordinary weighting or a probability? Knowing the types of scores not only helps to verify the correctness of modelling but also helps to develop rule-based optimization strategies for queries.

3. It is unclear whether these languages can be *manipulated*. Some of these languages, for instance SRAM, are actually translated into SQL to be processed, which means the responsibility of query optimization is bypassed. Although it might be reasonable to rely on underlying DB optimizer, but it is unclear how an optimizer can utilise the knowledge of scoring functions.

Therefore, a carefully designed representation for scoring functions is needed, and to overcome the shortcomings of previous works, the formalisation should have the following characteristics;

- It should be along side with PRA as a complement representation of scoring semantics, but not to be a duplicate or a replacement of PRA.
- It should be expressive for representing scoring functions, including differentiation of types of scores.
- It should be able to be manipulated by query optimizer or providing useful knowledge for optimization.

As discussed, such representation of scoring function is firstly a technique of annotation; and secondly, it is part of a query optimizer, directly or indirectly. The annotating functionality is illustrated in Figure 3.9. As a result, SCX was proposed. While annotating PRA expression, SCXs are affiliated to every operators of an expression (i.e. query plan) to present the scoring functions of an annotated sub-expression, therefore, they contain information such as how scoring functions are built up and how scores are propagated.

In the next section, we discuss the syntax and semantics of SCX.

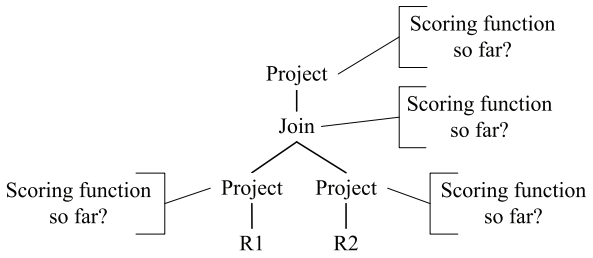


Figure 3.9: Annotating scoring functions of PRA sub-expressions

3.3.4 Syntax and Semantics

In general, a scoring expression is a hybrid of arithmetic-style or logic-style expression, which consists of instant constant, parameter, scoring variables, operators, and (symbols of) functions. Here we introduce separately the parts of SCX and then discuss how these parts are to be assembled. Above all, the syntax of SCX is given in Figure 3.10, and a summary of the semantics of SCX is given in Table 3.1.

3.3.4.1 Instant Constant and Parameter

Both instant constant and parameter are constants, the difference is that instant constants directly appear in SCX as numeric numbers, whereas parameters are names or alias with assigned constant values. In general, to use instant constant or parameter is purely depending on specific situations and convenience.

3.3.4.2 Variable

A scoring variable represents a class of tuple scores that have the same properties, which are indicated by different segments of a variable. A variable consist of one or up to three segments that are separated by dots, and each segment represents one of the properties of the *ownership of scores*, the *aggregation and grouping*, and the *type of scores*. In general, a variable is formulized in the following form:

$$\langle ownership\ of\ score \rangle . \langle schema\ of\ grouping \rangle . \langle type\ of\ score \rangle$$

While the properties are explained as follows:

Ownership of Score First of all, the ownership of score property tells what entity owns score, which can be tuples in a relation, or an accumulator. If the owner is a relation, then the ownership is represented by *relation property*, which consists of a mandatory alias of relation and an

<scx> ::= <geScx> | <inScx>
<geScx> ::= [‘this’ ‘.’ ‘s’ ‘=’] <scoTypeProp> | <eveExpr> | <artExpr>
<inScx> ::= ‘R’ <id> [‘.’ <grpProp>] ‘.’ <scoTypeProp> ‘=’ <artExpr>
<paramAssign> ::= <param> ‘=’ ‘c’ ‘[’ <decimal number> ‘]’

<id> ::= <unsigned integer>
<name> ::= <letter> | <name> (<letter> | <digit> | ‘_’)
<score> ::= <decimal number>
<param> ::= <name>
<relAlias> ::= ‘R’ <id> | ‘this’
<accProp> ::= ‘acc’
<ownProp> ::= <relAlias> [‘[’ <name> ‘]’]
<attrAlias> ::= ‘$’ (<name> | <number> | ‘∗’) | ‘%’ <name> ‘%’
<attrList> ::= <attrAlias> [‘,’ <attrList>]
<grpProp> ::= ‘[’ <attrList> ‘]’
<scoTypeAlias> ::= ‘s’ | ‘w’ | ‘p’ | ‘c’
<scoTypeProp> ::= <sctAlias> [‘[’ <score> ‘]’]
<var> ::= (<ownProp> | <accProp>) [‘.’ <grpProp>] ‘.’ <scoTypeProp>
<artOpr> ::= ‘+’ | ‘−’ | ‘∗’ | ‘/’
<eveOpr> ::= ‘!’ | ‘^’ | ‘v’
<stdFuncOpr> ::= ‘log’ | ‘ln’ | ‘pow’ | ‘exp’
<baseArg> ::= <score> | <var> | <param>
<stdArg> ::= <score> | <var> | <param> | <artExpr>
<stdFunc> ::= <stdFuncOpr> ‘(’ (<stdArg> | <baseArg> ’,’ <stdArg>) ‘)’
<conFunc> ::= ‘(’ (‘<’ | ‘>’) ‘?’ <stdArg> ‘:’ <stdArg> ‘)’
<aggSym> ::= ‘COUNT’ | ‘SUM’ | ‘MAX’ | ‘MIN’ | ‘AVG’
<aggProp> ::= <attrAlias> | ‘[’ <attrList> [‘|’ <attrList>] ‘]’
<aggFunc> ::= <aggSym> ‘(’ [‘DISTINCT’] <relAlias> [‘.’ <aggProp>] ‘)’
<artOperand> ::= <score> | <scoProp> | <var> | <stdFunc> | <conFunc> | <aggFunc>
<artExpr> ::= <artOperand> | ‘(’ <artOperand> <artOpr> <artExpr> ‘)’
<eveExpr> ::= <var> | ‘(’ <eveOpr> <var> ‘)’ | ‘(’ <var> <eveOpr> <eveExpr> ‘)’

<digit> ::= [0-9]
<letter> ::= [a-zA-Z]
<sign> ::= ‘+’ | ‘−’
<unsigned integer> ::= <digit> | <digit> <unsigned integer>
<decimal fraction> ::= ‘.’ <unsigned integer>
<number> ::= [1-9] [<unsigned integer>]
<decimal number> ::= [<sign>] <unsigned integer> [<decimal fraction>]

Figure 3.10: Syntax (BNF) of Scoring Expression (SCX)

Symbol or Expression of SCX	*Remark of Semantics*
(1.1) R2 (1.2) R2.p (1.3) ! R2.p (1.4) R2.s (1.5) ! R2.s (1.6) 'R2.$term' or 'R2.%term%' (1.7) R2[MagColl].$1 (1.8) R2.[$term].p (1.9) R2.[$term].s (1.10) this	(1.1) a relation alias with a unique id '2' (1.2) event probability $P(\tau)$, $\tau \in$ R2 (1.3) the complement of $P(\tau)$, $\forall\tau \in$ R2, $1 - P(\tau)$ (1.4) arbitrary score of tuple τ, $\tau \in$ R2 (1.5) not applicable, throw a semantic error (1.6) an attribute named 'term' of R2 (1.7) the first attribute of R2, R2 named 'MagColl' (1.8) similar to (1.2) but group $P(\tau)$ by $term (1.9) similar to (1.4) but group scores by $term (1.10) a relation itself, (1.1) to (1.9) are applicable

(3.1) ^R2.p
(3.2) ▽R2.p
(3.3) this.s = (acc.s ∗ R2.s)
(3.4) this.s = (1 − (acc.s ∗ (1 − R2.s)))
(3.5) this.s = (acc.s ∗ R2.s)
(3.6) this.s = (acc.s + R2.s)
(3.7) this.s = (<? acc.s : R2.s)
(3.8) this.s = (>? acc.s : R2.s)
	$\exists\tau_i \in$ R2 where $i = 1,\ldots,k$ and $k \leq \lvert R2\rvert$,
	(3.1) $\bigwedge_{i=1}^{k} P(\tau_i)$
	(3.2) $\bigvee_{i=1}^{k} P(\tau_i)$
	(3.3), (3.5), (3.7) are the arithmetic representations of (3.1), applicability depends on assumption
	(3.4), (3.6), (3.8) are the arithmetic representations of (3.2), applicability depends on assumption
	(3.3), (3.4) are applicable if independent
	(3.5), (3.6) are applicable if disjoint
	(3.7), (3.8) are applicable if subsumed
	(3.3) defined as $\prod_{i=1}^{m}$ R2.t$_i$
	(3.4) defined as $1 - \prod_{i=1}^{m}(1 -$ R2.t$_i)$
	(3.5) defined as $\prod_{i=1}^{m}$ R2.t$_i$
	(3.6) defined as $\sum_{i=1}^{m}$ R2.t$_i$
	(3.7) defined as for all given R2.t min{R2.t}
	(3.8) defined as for all given R2.t max{R2.t}
	for (3.3), (3.4), (3.7), initial acc.s = 1
	for (3.5), (3.6), (3.8), initial acc.s = 0

(4.1) R1.t ^ R2.t (4.2) R1.t ▽ R2.t	$\exists\tau_i \in$ R1 $\exists\tau_j \in$ R2, (4.1) $P(\tau_i) \wedge P(\tau_j)$ (4.2) $P(\tau_i) \vee P(\tau_j)$ (3.3) to (3.8) are applicable for similar situations
(5.1) COUNT(R2) (5.2) COUNT(DISTINCT R2) (5.3) COUNT(R2.$term) (5.4) COUNT(R2.[$term, $doc]) (5.5) COUNT(DISTINCT R2.$term) (5.6) COUNT(DISTINCT R2.[$term, $doc]) (5.7) COUNT(R2.[$doc | $term]) (5.8) COUNT(DISTINCT R2.[$doc | $term])	all expressions manipulate on relation R2 (5.1) count the number of tuples (5.2) similar to (5.1) but count distinct tuples (5.3) count the number of values of $term (5.4) similar to (5.3) but count paired values (5.5) count the number of distinct values of $term (5.6) similar to (5.5) but count distinct paired values (5.7) count the number of values of $doc and results are grouped by $term (5.8) similar to (5.7) but count distinct values
aggOpr = 'SUM' | 'MAX' | 'MIN' | 'AVG' (6.1) aggOpr(R2) (6.2) aggOpr(R2.$distance) (6.3) aggOpr(R2.[$price | $cid])	all expressions (try to) manipulate on relation R2 (6.1) not applicable, throw a semantic error (6.2) compute summation, maximum, minimum or average of values of $distance, note that the 'DISTINCT' constraint has no effect (6.3) similar to (6.2) but compute the values of $price and results are grouped by $cid

Table 3.1: Semantics of Scoring Expression

optional name of relation. An alias of relation is either composed of an uppercase 'R' followed by a numerical ID, or just composed of a word 'this' for a relation itself. A name of relation is the same to the name of a table or a view, and it is written between a pair of square brackets following immediately after alias of relation. For example, $R2$ for a relation property without name, or $R2[MagColl]$ or $this[MagColl]$ for a property with name.

In addition, if the owner is an accumulator, then the ownership is represented by *accumulator property*, which is simply composed of an abbreviation 'acc'. The use of accumulator will be discussed later in this section.

Schema of Grouping Secondly, grouping schema indicates how scores are grouped in owner relation. The schema is a list of attribute(s) separated by commas, and the property can be applied while knowing the score is obtained from conventional aggregations such as counting.

There are two ways to formulize an attribute. The first way is to use a dollar sign (i.e. '$') followed by an attribute name (i.e. a term) or a column number, and the second way is to write the attribute name between a pair of percentage signs (i.e. '%'). The difference between the first and the second ways is that the latter allows phrase (or space separated terms) to be used as attribute name. For example, an attribute can be formulized as $\$CarType$, \$2 or $\%Car\ Type\%$.

To formulize grouping schema, we place a comma separated attribute list between a pair of closed square brackets such as $[\$CarType, \$Carmaker]$. Especially, if a grouping is based on all attributes of a relation, then an anonymous formalisation can be applied, which is composed of a dollar with an asterisk, i.e. '$[\$*]$'.

What grouping means is that it indicates relational aggregations (including duplicate removal) have been performed on the given attribute(s), in other words, tuples are firstly grouped by the given attribute(s), and then for each group tuples would be aggregated or duplicates would be removed, while a score would be assigned to the (single) resulting tuple, which could be an aggregated weight (from scoring aggregation on the original tuples scores in the group) or a judiciously specified constant.

Types of Score Finally, the type of score classifies variables into four types based on different characteristics of the values of scores. The four applied types are: unspecified weighting, denoted by 's' for *s*core; normal weighting, denoted by 'w' for *w*eight; probabilistic weighting, denoted by 'p' for *p*robability; and constant weighting, denoted by 'c' for *c*onstant.

In particular, the value of a score can be specified if the score is a constant weighting, in

this case, a value is placed between a closed square brackets and affiliated to the 'c' type. For example, for scores constantly equal one, one can formulize the scores as $c[1.0]$.

To wrap up, variables are operands of SCX, and the properties supply necessary information to formulize score (or probability) estimation and aggregation, which are the bases for presenting and interpreting scoring functions. Finally, we give examples for all three types of variables as follows: 1) to present tuple scores of an unweighted relation, e.g. *MagColl*, one can formulize a variable with constant score 1.0, i.e. $R0[MagColl].c[1.0]$; 2) to present normal weightings grouped by an attribute $\$CarType$, one can formulize a variable as $R2.[\$CarType].w$; 3) to present probabilistic weightings grouped by attributes $\$Term$ and $\$DocId$, one can formulize a variable as $R3.[\$Term, \$DocId].p$.

3.3.4.3 Operators

There are two types of operators, i.e. *arithmetic operators* and *event operators*. All arithmetic operators are binary operators and they can be used for computations of all types of scores (i.e. the four types of weightings), whereas event operators could be unary or binary and they are only feasible for probabilistic weighting.

Arithmetic Operators Four ordinary arithmetic operators, i.e. 'plus' as '$+$', 'minus' as '$-$', 'multiply' as '$*$', and 'divide' as '$/$', are used, in which corresponding mathematical meanings are applied.

Event Operators Event operators are Boolean logical operators, three operators including logical 'not' as '!' (same as '$\neg$'), logical 'and' as '^' (same as '$\wedge$'), and logical 'or' as '$\vee$' (same as '$\vee$'). For operator '!', it is an unary operator which operand must be a variable with score type as probabilistic weighting; it stands for complement of probability, for instance, $!R2.p$ means $1 - R2.p$. Different from operator '!', both operators '^' and '$\vee$' can have one or two operands, and the exact mappings to arithmetic operations depending on certain probabilistic assumptions. The reasons of having event operators will be discussed later together with event-based expressions.

3.3.4.4 Functions

SCX allows two types of functions, i.e. actual functions and symbolic functions, to be embedded in arithmetic expressions[2], in which actual functions include *standard functions* and *conditional selection*, whereas symbolic functions include *aggregate symbols*. In general, for each type of

[2]For event expressions, they will be eventually interpreted into arithmetic expressions.

functions, three aspects will be discussed: 1) why a type of functions is included in SCX, 2) what argument(s) is/are taken or allowed to be taken by functions, and 3) the semantics of specific functions.

Standard Functions We have been being cautious to introduce standard mathematical functions into SCX. The reasons are, first, SCX was designed for representing purpose of scoring semantics of PRA expressions, so the expressiveness of SCX should be just enough for this aim; and secondly, the names of some standard functions such as 'max' and 'min' are overlapped with (DB's) aggregate functions, so in order not to introduce unnecessary ambiguity these functions are not included in standard functions but have been classified into another type, i.e. conditional selection.

Currently there are four functions in this class, which are logarithm as 'log', natural logarithm as 'ln', power as 'pow', and exponentiation as 'exp'. Moreover, two kinds of arguments are allowed: value of score and variable of scores. For functions 'log' and 'pow' two arguments are expected: the first as base and the second as parameter. For instance, $\log(2, R1.w)$ stands for $\log_2 R1.w$ and $\text{pow}(R2.w, 2)$ means $R2.w^2$. While for functions 'ln' and 'exp' one argument is expected for parameter.

Conditional Selection As aforementioned, conditional selection functions are applied to act as the 'max' and 'min' standard functions and to avoid naming overlapping with aggregate functions. The syntax of conditional selection looks like $\langle predicate \rangle ? \langle firstArg \rangle : \langle secondArg \rangle$, where predicate is either 'less than' (i.e. '$<$') or 'greater than' (i.e. '$>$'), and allowed arguments include variable of scores and unary event expressions taking variable as operand.

Readers who are familiar with C programming language may find that it is very similar to the conditional evaluation or assignment statement in C, in fact, it is taken from the C syntax but slightly modified. The conditional selection means comparing the first argument to the second argument with a given predicate, i.e. less than or greater than, then take the first argument if the predicate is true or take the second if it is false. For example, $>? R1.w : R2.w$ stands for if $R1.w > R2.w$ then take $R1.w$ otherwise take $R2.w$.

Aggregate Symbols Aggregate symbols are symbolic functions, which do not actually compute or manipulate scores at all, so they are only 'annotations' of corresponding aggregate functions such as counting or summing. There are five aggregate symbols in SCX, which are: 'COUNT' denotes counting, 'SUM' denotes summing, 'MAX' denotes maximising, 'MIN' denotes min-

imising and 'AVG' denotes averaging. In general, aggregations are bridges between conventional relations and weighted (including probabilistic) relations, hence the aggregate symbols are used to represent probability estimations (see Section 2.5.2) in conventional aggregations. In other words, they indicate how scores are generated from counting or computing on tuples or attribute values of relations.

In general, the formulation of aggregate symbols is given as follow:

$$\langle aggregation \rangle (\langle constraint \rangle \ \langle relation\ alias \rangle . [\langle target\ list \rangle \mid \langle group\ list \rangle])$$

in which *aggregation* is one of the five aggregate symbols; and the only *constraint*, which is 'DISTINCT' constraint, is optional; while *relation alias* is similar to the ownership property of scoring variable, which indicates where aggregated tuples come from; and *target list* is an attribute(s) list that is varied to different aggregations, which will be discussed later; similarly, *group list* is also an attribute(s) list, but it indicates a grouping operation which is comparable to the function of 'GROUP BY' statement in SQL.

Depending on whether scores are computed by counting or by mathematical operations, the five symbols can be categorised into two groups, where 'COUNT' is in a group with itself, and the other aggregations are in another.

Firstly, the 'COUNT' symbol indicates scores are computed by counting on tuples or on attribute values, in which obtained score(s) might be grouped or ungrouped by certain attribute values. While 'DISTINCT' constraint is not applied, the result of counting tuples is equivalent to the result of counting attribute values. For instance, $COUNT(R2.\$term)$ means the same to $COUNT(R2.\$*)$, and similarly, $COUNT(R2.[\$Term \mid \$DocId])$ stands for the same to $COUNT(R2.[\$* \mid \$DocId])$. Specially, to count the total number of tuples in a relation, the attribute lists (including target list and group list) can be save, i.e. one can just write $COUNT(R2)$.

Next, calculation-based aggregations indicate taking attribute values for the production of scores. Therefore, a target list containing a single attribute has to be provided, which data type must be numeric. For example, $SUM(R2.[\$Price])$ or $SUM(R2.[\$Cost \mid \$CarModel])$.

It is important to note that one can always find compatible SQL statements to certain formalisation of aggregate functions in SCX.

3.3.4.5 Expressions

In SCX, there are two types of formulations that can be applied to express scoring functions. Depending on whether source relations are probabilistic or not, a formulation may base on *arithmetic-style expression* or *logic-style expression*. In addition, while collaborating SCX with PRA expressions to demonstrate scoring semantics, for each sub-expression, an 'intuitive' SCX is generated at first, such SCX is called *generated SCX*; and then a 'heuristic' SCX is produced bases on generated SCX and other supplied knowledge, and this SCX is call *interpreted SCX*. A syntax of SCX complying with Backus Naur Form (BNF) is given in Figure 3.10.

Arithmetic-style Expression The most commonly used expressions are arithmetic-style expressions. To get started, we demonstrate the SCXs of three IR models, including basic *tf-idf*, BM25 and LM (see Section 2.2.2), with a toy *MagColl* relation (see Section 2.5.2 and Table 2.3).

Example 3.3.1. Basic *tf-idf* (see Formula 2.1, page 32)

Assuming there are two views (i.e. intermediate relations) named $P\_C\_t\_d$ and df_t which tuples contain scores associated respectively to within-document *tf* and document frequency. Let $R3$ be alias of $P\_C\_t\_d$ and $R7$ be alias of df_t, the following arithmetic expression presents Formula 2.1 (see page 2.1) in an external form when using scoring expression in self-defined mode:

```
P_C_t_d * log(1 / df_t)
```

whereas an internal arithmetic expression would be generated by an optimizer as the following form:

```
R3[P_C_t_d].[$Term, $DocId].p * log(1 / R7[df_t].[$Term].w)
```

This expression tells:

- Tuple scores from $R3$ are probabilistic weightings which have been grouped by attributes $\$Term$ and $\$DocId$, which means the scores are obtained from aggregation;
- Tuple scores from $R7$ are normal weightings which have been grouped by attribute $\$Term$, which also indicates aggregation has been used to compute the scores;
- Scores from $R3$ are to be multiplied to negative logarithm of scores from $R7$.

Example 3.3.2. Saturation function of BM25 (see Formula 2.5, page 33)

In BM25, the within-document *tf* is to be normalised by document length, which is obtain from a view named *DocLen*. In addition, assuming the average document length is given by a view named *AvgDL*. Apart from variables, BM25 introduces k and b parameters to adjust the levering effect of normalisation, which could be also reflected in a SCX. Here three variables are included in a SCX, which means corresponding relational algebra may involve three-way (or multiway) join (see e.g. [Ullman, 1989], Chapter 11, Section 11.8).[3] The following arithmetic expression in external mode demonstrates the saturation function:

```
(n_L_t_d * (k1 + 1)) / (n_L_t_d + (k1 * (1 - (b + (b * (DocLen / AvgDL)) )) ))
```

While similarly, the following internal arithmetic expression could be generated by optimizer:

```
(R3[n_L_t_d].[$Term, $DocId].p * (k1 + 1)) / (R3[n_L_t_d].[$Term, $DocId].p
  + (k1 * (1 - (b + (b * (R5[DocLen].[$DocId].w / R9[AvgDL].w)) )) ))
```

Readers may refer to Example 3.3.1 for the meanings of scoring variables. Besides, $k1$ and b correspond to BM25 parameters (assignments are not shown).

Example 3.3.3. Linear Mixture of Language Modelling (see Formula 2.12, page 34)

The following external scoring expression formulizes the linear mixture of language modelling:

```
lambda * P_C_t + (1 - lambda) * P_C_t_d
```

While the following internal arithmetic expression could be generated by optimizer:

```
(lambda * R3[P_C_t_d].[$Term, $DocId].p) + ((1 - lambda) * R6[P_C_t].[$Term].p)
```

Where *lambda* is LM's parameter, and relation *P_C_t* provides tuple scores of within-collection *tf*.

As they have been shown, arithmetic-style expressions not only look similarly to mathematical expressions but also comply with straightforward mathematical semantics, which on the one

[3] For relational algebra do not include multiway join, BM25 can be implemented by multiple two-way joins, which can be interpreted by SCX that includes multiple aggregate functions.

hand, are readable and understandable by human users, while on the other hand, are able to be processed by programs for calculating scores.

Furthermore, arithmetic-style SCX can present various aggregate expressions along with a special variable: accumulator (i.e. 'acc'). Based on recursion, aggregations for summation, multiplication, maximisation and minimisation are given as follows.

Definition 3.3.7. *Aggregate summation of scores*, denoted as $\sum()$ and formulized in SCX as $acc.s + R.s$, is the addition of a set of scores; the result is their sum or total. The SCX formulation means the result is obtained from recursive addition.

$$\begin{array}{rcl} \sum() & = & acc.s + R.s \\ & =_{\text{def}} & \left(\begin{array}{l} acc.s = 0 \\ acc.s = acc.s + R.s \end{array} \right) \end{array}$$

Be aware of the '$+$' operator not only compute the sum of two operands, but also inputs the result to $acc.s$.

Definition 3.3.8. *Aggregate multiplication of scores*, denoted as $\prod()$ and formulized in SCX as $acc.s * R.s$, is to scale integer 1 by a set of numbers (scores); the result is their product. The SCX formulation means the result is obtained from recursive multiplication:

$$\begin{array}{rcl} \prod() & = & acc.s * R.s \\ & =_{\text{def}} & \left(\begin{array}{l} acc.s = 1 \\ acc.s = acc.s * R.s \end{array} \right) \end{array}$$

Similarly, the '$*$' operator not only compute the multiplication of two operands, but also inputs the result to $acc.s$.

Definition 3.3.9. *Aggregate maximisation of scores*, denoted as $\max()$ and formulized in SCX as $>?\ acc.s : R.s$, is to select the maximum score of a set of scores; the result is their maximum. The SCX formulation means the result is obtained from recursive maximisation:

$$\begin{array}{rcl} \max() & = & >?\ acc.s : R.s \\ & =_{\text{def}} & \left(\begin{array}{l} acc.s = R.s \\ acc.s = (>?acc.s : R.s) \end{array} \right) \end{array}$$

Definition 3.3.10. *Aggregate minimisation of scores*, denoted as min() and formulized in SCX as $<?\ acc.s : R.s$, is to select the minimum score of a set of scores; the result is their minimum. The SCX formulation means the result is obtained from recursive minimisation:

$$\begin{aligned} \min() &= <?\ acc.s : R.s \\ &=_{\text{def}} \begin{pmatrix} acc.s = R.s \\ acc.s = (<?\ acc.s : R.s) \end{pmatrix} \end{aligned}$$

Be aware of the difference between aggregate expressions and aggregate symbols: the former are aggregations specific for tuple scores, whereas the latter represents (conventionally) relational aggregations on tuples or attribute values. Nevertheless, there are connections could be drawn between the two kind of aggregations, which will be discussed later in the section.

Logic-style Expression The second form of SCX is logic-style expression. A logic-style SCX cannot be used for direct computation of score, but it indicates certain scoring function for probabilistic events while probabilistic assumption is missing or has not yet been considered, therefore, the scoring variables of logic-style SCX must be probabilistic weightings.

The reason of using logic-style expression is to accurately reflect the intensional semantics of PRA expressions for conjunctive queries which apply distributive law for aggregating probabilistic events. For example, for independent events A, B and C, event expressions for conjunctive queries in the form of $A \wedge (B \vee C)$ but would be processed as $(A \vee B) \wedge (A \vee C)$, which applies intensional evaluation for processing event expressions (see Section 2.4). On the other hand, to meet requirements of efficiency and scalability, probabilistic databases process queries basing on extensional semantics, where pipelined processing or data (tuples) streaming is deployed. However, extensional evaluation does not comply to intensional semantics (which is carried by event expressions) while aggregating scores so that incorrect scores are to be yielded.

As a result, logic-style expression is utilised to tackle the incompatible problem between intentional semantics and extensional evaluation. How does it work would be discussed later in this section, and here an example is given for illustrating logic-style SCX:

Example 3.3.4. Conjunctive query

Assuming both $R1$ and $R2$ representing intensional relations that contain probabilistic scores, a logic-style SCX for a conjunctive query is demonstrated as follow:

```
v(R1.p ^ R2.p)
```

Here $R1.p$ and $R2.p$ are event probabilities of tuples of $R1$ and $R2$ respectively. The expression says to aggregate conjunctive events made from tuples of $R1$ and $R2$.

3.3.5 Generated SCX and Interpreted SCX

Before we explain how to use SCX to present scoring functions of PRA, let us first discuss *generated SCX* and *interpreted SCX*. As mentioned in the previous section, generated SCX intuitively explains scoring semantics that bases on extensional semantics; whereas *interpreted SCX* is an alternative representation while considering intensional semantics and propagated information including aggregations.

One of the main features of PRA is its capability for internal probability estimations for IR models, which is achieved by deploying a composed operator, the relational Bayes (see 2.5.2.2 and 2.5.2.3). On the other hand, it is possible to implement similar functionality (i.e. internal probability estimations) in traditional SQL, for example, by combining aggregate functions, arithmetic functions, and joins (e.g. see [Grossman et al., 1997, Grossman and Frieder, 2004]). Especially, counting based function is the basis for obtaining various frequencies needed by probability estimations.

Theoretically, counting is implemented as a counter, which includes an accumulator and an iterator, and a counting process usually consists of following two steps. Step one, initialisation: the accumulator is set to zero when counting is started, and the iterator is placed at the starting position (of a list of tuples). Step two, iteration: the iterator is moved one step forward to the next position, while the accumulator is increased by one; repeatedly moving iterator and increasing accumulator until the iterator is moved forward from the last tuple. Therefore, the accumulator records how many steps that the iterator has been moved, which is equivalent to aggregately sum up a set of tuple scores that all are constant ones.

In order to relate counting functions to scoring aggregations, the following theorems are introduced. For the convenience of discussion, descriptions of certain patterns of variables and their remarks of indication are given in Table 3.2.

Theorem 3.3.1. *Let R be an unweighted relation, then the result of counting for the total number of tuples in the relation is equal to the result of aggregate summation for an accumulative score by given tuple scores of R with constant-ones.*

Pattern of variable	*Description*	*Remark of Indication*
$R.c[1.0]$	tuple score of R with constant-one	for all tuples in R which tuple scores are all constant-ones
$R.w$, or $R.p$	tuple score of R	for all tuples in R which tuple scores are arbitrary
$R.[X].c[1.0]$	grouped tuple score with constant-one based on X of R	duplicate removal has been performed on R based on attribute(s) X, which results the tuple scores of remained distinct tuples to be constant-ones
$R.[X].w$, or $R.[X].p$	grouped tuple score based on X of R	an (relational) aggregate function has been performed on R based on attribute(s) X, which results the tuple scores of remained distinct tuples to be aggregated scores
$acc.w$	accumulative score	a score is to be accumulated within an accumulator
$acc.[X].c[1.0]$	grouped-and-accumulative score with constant-one based on X	scores are produced for duplicate removal based on X, which results $R.[X].c[1.0]$ eventually
$acc.[X].w$, or $acc.[X].p$	grouped-and-accumulative score based on X	scores are grouped and accumulated based on X, which results $R.[X].w$ or $R.[X].p$ eventually

Table 3.2: Patterns of variables, descriptions, and remarks of indications

$$\begin{aligned} COUNT(R) &= acc.w + R.c[1.0] \\ &=_{def} \begin{pmatrix} acc.w = 0 \\ acc.w = acc.w + R.c[1.0] \end{pmatrix} \end{aligned}$$

Proof. Let $|R|$ be its cardinality, then a count of the number of tuples in R equals to $\sum_1^{|R|} 1.0$; let s be an arbitrary score of the tuple in R, then aggregate summation of all tuples scores equals $\sum_{i=1}^{|R|} s_i$. Let us assume $\sum_{i=1}^{|R|} s_i \neq \sum_1^{|R|} 1.0$, then it must exist at least one tuple score s_i where $s_i \neq 1.0$, but this is impossible according to the premise of Theorem 3.3.1, which says R is unweighted so that all of its tuple scores are deemed to be constant-ones. As a result, there are no inequations are hold for all s where $\sum_{i=1}^{|R|} s_i \neq \sum_1^{|R|} 1.0$. Thus, Theorem 3.3.1 is sound. □

Lemma 3.3.2. *Let R be an unweighted relation, then the result of counting for the number of values of attribute(s) X in the relation without given grouping constraint is equal to the result of aggregate summation for an accumulative score by given tuple scores of R with constant-ones.*

$$\begin{aligned} COUNT(R.[X]) &= acc.w + R.c[1.0] \\ &=_{def} \begin{pmatrix} acc.w = 0 \\ acc.w = acc.w + R.c[1.0] \end{pmatrix} \end{aligned}$$

Proof. While grouping constraint is given, counting for the number of values of attribute(s) X yields the same result as counting for the number of tuples in the relation. Therefore, according to Theorem 3.3.1, Lemma 3.3.2 is sound. □

Theorem 3.3.3. *Let R be an unweighted relation, then the results of counting for the number of values of attribute(s) X_a with given grouping constraint of attribute(s) X_b is equal to the results of aggregate summation for grouped-and-accumulative scores based on X_b by given tuple scores of R with constant-ones.*

$$\begin{aligned} COUNT(R.[X_a|X_b]) &= acc.[X_b].w + R.c[1.0] \\ &=_{def} \begin{pmatrix} acc.[X_b].w = 0 \\ acc.[X_b].w = acc.[X_b].w + R.c[1.0] \end{pmatrix} \end{aligned}$$

Proof. Let R to be partitioned into n groups (i.e. sub-relations) G_k based on the (distinct) values of attribute(s) X_b, let $|G_k|$ be the cardinality of the k^{th} group, where $k = 1, \ldots, n$, and $\bigcap_{k=1}^{n} G_k = \emptyset$, and $\bigcup_{k=1}^{n} G_k = R$. Therefore, within each group of all groups, according to Theorem 3.3.1, we have $\sum_{i=1}^{|G_k|} s_i = \sum_{1}^{|G_k|} 1.0$, where s_i is the i^{th} tuple score in a group; moreover, according to Lemma 3.3.2, we conclude that $COUNT(G_k.[X_a]) = acc.[X_b].w + G_k.c[1.0]$, where a grouped-and-accumulative score $acc.[X_b].w$ accumulates a tuple score specific for group X_b. Because this equation is held for all groups, thus Theorem 3.3.3 is sound. □

Theorem 3.3.4. *Let R be an arbitrary relation, then the result of counting for the number of distinct values of attribute(s) X without given grouping constraint is equal to the result of aggregate summation for an accumulative score by given grouped tuple scores with constant-ones based on X of R.*

$$\begin{aligned} COUNT(DISTINCT\ R.[X]) &= acc.w + R.[X].c[1.0] \\ &=_{def} \begin{pmatrix} acc.w = 0 \\ acc.w = acc.w + R.[X].c[1.0] \end{pmatrix} \end{aligned}$$

Proof. Let R_{X_b} be an intensional relation consists of distinct values of attribute(s) X_b, and let $|R_{X_b}|$ be the cardinality of distinct values of attribute(s) X. In other words, a count of distinct values of attribute(s) X yields $|R_X|$. According to Table 3.2, $R.[X].c[1.0]$ indicates that duplicate removals are performed to remove redundant values of attribute(s) X, while constant-ones are assigned as tuple scores to the results (tuples) of removals. As a result, the aggregate summation computes $\sum_1^{|R_X|} 1.0$, which is equal to $|R_X|$. Thus, Theorem 3.3.4 is sound. □

Theorem 3.3.5. *Let R be an arbitrary relation, then the result of counting for the number of distinct values of attribute(s) X_a with given grouping constraint of attribute(s) X_b is equal to the result of aggregate summation for grouped-and-accumulative scores based on X_b by given grouped tuple scores with constant-ones based on X_a of R.*

$$\begin{aligned} COUNT(DISTINCT\ R.[X_a|X_b]) \quad &= \quad acc.[X_b].w + R.[X_a].c[1.0] \\ &=_{def} \quad \left(\begin{array}{l} acc.[X_b].w = 0 \\ acc.[X_b].w = acc.[X_b].w + R.[X_a].c[1.0] \end{array} \right) \end{aligned}$$

Proof. Let R_{X_b} be an intensional relation consists of distinct values of attribute(s) X_b, and let R to be partitioned into n groups (i.e. sub-relations) $G_{k_{X_b}}$ based on the (distinct) values of attribute(s) X_b, while let $|G_{k_{X_b}}|$ be the cardinality of distinct values of attribute(s) X in the k^{th} group, where $k = 1, \ldots, n$, and $\bigcap_{k=1}^{n} G_{k_{X_b}} = \emptyset$, and $\bigcup_{k=1}^{n} G_{k_{X_b}} = R_{X_b}$. Therefore, within each group of all groups, according to Theorem 3.3.4, we have $COUNT(DISTINCT\ G_{k_{X_b}}.[X_a]) = acc.[X_b].w + G_{k_{X_b}}.c[1.0]$, where a grouped-and-accumulative score $acc.[X_b].w$ accumulates a tuple score specific for group X_b. Because this equation is held for all groups, thus Theorem 3.3.5 is sound. □

While presenting scoring aggregations of PRA, SCXs are generated base on formulations of recursive accumulation, i.e. with accumulator. By applying the above Theorems, generated SCXs can be interpreted by SCXs employing relational aggregate functions. For example, the arithmetic-style SCX in Example 3.3.1 can be interpreted by the following interpreted SCX:

```
(COUNT(R0.[$* | $Term, $DocId]) / COUNT(R0.[$Term | $DocId]))
  * log((1 / (COUNT(DISTINCT R0.[$Term, $DocId | $Term])
  / COUNT(DISTINCT R0.[$DocId])) ))
```

while the score of average document length, i.e. $R9[AvgDL].w$, in Example 3.3.2 can be presented by an interpreted SCX as follow:

```
(R3[n_L_t_d].[$Term, $DocId].p * (k1 + 1)) / (R3[n_L_t_d].[$Term, $DocId].p
  + (k1 * (1 - (b + (b * (R7[DocLen].[$DocId].w
  / (COUNT(R0) / COUNT(DISTINCT R0.[$DocId])) )) )) ))
```

In short, the purpose of interpreting accumulator-based SCXs by aggregate-function-based SCXs is to supply information for indexes selection in scoring-drive optimization, in which more details will be discussed later in the rest of this chapter.

3.4 Scoring-Driven Optimization

So far, what have been discussed about SCX mainly focus on its logical designs including concepts, syntax, and semantics. From now on, we start to introduce how SCX is to be employed in the optimization of query processing of PRA expressions. The technique is called scoring-driven optimization, which utilises scoring expressions to *direct* query processing techniques such as logical-physical mapping of PRA operations and exploitation of indexes.

In this section, the discussions would be focused on the following issues: 1) how to SCX is generated for PRA; 2) how SCX can be manipulated and transformed; 3) how scoring semantics can be analysed; 3) how scoring semantics of PRA expressions are interpreted; and 4) how to direct query processing after scoring semantics has been acquired.

3.4.1 Generating SCX for PRA

Above all, let us discuss how SCXs are generated and associated to PRA operators. As previous discussions in Section 2.4.2 and Section 2.5.2.2, PRA extends traditional relational algebra by incorporating probability aggregations and estimations within its operators, in order words, scoring functions (for probability aggregations and estimations) are *implied* and applied while relational operations are performed. Therefore, SCXs are generated to articulate the implied scoring functions of PRA operations.

Generating SCX for PRA means to produce scoring expressions based on pre-defined semantics, and then each generated SCX is mounted to (or mapped to) a particular PRA operator. Since PRA supports probability estimations from traditional non-probabilistic (or unweighted) relations, hence the generation of SCX is addressed according to the characteristics of input

relation(s) taken by PRA operators, where three kinds of inputs will be discussed including unweighted input, weighted but non-probabilistic input and probabilistic input. For each type of input, generated SCX(s) is/are assigned to an associated PRA operator where difference may be applied depending on a given (probabilistic) assumption.

First, the generated SCXs for PRA operators taking unweighted relation(s) are given in Table 3.3, in which example relations are employed for demonstration purposes only[4].

PRA Expression	*Assumption*	*Generated SCX*
MagColl	N/A	c[1.0]
SELECT [$1='car'](MagColl)	N/A	R0[MagColl].c[1.0]
PROJECT *assumption* (MagColl)	N/A	R0[MagColl].c[1.0]
	DISTINCT	acc.[$*].c[1.0] + R0[MagColl].c[1.0]
	SUM	acc.[$*].w+R0[MagColl].c[1.0]
BAYES *assumption* [](MagColl)	N/A or DISJOINT	R0[MagColl].c[1.0] / (acc.w + R0[MagColl].c[1.0])
BAYES *assumption* [$doc](MagColl)	N/A or DISJOINT	R0[MagColl].c[1.0] / (acc.[$doc].w+R0[MagColl].c[1.0])
JOIN *assumption* [$1=$1](QTerm, MagColl)	N/A	c[1.0]
UNITE *assumption* (MagColl, BookColl)	N/A or DISTINCT or ALL	c[1.0]
SUBTRACT *assumption* (MagColl, BookColl)	N/A	c[1.0]

Table 3.3: Generated SCX for PRA operators with unweighted input(s)

Here, a convention is applied for articulating unweighted relations, which is that the tuple scores of unweighted relations are always constant-ones. For instance, given a relation named *MagColl*, then its generated SCX is $c[1.0]$, in which c is its scoring type, and $c[1.0]$ means scores are 1.0. Note that the generated SCX for a relation does not require alias of ownership, which means the relation itself owns the scores.

Next, the generated SCX for Select operator is very simple, assuming $R0$ is the relation alias of *MagColl*, then tuple scores of selection is $R0[MagColl].c[1.0]$. The only difference is the ownership of scores is specified, so other users know that the scores can be taken from *MagColl*.

Then for Project operator, generated SCXs depend on a given assumption. While there is no assumption, then the SCX for projection is the same to selection. Whereas if the assumption is

[4]E.g. *MagColl* for magazine collection, *BookColl* for book collection, and *QTerm* for query terms, the same convention is applied in later discussions unless further clarified.

distinct, which means the projection also performs duplicate removal of tuples, in consequence, the tuple scores of results should be constant-ones. To articulate duplicate removal, a generated SCX is given as $acc.[\$*].c[1.0] + R0[MagColl].c[1.0]$, where an accumulator groups tuple scores by \$* [5] and a constant-one is assign to each group (also see Section 3.3.5). Furthermore, if an assumption SUM is given, then scores are aggregated in groups, which the generated SCX is given by $acc.[\$*].w + R0[MagColl].c[1.0]$. It is similar to duplicate removal, but the property of type of score is replaced by w, which shows scores are obtained from aggregations instead of pre-set constants.

And then for Bayes operator, we separate the situations where evidence key(s) (see Section 2.5.2.2) is/are provided or not. In general, Bayes operator implies division is involved. For cases where evidence key(s) are ignored (i.e. not given), then Bayes estimates probabilities for tuple scores based on whole relation, in which the denominator of division is obtained by accumulating scores without grouping preference, for example, $R0[MagColl].c[1.0]/(acc.w + R0[MagColl].c[1.0])$. On the other hand, if evidence key(s) are provided, then Bayes estimates probabilities based on the values of evidence, where group-based denominators are accumulated and applied for division, for another example, $R0[MagColl].c[1.0]/(acc.[\$doc].w + R0[MagColl].c[1.0])$, where attribute $\$doc$ is an evidence key. It is worth noting that a default assumption, i.e. *disjoint*, for probability estimation is applied.

At last, for the three binary operators of PRA, i.e. Join, Unite and Subtract, their generated SCXs are the same if their dual inputs are both unweighted relations, which are simply articulated as $c[1.0]$. Because these PRA operators yield results which tuple scores can be traced elsewhere, so the views (i.e. intermediate relations) they are generating are the owners of scores.

Second, the generated SCXs for PRA operators taking ordinarily weighted relation(s) are given in Table 3.4.

As they are illustrated, the generated SCXs for relation itself, Select, Project and Bayes operators are very similar to the counterparts for unweighted relation, in which changes are reflected from input tuples scores, i.e. scores of constant-ones are replaced by normal weightings w, while accumulators for aggregations remain the same.

However, changes are more significant for binary operators. For Join, tuple scores are represented by arithmetic multiplication as $R1[QTerm].w * R2[wColl].w$. Whereas for Unite, scor-

[5] Grouping attribute list changes accordingly to projecting attribute(s).

PRA Expression	*Assumption*	*Generated SCX*
MagColl	N/A	w
SELECT [$1='car'](wColl)	N/A	R1.w
PROJECT *assumption* (wColl)	N/A	R1.w
	DISTINCT	acc.[$*].c[1.0] + R1.w
	SUM	acc.[$*].w + R1.w
BAYES *assumption* [](wColl)	N/A or DISJOINT	R1.w / (acc.w + R1.w)
BAYES *assumption* [$doc](wColl)	N/A or DISJOINT	R1.w / (acc.[$doc].w + R1.w)
JOIN *assumption* [$1=$1](QTerm, wColl)	N/A	R1[QTerm].w * R2[wColl].w
UNITE *assumption* (wColl1, wColl2)	N/A or DISTINCT	c[1.0]
	ALL	w
	SUM	(acc.[$*].w + R1.w) + (acc.[$*].w + R2.w)
SUBTRACT *assumption* (wColl1, wColl2)	N/A	R1[wColl1].w − R2[wColl2].w

Table 3.4: Generated SCX for PRA operators with weighted but non-probabilistic input(s)

ing expressions are based on assumption: while no assumption is given, *distinct* is deployed as default, and pre-set constant scores $c[1.0]$ is applied (i.e. the same to the situation of unweighted relation); while assumption *all* is used, then its generated SCX is w; and if assumption *sum* is employed, then accumulators are used for indicating aggregations, where a SCX is generated as $(acc.[\$*].w + R1.w) + (acc.[\$*].w + R2.w)$, which means an aggregation is separated into two steps: firstly, accumulations are applied to either input respectively, and then an addition is performed to finalise the aggregation. Moreover for Subtract, a SCX is given as $R1[wColl1].w - R2[wColl2].w$ according to the definition of the operator.

Third and final, the generated SCXs taking probabilistic relation(s) are given in Table 3.5, which demonstrates how probabilities are aggregated with PRA operators by given probabilistic assumptions.

Initially, SCXs for probabilistic relations and selections performed upon probabilistic relations are similar to their counterparts for normally weighting relations, for instance, a relation is articulated as p, and a selection is formulized as $R0[MagColl].p$. While for other PRA operators, their generated SCXs are a bit more complicated, because these operators are applied either for probability aggregations or for probability estimations. In both situations, generated SCXs could be presented in either event (logic-style) expressions or arithmetic expressions (with probabilistic

assumptions given); in cases where probabilistic assumptions have not been explicated, default assumptions would be applied.

PRA Expression	*Assumption*	*Generated SCX*
MagColl	N/A	p
SELECT [$1='car'](MagColl)	N/A	R0[MagColl].p
PROJECT *assumption* (MagColl)	N/A	$\vee$ R0[MagColl].[$*].p
	INDEPENDENT	$1 - (\text{acc.[\$*].p} * (1 - \text{R0.p}))$
	DISJOINT	acc.[$*].p + R0.p
	SUBSUMED	$>?$ acc.[$*].p : R0.p
BAYES *assumption* [](MagColl)	N/A	R0[MagColl].p / ($\vee$ R0[MagColl].p)
	INDEPENDENT	$\text{R0.p} / (1 - (\text{acc.p} * (1 - \text{R0.p})))$
	DISJOINT	R0.p / (acc.p + R0.p)
	SUBSUMED	R0.p / ($>?$ acc.p : R0.p)
BAYES *assumption* [$doc](MagColl)	N/A	R0[MagColl].p / ($\vee$ R0[MagColl].[$doc].p)
	INDEPENDENT	$\text{R0.p} / (1-(\text{acc.[\$doc].p} * (1-\text{R0.p})))$
	DISJOINT	R0.p / (acc.[$doc].p + R0.p)
	SUBSUMED	R0.p / ($>?$ acc.[$doc].p : R0.p)
JOIN *assumption* [$1=$1](QTerm, MagColl)	N/A	R1[QTerm].p ^ R0[MagColl].p
	INDEPENDENT	$\text{R1.p} * \text{R0.p}$
	DISJOINT	c[0.0]
	SUBSUMED	$<?$ R1.p : R0.p
UNITE *assumption* (MagColl, BookColl)	N/A	R0[MagColl].[$*].p $\vee$ R2[BookColl].[$*].p
	INDEPENDENT	$1 - ((\vee \text{R0.[\$*].p}) * (\vee \text{R2.[\$*].p}))$
	DISJOINT	$(\vee \text{R0.[\$*].p}) + (\vee \text{R2.[\$*].p})$
	SUBSUMED	$>?$ ($\vee$ R0.[$*].p) : ($\vee$ R2.[$*].p)
SUBTRACT *assumption* (MagColl, BookColl)	N/A	R0[MagColl].p ^ (!R2[BookColl].p)
	INDEPENDENT	$\text{R0.p} * (1 - \text{R2.p})$
	DISJOINT	R0.p
	SUBSUMED	R0.p − R2.p *(if R0.p > R2.p)*
		c[0.0] *(if R0.p ≤ R2.p)*

Table 3.5: Generated SCX for table or PRA operators with probabilistic input(s)

Firstly, projection is the most commonly used aggregate function for probabilistically weighted tuples, which logic-style SCX is written as, for example, $\vee$ $R0[MagColl].[\$*].p$[6]. All arithmetic-style SCXs for Project employ accumulators, respectively, SCXs are generated based on Project's definition, which are, for instance, $1 - (acc.[\$*].p * (1 - R0.p))$ for independent events, and $acc.[\$*].p + R0.p$ for disjoint events, and $>?\ acc.[\$*].p : R0.p$ (i.e. taking the maximum probability) for subsumed events. In default, assumption *disjoint* would be applied.

Moreover, while estimating probabilities upon probabilistic relations, Bayes is applied, which

[6]Similarly, grouping attribute list changes accordingly to projecting attribute(s).

logic-style SCXs are given by, for example, $R0[MagColl].p/(\vee\, R0[MagColl].p)$ where evidence key(s) is/are not given and $R0[MagColl].p/(\vee\, R0[MagColl].[\$doc].p)$ in which an evidence key is available. For arithmetic-style expressions based on specific assumptions, different formulations are used for denominator while the formulations of numerator remain the same. Be aware that Bayes is a composed operator of projections, hence the denominator (of generated SCX) could be articulated as aggregations as those in projection's. Similarly, *disjoint* is used as the default assumption.

Next, Join is used for computing the probabilities of conjunctive events, which a logic-style SCX is formulized as e.g. $R1[QTerm].p \wedge R0[MagColl].p$. In default, conjunctive events are considered to be obtained from independent events, where a joint probability is computed by multiplication of independent events, which is, e.g. in arithmetic-style SCX, $R1.p * R0.p$. Because the conjunction of disjoint events yields zero probability, hence while articulated in SCX, a constant-zero is applied, i.e. $c[0.0]$. In addition, the joint probability of subsumed events is to take the minimum probability from all event, so that the SCX for subsumption is generated as $<?\, R1.p : R0.p$.

Furthermore, since Unite (i.e. union) is used for aggregating two probabilistic relations, for example *MagColl* and *BookColl*, so that its generated SCX in logic-style is written as $R0[MagColl].[\$*].p \vee R2[BookColl].[\$*].p$: the grouping properties appeared in both inputs should be identical, which indicate the aggregated scores should be obtained firstly based on either relation (e.g. by deploying projections), and then a second aggregation yields the final score for a united tuple. Here generated SCXs could be presented in a mixed style of logic-based and arithmetic-based expressions for assumption specific situations. For instance, $1 - ((\vee\, R0.[\$*].p) * (\vee\, R2.[\$*].p))$ for *independent* assumption, while $(\vee\, R0.[\$*].p) + (\vee\, R2.[\$*].p)$ for *disjoint* assumption, and $>?\, (\vee\, R0.[\$*].p) : (\vee\, R2.[\$*].p)$ for *subsumed* assumption.

Finally, the generated SCX for Subtract, which is rarely used for modelling IR strategies, is given as e.g. $R0[MagColl].p \wedge (!\, R2[BookColl].p)$. In addition, the arithmetic-style representations are: $R0.p * (1 - R2.p)$ for assumption as independent, $R0.p$ for assumption as disjoint, and $R0.p - R2.p$ or $c[0.0]$ for assumption as subsumed (see Table 3.5 for details, also see Seciton 2.5.2.2 and Figure 2.9 for definitions).

3.4.2 Principles of SCX Manipulation

The data structure for implementing SCX could be based on binary tree, which is similar to operator tree for relation algebra (including PRA) or so called query plan. Similarly, the implementation structure of SCX is called SCX operator tree.

In general, there are two kind of nodes in an operator tree: inner-node and leaf-node. An inner-node may not have a parent, but it must have either a child or two children. In contrast, a leaf-node must not have any child, but similarly, it may not have a parent either. Therefore, the root-node of an operator tree could be either an inner-node or a leaf-node.

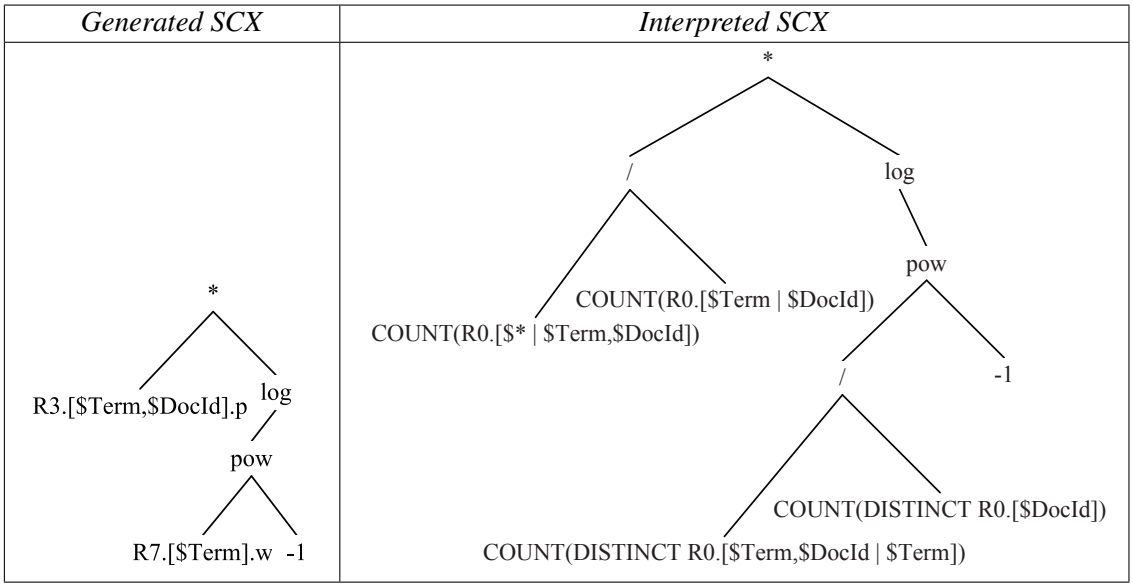


Figure 3.11: SCX operator trees for *tf-idf* model

In a SCX operator tree, an inner-node must be either arithmetic operators, or event operators, or actual functions (i.e. standard functions and conditional selection); whereas a leaf-node must be one of scoring variables, parameters, scores, and symbolic functions (i.e. aggregate symbols). Note that bottom-up evaluation is applied for processing SCX operator tree, but it is also possible to adapt the methods to a top-down evaluating manner.

For instance, Figure 3.11 illustrates the SCXs for *tf-idf* model in Example 3.3.1 (and see Section 3.3.5 for its interpreted SCX). Furthermore, it also demonstrates the basic ideas of transforming SCX for scoring-driven optimization. First, a generated SCX operator tree are built for certain PRA operator simply based on initial settings (for that PRA operator), which details have been discussed in Section 3.4.1; second, a rule-based SCX interpreter analyses the scoring semantics of generated SCX as well as its associated PRA sub-expression, and if possible, then the

interpreter manipulates an intensional semantics based (sub) SCX into an extensional semantics based transformation, which details will be addressed in Section 3.4.4.

Before we discuss how SCX is to be collaborated with PRA, let us have a look at the manipulations of SCX operator tree, which is performed as rotations around a chosen SCX operator. And then, we investigate the rules of SCX transformations based on mathematical or logical equalities.

3.4.2.1 Rotation-Based Manipulations

Rotation is a basic manipulation to transform a binary SCX operator (sub) tree into another form of (sub) tree. The method of rotations are illustrated in Figure 3.12.

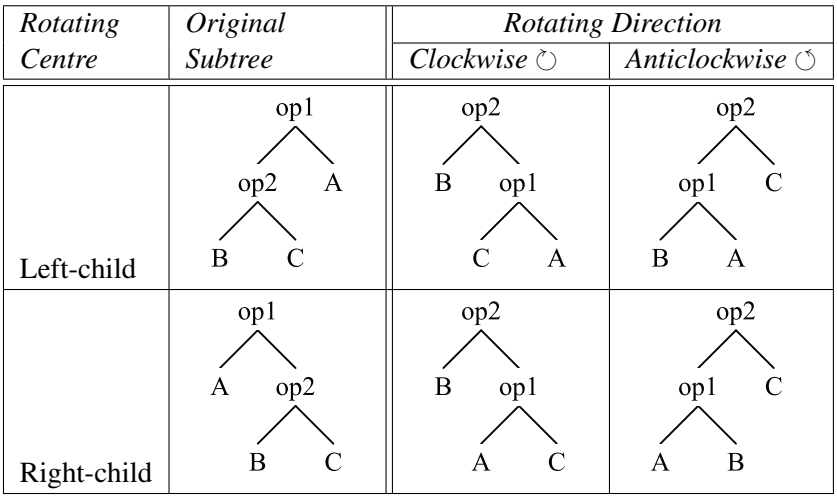


Figure 3.12: Rotating binary SCX operator (sub) tree

To perform rotation on an operator tree, first is to choose a rotated node and its rotating centre, and second is to rotate the node around the rotating centre clockwise or anticlockwise. Let us take examples from Figure 3.12. Let $op1$ to be selected as rotated node to rotate around its child $op2$ (i.e. rotating centre). Initially, $op1$ is the parent of $op2$. In general, a rotation swaps the parent-child relationship of the rotated node and its rotating centre, and it also rearranges other related nodes accordingly.

Firstly, let $op2$ be the left-child of $op1$: to rotate $op1$ clockwise, $op1$ becomes the right-child of $op2$, and the original right-child of $op2$ (i.e. node C) becomes the left-child of $op1$; whereas to rotate $op1$ anticlockwise, then $op1$ becomes the left-child of $op2$, while the original left-child of $op2$ (i.e. node B) becomes the left-child of $op1$.

Similarly, let $op2$ be the right-child of $op1$: to rotate $op1$ clockwise, $op1$ becomes the right-

child of $op2$, and the original right-child of $op2$ (i.e. node C) becomes the right-child of $op1$; whereas to rotate $op1$ anticlockwise, then $op1$ becomes the left-child of $op2$, while the original left-child of $op2$ (i.e. node B) becomes the right-child of $op1$.

An algorithm to perform clockwise rotation is given in Figure 3.13, while an algorithm for anticlockwise rotation is shown in Figure 3.14.

Algorithm: RotateClockwise
Input: SCX operator (sub) tree, rotated node, rotating centre
Output: SCX operator (sub) tree

```
1  begin
2    if RotatingCentre is Node->LeftChild
3      tmpNode = Node->LeftChild;
4      Node->LeftChild = tmpNode->RightChild;
5    else if RotatingCentre is Node->RightChild
6      tmpNode = Node->RightChild;
7      Node->RightChild = tmpNode->RightChild;
8    endif
9    tmpNode->RightChild->Parent = Node;
10   tmpNode->RightChild = Node;
11   if Node is Node->Parent->LeftChild
12     Node->Parent->LeftChild = tmpNode;
13   else if Node is Node->Parent->RightChild
14     Node->Parent->RightChild = tmpNode;
15   endif
16   tmpNode->Parent = Node->Parent;
17   Node->Parent = tmpNode;
18 end
```

Figure 3.13: Rotate clockwise

3.4.2.2 Transformations of SCX

In principle, SCX transformations comply with mathematical equalities or logical equivalences, and they are performed (achieved) by SCX manipulation(s) on operator tree such as rotation and substitution. Though whether a transformation is applicable to interpret a specific PRA expression also depending on certain circumstances, for instance, the features of actual relation(s), and the conditions applied for PRA (sub) expression, and also if input views (i.e. intensional relation or intermediate relations) can be derived from the same extensional relation(s).

Nevertheless, leaving along specific PRA expressions and discussing independently SCX transformations would still make sense, because scoring expressions only focus on aggregations and calculations of tuple *scores*, which are independent from relational operations (of PRA). On the other hand, since the transformations of SCX can be articulated, then developing rule-based interpretations of scoring semantics in SCX for PRA expressions becomes practical. Therefore,

Algorithm: RotateAnticlockwise
Input: SCX operator (sub) tree, rotated node, rotating centre
Output: SCX operator (sub) tree

```
1  begin
2    if RotatingCentre is Node->LeftChild
3      tmpNode = Node->LeftChild;
4      Node->LeftChild = tmpNode->LeftChild;
5    else if RotatingCentre is Node->RightChild
6      tmpNode = Node->RightChild;
7      Node->RightChild = tmpNode->LeftChild;
8    endif
9    tmpNode->LeftChild->Parent = Node;
10   tmpNode->LeftChild = Node;
11   if Node is Node->Parent->LeftChild
12     Node->Parent->LeftChild = tmpNode;
13   else if Node is Node->Parent->RightChild
14     Node->Parent->RightChild = tmpNode;
15   endif
16   tmpNode->Parent = Node->Parent;
17   Node->Parent = tmpNode;
18 end
```

Figure 3.14: Rotate anticlockwise

here we introduce SCX transformations while not yet consider specific PRA semantics, but we discuss the possible circumstances where a transformation could be applicable.

Transformation 3.4.1. *Swapping the order of multiplication and division.* This is based on the associative law of multiplication and division, i.e. $a \cdot (b/c) = (a \cdot b)/c$. The transformation is demonstrated in Figure 3.15, which is performed by an anticlockwise rotation, where operator '$*$' (i.e. multiply) is the parent and rotated node, and division operator '$/$' is the right-child of '$*$' and rotating centre.

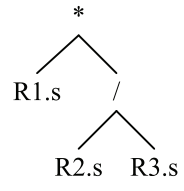
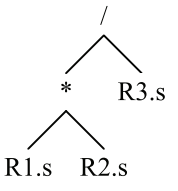
Representation	*Original SCX*	*Transformed SCX*
Expression	$R1.s * (R2.s/R3.s)$	$(R1.s * R2.s)/R3.s$
Operator tree		

Figure 3.15: From Division-Multiplication to Multiplication-Division

Informally speaking, because the above transformation involves at least one Join[7], hence an

[7]It would involve one join operation in PRA but could involve two joins in conventional RA.

essential feature of the transformation is that the order of calculations can be swapped while given a premise that the tuples (i.e. the owner of scores) of different inputs can be always matched up.

Transformation 3.4.2. *Simplifying multiplication which one of the multiplier is constant one.* This is based on equality of $1 \cdot a = a$. The transformation is demonstrated in Figure 3.16, which is by replacing the multiplication sub-tree to one of its children which could be either a non-constant-one variable, or a judiciously selected variable.

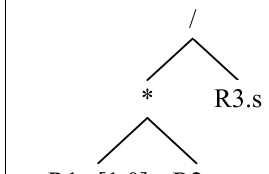
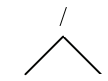
Representation	*Original SCX*	*Transformed SCX*
Expression	$(R1.c[1.0] * R2.s)/R3.s$	$R2.s/R3.s$
Operator tree		

Figure 3.16: Simplifying multiplication with one

Transformation 3.4.3. *Reformulating accumulation of unit fractions to ordinary fraction.* A unit fraction is the reciprocal of positive integer, where the numerator is one and the denominator is a positive integer, i.e. $\frac{1}{n}$, where $n \in \mathbb{N}$ is a positive natural number. The transformation is based on equality of $\sum_1^k \frac{1}{n} = \frac{\sum_1^k 1}{n}$. The transformation is demonstrated in Figure 3.17, which is performed by an anticlockwise rotation, where operator '$+$' (i.e. plus) is the parent and rotated node, and division operator '$/$' is the right-child of '$+$' and rotating centre.

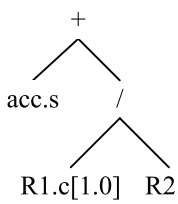
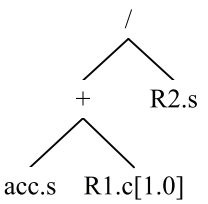
Representation	*Original SCX*	*Transformed SCX*
Expression	$acc.s + (R1.c[1.0]/R2.s)$	$(acc.s + R1.c[1.0])/R2.s$
Operator tree		

Figure 3.17: From Division-Accumulation to Accumulation-Division

Transformation 3.4.4. *Swapping the order of conjunction and disjunction.* This is based on the distributive law of logical conjunction and disjunction, i.e. $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$. The

transformation is demonstrated in Figure 3.18, which is performed by a clockwise rotation or an anticlockwise rotation, where operator '$\vee$' (i.e. logical OR) is the parent and rotated node, and logical AND operator '$\wedge$' is the right-child of '$\vee$' and rotating centre.

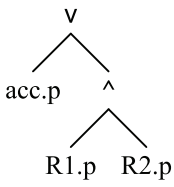
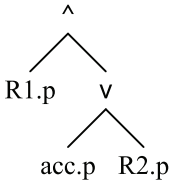
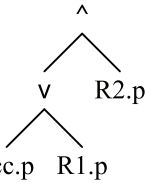
Representation	*Original SCX*	*Transformed SCX (CW)*	*Transformed SCX (ACW)*
Expression	$\vee\ (R1.p \wedge R2.p)$	$R1.p \wedge (\vee\ R2.p)$	$(\vee\ R1.p) \wedge R2.p$
Operator tree			

Figure 3.18: From Conjunction-Disjunction to Disjunction-Conjunction

Transformation 3.4.4 describes a special case of logic-style SCX: firstly, for a logical expression does not have a left-hand-side operand, i.e. a passive operand, a SCX operator tree builder insert a probability accumulator (i.e. for probability aggregation) to the left-child node position of the logical operator, such as they are shown in Figure 3.18; secondly, the transformation can be performed in two rotating directions, i.e. clockwise or anticlockwise, while each direction should be applied is depending on the relational semantics of a PRA expression.

3.4.3 Automatic Analysis for SCX

Here we discuss how to automatically analyse generated SCXs by utilising semantic graph. A semantic graph tool for text and image analysis was introduced in [Hébert, 2006], which is based on conceptual graphs (or semantic networks) proposed earlier in [Sowa, 1984]. However, our purposes are much simpler and more focused, which only aims to automatically analyse generated SCX for interpretation. Therefore, we adapted the technique addressed in [Hébert, 2006] and developed a simplified variant for scoring semantics analysis.

The elements that make up a semantic structure include entity, relationship and direction. An entity is a vertex of a graph, while a relationship and a direction must appear together in a graph to form a directed link, which is also an edge of a graph. Here an example shows a semantic structure of division operation by articulating the relationships between operator and operands, where numerator is a passive operand of division, whereas denominator is an active operand:

$$[\text{numerator}] \leftarrow (\text{PASOPD}) \leftarrow [\text{DIV}] \rightarrow (\text{ACTOPD}) \rightarrow [\text{denominator}]$$

Table 3.6 summarises the constituent elements and the symbols for representing a semantic structure in either textual format or graphic format.

Element	*Type*	*Symbols*	
		Textual Format	*Graphic Format*
entity	vertex	square brackets: [...]	rectangle
relationship	edge	parentheses: (...)	ellipse
direction	edge	arrow: $\rightarrow$ or $\leftarrow$	arrow

Table 3.6: Elements of a semantic graph for SCX

Currently nine symbols corresponding to nine possible relationships between every two entities are applied, which are demonstrated and explained in Table 3.7. Among them, most relationships (8 out of 9) are applied to form a template of the semantic graph for analysing SCX, whereas '(MEANS)' (i.e. means) would be deployed dependently to actual entities.

In addition, Table 3.8 depicts the meanings of semantic structures formed by entities and directed links, in which the roles of entities in a structure are clarified.

Now we are ready to discuss how to analyse scoring expressions using semantic graphs. As aforementioned in Section 3.4.2, a SCX would be implemented in a binary operator tree, and only inner nodes are needed to be analysed. During analysis, an analyser investigate the semantics of every inner nodes in the SCX operator tree from bottom-up, while manipulations would be triggered when pre-defined conditions are satisfied.

In general, the semantic graphs for SCX operators are created by using a template shown in Figure 3.19; for different type of entities, Table 3.9 describes their meaning and usage.

To explain how does semantic analysis work in short, an analyser program enters a graph from an entry point, which could be either from the generalised SCX operator or from the score (i.e. from different point of views), and traverses through the graph following directed links. With regard to how to choose and entry point, the analyser enters from the operator viewpoint to analyse an operator, whereas it enters from the score viewpoint to analyse the inputted variable(s) of an operator.

To walk through a graph from the operator entry, analyser examines: 1) what is the specialised operation of the operator; 2) what is its active operand and does it have an passive operand as well? 3) what is the expected result (goal) and what is actually obtained (effect)? 4) is there any manipulation would be triggered if certain effect occurs?

Similarly, while analysing from the score entry, analyser checks: 1) how the resulting scores

Symbol	*Meaning*	*Definition*
(SPEC)	specialisation	the specialisation of an entity
(PASOPD)	passive operand	the passive operand of an operator
(ACTOPD)	active operand	the active operand of an operator
(GOAL)	goal	expected result or sought effect
(EFFECT)	effect	actual result or consequence
(CLASS)	class	an element of a class
(PROP)	property	attribute or characteristic
(REC)	receiver	entity that receives a transmission
(MEANS)	means	means used, instrumental

Table 3.7: Symbols of relationships

[a specialised element]	←	(SPEC)	←	[a generalised element]
[a passive operand of the operator]	←	(PASOPD)	←	[operator]
[an active operand of the operator]	←	(ACTOPD)	←	[operator]
[element of desired result]	←	(GOAL)	←	[operator]
[element of actual result or effect]	←	(EFFECT)	←	[operator]
[element of class]	←	(CLASS)	←	[element being classified]
[specific characteristic]	←	(PROP)	←	[element to which the property is given]
[element that receives the result]	←	(REC)	←	[result being transmitted]
[element being used]	←	(MEANS)	←	[element to which the means is applied]

Table 3.8: Meanings of entities and directed links in a semantic structure

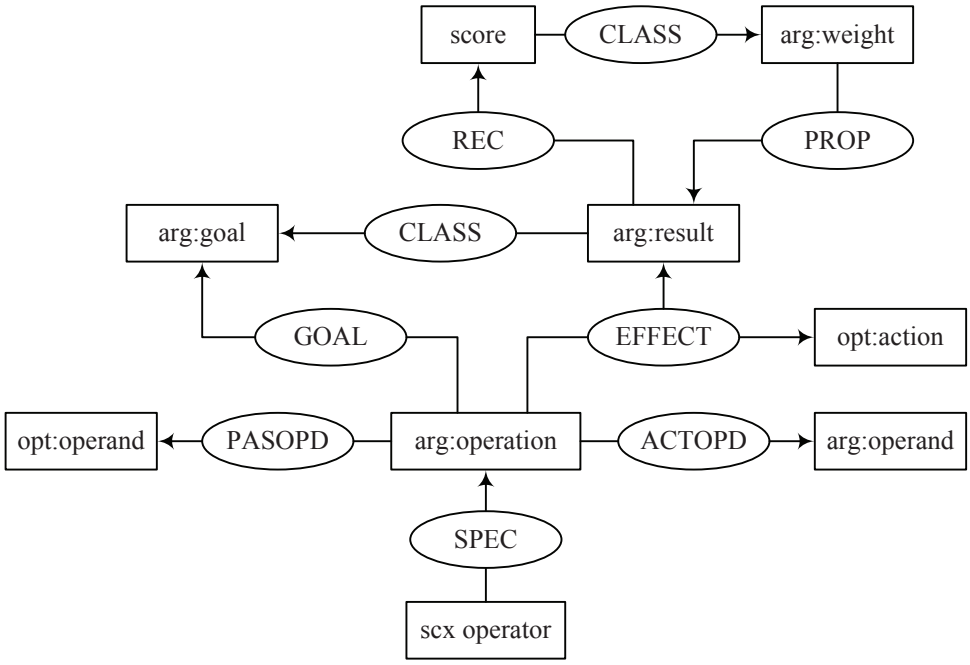


Figure 3.19: A template of semantic graph for analysis

Form of Entity	*Description*	*Remark*
<name>	static entity	analysing entry
arg:<name>	mandatory argument	pre-defined, setting based on operations
opt:<name>	optional argument	pre-defined, setting based on operations and rules

Table 3.9: Forms of entity in a SCX semantic graph

of an operator can be classified: i.e. are they normal weightings, or constants, or probabilities; 2) is there other property that further describes the characteristics of scores, e.g. for scores to be recognised as fractions or unit fractions.

The template could be implemented in any potential data structure. A portable (and flexible) option is to store this template in an XML format as follow, while in-memory data structure of graph can be built during analysis.

```
<scxsg>
  <scxopr>
    <spec>arg:operation</spec>
    <pasopd>opt:operand</pasopd>
    <actopd>arg:operand</actopd>
    <goal>arg:goal</goal>
    <effect>arg:result</effect>
  </scxopr>
  <score>
    <class>arg:weight</class>
    <prop>arg:result</prop>
  </score>
  <arg>
    <operation></operation>
    <operand></operand>
    <goal></goal>
    <result></result>
    <weight></weight>
  </arg>
  <opt>
    <operand></operand>
    <action></action>
  </opt>
</scxsg>
```

Pre-defined rules will be discussed in the next section, while more examples for analysing SCX with semantic graphs are given in Appendix A.

3.4.4 Commencing Scoring-Driven Optimization

So far we have addressed the preliminary techniques that are necessary for interpreting scoring semantics of PRA expressions, so that we are now ready to discuss the procedures of scoring-driven optimization. In this sub-section, we discuss the related algorithms for a rule-based opti-

mizer; and then, how to interpret generated SCXs so that (logical) query plan can be mapped to (physical) execution plan utilising sophisticated index; moreover, how to adjust scoring functions based on intensional semantics for physical operators; and final, how to verify scoring equivalence for PRA expressions.

3.4.4.1 Algorithm and Rules

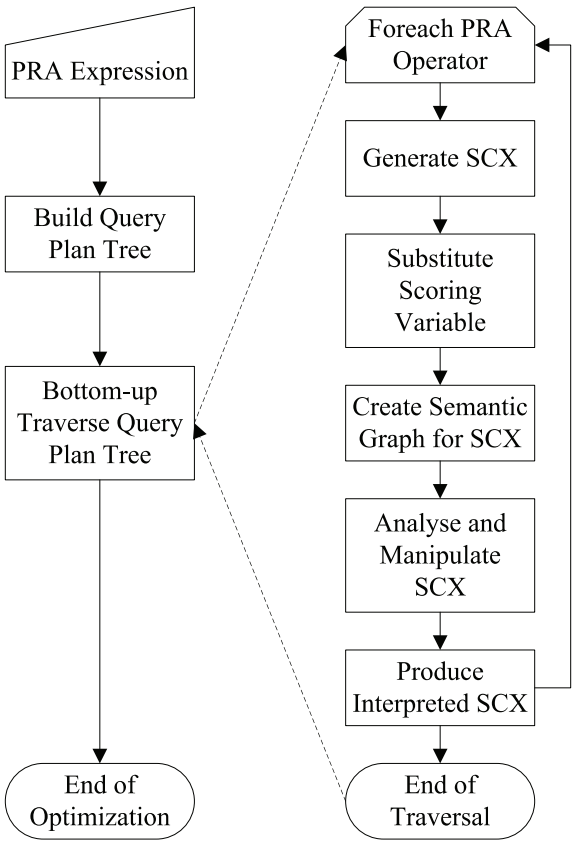


Figure 3.20: Flowchart for the procedure of rule-based optimizer

In general, a scoring-driven optimizer takes PRA expressions as input and output query plans articulated by scoring expressions, a flowchart in Figure 3.20 illustrates the procedures of the optimization. In short, a binary plan tree (or PRA operator tree) is built at first for a given PRA expression, where unique identifiers would be assigned to every nodes in the plan tree; and then, an algorithm performs a bottom-up traversal to visit every nodes throughout the plan tree in order to articulate the scoring functions of PRA operators in SCXs; at the end of the traversal, all nodes in the query plan would have been marked by an interpreted SCX and the optimization procedure finishes. The output of the optimization, which is called *articulated query plan*, would be passed to another process, where logical plans are mapped to execution plans; though how

to exploit interpreted SCXs in the mappings of logical-physical operators depends on the actual implementation of a query engine, but we will address later how scoring information could be used.

Algorithm: ScxArticulate
Input: PRA operator tree
Output: Articulated PRA operator tree

```
1  /* Visit nodes of PRA operator tree in post-order traversal */
2  function PostTraverse(PNode)
3  begin
4    if PNode->Type is not TABLE
5      PostTraverse(PNode->LeftChild);
6      PostTraverse(Node->RightChild);
7    endif
8    PNode = Articulate(PNode);
9  end

11 /* Articulate scoring semantics in SCX for a PRA operator node */
12 function Articulate(PNode)
13 begin
14   create initial SCX for PNode and set PNode-GenScx;
15   set PNode->InterScx = PNode->GenScx;
16   if PNode->Type is not TABLE
17     if PNode->Valency equals one
18       replace the scoring variable of PNode->InterScx by PNode->
             LeftChild->InterScx;
19     else if PNode->Valency equals two
20       replace corresponding scoring variables of PNode->InterScx by
             PNode->LeftChild->InterScx and PNode->RightChild->InterScx;
21     endif
22     create semantic graph(s) for PNode->InterScx;
23     analyse and manipulate PNode->InterScx;
24     if the number of owners of scoring variables of PNode->InterScx
           is greater than PNode->Valcency
25       PNode->InterScx = PNode->GenScx;
26     endif
27   endif
28   return PNode;
29 end
```

Figure 3.21: Articulating scoring function for PRA operators with SCX

The procedure of articulating a PRA operator contains several steps: first, a pre-defined SCX would be generated following the rules in Section 3.4.1, note that initially a generated SCX would include no more than two non-accumulator scoring variables[8]; second, if the PRA operator is not an extensional relation (i.e. table), then the procedure substitutes every non-accumulator scoring variable of the generated SCX by an interpreted SCX of that variable, which could be found from a corresponding child node of the PRA operator; third, one or several semantic

[8]Because the maximum valency of a PRA operator is binary.

graph(s) (see Section 3.4.3) would be created for the SCX; fourth, analysis is carried out based on semantic graphs, while various manipulations (see Section 3.4.2) for SCX could be triggered; final, if the owners of all non-accumulator variables in the SCX are extensional relations only, then the procedure keeps the SCX as an interpreted SCX for the PRA operator, otherwise it rewinds the original generated SCX (i.e. before scoring variable substitution) for interpretation. To summarise, an algorithm for the articulating procedure, called *ScxArticulate*, is shown in Figure 3.21

For analysing SCX, an analyser applies semantic graphs (see Section 3.4.3) and follows below rules:

1. Substitute SCX sub-expressions (e.g. see Section 3.3.5) of aggregate summation (i.e. with accumulator) by symbolic functions (i.e. COUNT etc) when it is applicable.

2. Manipulate the new SCX operator tree (after variable-subtree substitution) based on the transformations in Section 3.4.2. While rotation is to be performed then takes the root-node of the new SCX tree as rotated node, and the root-node of substituting subtree as rotating centre.

Moreover, all manipulations of SCX must comply to a *conflict free* principle, which is defined as follow:

Definition 3.4.1. Conflict Free SCX Manipulation. Any manipulations to be applied for SCXs must not conflict to the relational semantics of PRA expressions; that is, if a transformation of SCX indicates different cardinality of result to the articulating PRA expression, then the transformation is in conflict to the relational semantics of the PRA expression, and a SCX manipulation applying the transformation must not be performed. For a manipulation applying a transformation to conflict free, the SCXs in a transformation must satisfy the following two criteria:

1. the number of scores indicated by the SCXs in a transformation must indicate the same cardinality of resulting relation;

2. the mappings of scores to tuples indicated by the two SCXs in a transformation must identical.

Next, we demonstrate the analysis and manipulations with examples.

3.4.4.2 Assisting Index Selection

In principle, index selection technique can be employed in IR+DB systems, where different types of index are implemented and deployed for supporting efficient processing for complex queries.

Similar techniques have been widely used in the applications of databases such as decision support systems and data warehouse systems (e.g. see [Golfarelli et al., 2002]). Generally, multiple types of index would be implemented in databases, such as Tuple-ID-based index (or TID-List index) and Bitmap index, and then a broker program decides what type of index would benefit the most for specific queries. so that the broker needs sophisticated algorithms for it to perform selection. For example, a broker program may depend on pre-defined cost models, so it can estimate the cost of a given physical operation while using a candidate index; by investigating all possible options (indexes), then the broker can nominate an index for the operation.

Though we do not discuss index selection for IR+DB systems in this thesis, which could be one of the most interesting topics for future work, but we explain what information provided by SCXs that may assist index selection in principle.

Previously, we presented in Section 2.5.2 the basic statistics for modelling popular IR strategies, because these statistics, including within-collection or within-document *tf* and document-based frequencies, can be obtained from counting-based aggregations in IR+DB systems. On the other hand, PRA does not apply counting as its basic operation; instead, it uses an equivalent accumulative summation, which acts as the same as counting when inputs are all ones. Therefore, when modelling IR strategies in PRA, it is unable to distinguish basic frequencies (and statistics from computing frequencies) from probabilities, which poses difficulties for a query execution engine to utilise indexes. By employing SCX to articulate counting in PRA, this problem can be solved: Table 3.10 demonstrates how aggregations can be represented in symbolic functions of counting so that they can be associated to IR statistics (see Section 2.5.2.1 for notions).

As they are shown, each symbolic function in Table 3.10 maps to an IR statistic concept, which means IR-style inverted index can be employed in IR+DB systems. Though one can "simulate" IR-style inverted index by vanilla B+-tree index in conventional databases, but because inverted index has never been viewed as the first citizen in DB, therefore it is impossible to employed popular IR indexing techniques in databases without significantly re-engineering DB's query engine. On the other hand, in order to allow an IR+DB query engine to utilise IR-style inverted index, such as an relation inverted index (RIX) which will be introduced in the next

Symbolic Function for Aggregation	*Notion*	*Description*
COUNT(R)	$\|\mathcal{R}^m\|$	the (total) number of tuples
COUNT(DISTINCT R.[$Term])	$\|\mathcal{R}(t)^s\|$	the number of distinct terms
COUNT(DISTINCT R.[$DocId])	$\|\mathcal{R}(d)^s\|$	the number of distinct documents
COUNT(R.[$* | $Term, $DocId])	$\|\mathcal{R}(t,d)^m\|$	within-document term count
COUNT(R.[$* | $Term)	$\|\mathcal{R}(t)^m\|$	within-collection term count
COUNT(R.[$Term | $DocId])	$\|\mathcal{R}(d)^m\|$	document length
COUNT(DISTINCT R.[$Term, $DocId | $Term])	$\|\mathcal{R}(t,d)^s\|$	document frequency

Table 3.10: Symbolic functions in SCX and corresponding IR statistics

chapter (see Chapter 5), a method is needed to set up communications between logical PRA and execution engine, while SCX provides the methods to allow query engine to select proper indexes to do the job.

We use the PRA modelling examples from Section 2.5.2.3 to explain further. Here let us first clarify some conventions: when articulating scoring functions with SCXs, a SCX is written in a textual form of <head> = <body>, where the head is a scoring variable that corresponds to the *score viewpoint* (see Section 3.4.3) in a semantic graph, and the body is a scoring expression representing a scoring function. To initialise a generated SCX for a PRA node, the head is always given by an anonymous variable "this.s", then at the end of an interpretation, the head will be replaced by an articulated variable whose ownership of scores and type of scores are specified. During an interpretation process, it is the scoring expression of body which will be built into a SCX operator tree for analysis and manipulation. These conventions will be complied for all of the later discussions.

Within-collection term frequency $P_C(t)$**.** Firstly, let us have look at an example of articulating PRA expression for within-collection term frequency (i.e. $P_C(t)$, also see Definition 2.5.1).

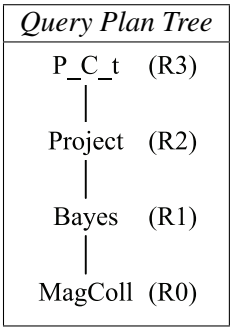
The process for $P_C(t)$ is illustrated in Figure 3.22, in which Figure 3.22a recall the PRA expression, while Figure 3.22b is a query plan tree built for evaluation, and Table 3.22c demonstrates the interpretation procedure and shows the (initial) generated SCXs and the (final) interpreted SCXs. Let the query plan to be referred as the P_C_t plan, at the bottom of the plan is an unweighted extensional relation called *MagColl* (see "original table" in Table 2.3) and it is the input of the query, while at the top of the plan is the output which is a probabilistic view called P_C_t.

To perform bottom-up traversal to articulate the P_C_t plan, the leaf-node at the bottom, i.e.

R0, is to be visited at first. Because R0 is an unweighted table, thus an initial SCX is generated as $this.s = c[1.0]$; to interpret, the program specifies the ownership of scores and the name of the owner of the scoring variable in the body, which becomes $R0[MagColl].c[1.0]$; while the scoring variable in the head is replaced by $R0.c$.

```
P_C_t = PROJECT DISJOINT [$Term] (BAYES DISJOINT [] (MagColl));
```

(a) PRA expression for $P_C(t)$



(b) PRA operator tree

Plan Tree Node	*Generated SCX*	*Interpreted SCX*
R3 (P_C_t)	this.s = R2.p	R3.p = COUNT(R0.[$* | $Term]) / COUNT(R0)
R2 (PROJECT)	this.s = acc.[$Term].w + R1.p	R2.p = COUNT(R0.[$* | $Term]) / COUNT(R0)
R1 (BAYES)	this.s = R0.c[1.0] / (acc.w + R0.c[1.0])	R1.p = R0.c[1.0] / COUNT(R0)
R0 (MagColl)	this.s = c[1.0]	R0.c = R0[MagColl].c[1.0]

(c) Articulating scoring functions with SCX (read bottom-up)

Figure 3.22: Scoring expression for Project-Bayes PRA expression for $P_C(t)$

Next, the traversal algorithm moves one level up to R1 from the leaf-node, where it finds out it is a Bayes operator, which means a probability estimation is performed here. To articulate Bayes, the process checks the probabilistic assumption, which is disjoint, and the evidence attributes, which is empty, and then a SCX is generated accordingly (see Section 3.4.1 and Table 3.3); according to Theorem 3.3.1, the denominator of the division can be substituted by symbolic function $COUNT(R0)$; since there is no further SCX manipulation can be performed, so the process finalises the interpreted SCX of R1 by specifying its head variable as $R1.p$, which indicates the scores obtained from Bayes are probabilities.

And then, the process move up again to R2, which is a projection on attribute $\$Term$ while given a probabilistic assumption as disjoint. The generated SCX is created as $this.s =$

$acc.[\$Term].w + R1.p$, in which an accumulator indicates that an aggregation is performed by accumulative summation while aggregated scores are grouped by attribute $\$Term$. After the input variable $R1.p$ is replaced by its interpreted SCX, i.e. $R0.c[1.0]/COUNT(R0)$, we obtain a SCX as $acc.[\$Term].w + (R0.c[1.0]/COUNT(R0))$, which would be found by a semantic analyser that its pattern matches the original SCX (see Transformation 3.4.3), so an anticlockwise rotation is perform to give a transformed SCX as $(acc.[\$Term].w + R0.c[1.0])/COUNT(R0)$; and then furthermore, the accumulative summation sub-tree is eligible to be replaced by a symbolic function of counting according to Theorem 3.3.3, that is, $COUNT(R0.[\$* \mid \$Term])/COUNT(R0)$; the final step set the head variable to $R2.p$, because the projection is after Bayes and it is given a probabilistic assumption, therefore it is deemed to yield probabilities.

In the end, the traversal reaches the root-node of the P_C_t plan, which is R3. For the node, the articulating process is simple, which only need to replace the body by the interpreted SCX of $R2.p$, and then the articulation is finished.

Because the articulating process involves a SCX transformation, we must guarantee it is a *conflict free* manipulation (see Definition 3.4.1) for the P_C_t plan.

Theorem 3.4.1. *SCX Transformation 3.4.3 is a conflict free manipulation for Project-Bayes queries of PRA while disjointness is given as probabilistic assumptions for Project and Bayes.*

Let R_X be a relation as the input of a Project-Bayes query, and let $|R|$ be its cardinality. Let X_e be a list of evidence attributes of Bayes, where $X_e \subseteq X$; and let X_p be a list of attributes which would be projected, and $X_p \subseteq X$. For a Project-Bayes query which is performed for probability estimation, it is certain that aggregations would be performed.

Proof. Because the cardinality of the result of a Project-Bayes query is decided by the projection, given X_p as projected attributes, where $X_p \geq 1$, we obtain the cardinality of the result is $|R(X_p)^s|$ (see Section 2.5.2.1 and Table 2.4 for notion).

In general, the original SCX for a Project-Bayes query can be written as $acc.[X_p].p + (R1.c[1.0]/R2.[X_e].w)$, which represents the original scoring semantics of Project-Bayes query: first, Bayes aggregates the scores for denominator(s) based on the evidence attributes X_e, so there would be $|R(X_e)^s|$ number of distinct denominators, while the cardinality of the intermediate result yielded by Bayes is the same to the cardinality of R, i.e. $|R|$. Second, Project aggregates the result of Bayes and it groups the result by attribute(s) X_p, in other words, $|R(X_p)^s|$ scores are mapped to $|R(X_p)^s|$ number of distinct tuples.

Moreover, for a probability estimation to be eligible, the scores, denoted as s, yielded by a Project-Bayes query must satisfy $s \in [0,1]$; as a result, we can obtain that $X_e \subseteq X_p$ must be satisfied, and thus $|R(X_e)^s| \leq |R(X_b)^p|$.

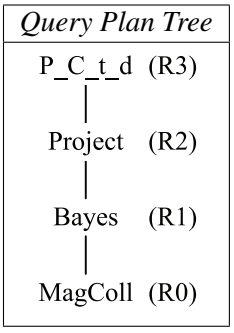
On the other hand, the transformed SCX can be formulized as $(acc.[X_p].p + R1.c[1.0])/R2.[X_e].w$, which indicates that aggregations would be performed before division. Because of $X_e \subseteq X_p$ and $|R(X_e)^s| \leq |R(X_p)^s|$, we can conclude the scores are grouped by attribute(s) X_p and the mappings of scores to tuples would be $|R(X_p)^s|$ number of scores for the same number of distinct tuples, which is the same to the situation give by original SCX.

As a result, manipulations applying Transformation 3.4.3 are conflict free for articulating Project-Bayes queries, thus Theorem 3.4.1 is sound. □

Within-document term frequency $P_C(t|d)$**.** Similarly, a SCX articulation for the PRA expression modelling within-document term frequency (i.e. $P_C(t|d)$, also see Definition 2.5.2) is given in Figure 3.23.

```
P_C_t_d = PROJECT DISJOINT [$Term, $DocId] (BAYES DISJOINT [$DocId](MagColl));
```

(a) PRA expression for $P_C(t|d)$



(b) PRA operator tree

Plan Tree Node	*Generated SCX*	*Interpreted SCX*
R3 (P_C_t_d)	this.s = R2.p	R3.p = COUNT(R0.[$* | $Term, $DocId]) / COUNT(R0.[$* | $DocId])
R2 (PROJECT)	this.s = acc.[$Term, $DocId].w + R1.p	R2.p = COUNT(R0.[$* | $Term, $DocId]) / COUNT(R0.[$* | $DocId])
R1 (BAYES)	this.s = R0.c[1.0] / (acc.[$DocId].w + R0.c[1.0])	R1.p = R0.c[1.0] / COUNT(R0.[$* | $DocId])
R0 (MagColl)	this.s = c[1.0]	R0.c = R0[MagColl].c[1.0]

(c) Articulating scoring functions with SCX (read bottom-up)

Figure 3.23: Scoring expressions for Project-Bayes PRA expression for $P_C(t|d)$

In principle, the analysis procedure for articulating the PRA expression for $P_C(t|d)$ performs

almost the same to the analysis for $P_C(t)$, whereas there are there differences: the first is an evidence attribute $\$DocId$ is given to Bayes for probability estimation; the second is that there are two projected attributes instead of one, which are $\$Term$ and $\$DocId$; and the third is that replacements of accumulative summations by symbolic functions, including the one for aggregating denominator in Bayes and the one for aggregation in projection, are based on Theorem 3.3.3. In addition, Theorem 3.4.1 is also applicable to support Transformation 3.4.3 to be employed as a conflict free manipulation for interpreting the scoring semantics of the PRA expression for modelling $P_C(t|d)$.

Document Frequency $\mathbf{df}(t)$**.** Both previous modelling of *tf*s in PRA involves the same Project-Bayes query, now let us discuss another pattern of PRA expression that involves Project-Join-Bayes query. Here an example of modelling document frequency (i.e. $df(t)$, also see Definition 2.5.3) in PRA and articulating in SCX is given in Figure 3.24.

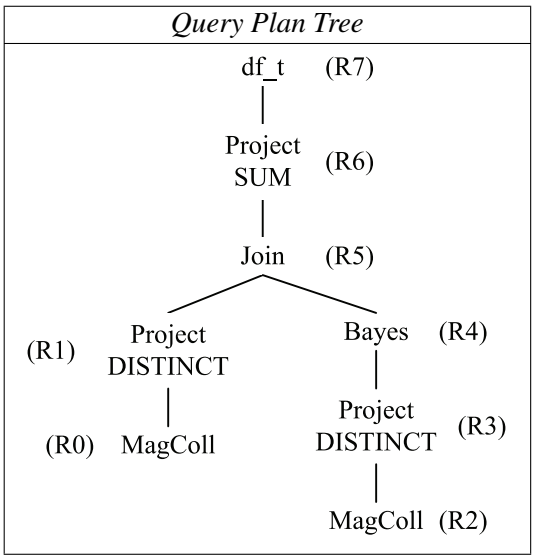
First of all, let us recap some concepts from the discussions in Section 2.5.2.1, that is, both *tf*s are to be estimated based on *tuple space*, which can be usually taken from extensional relations; whereas *df* is to be estimated based on a different event space from the *tf*s' called *document space* (or *subject space*), in which events are distinct documents. Therefore, different from *tf*s which can be estimated directly from extensional relations, to compute *df*, we have to generate document space before estimation can be taken place. As a result, the modelling of *df* in PRA can be divided into several sub-queries (see Figure 3.24a) where each sub-query yields an intermediate result different concepts including a document space and other related contexts. To combine the sub-queries into a single query plan, which is referred as the df_D_t plan and it is shown in Figure 3.24b, then similarly, the df_D_t plan can be articulated by SCX for interpretation of scoring semantics.

Let us walk through the df_t plan with bottom-up traversal. Since a Join operation taking two sub-queries as inputs forms two branches in a plan tree, a post-order traversal would visit the left branch at first, and then the right branch at second, and then the Join operator at last. In addition, the order of the visiting for articulation can also be found out from the aliases with unique identifiers of the nodes (i.e. R0 - R7).

We skip node R0, because it is the same to the previous discussions, and get started from node R1. It is found that a distinct projection is performed and no projected attributes are specified, so according to Section 3.4.1 (also see Table 3.3), an initial SCX is generated as

```
dColl = PROJECT DISTINCT(MagColl);
docSpace = BAYES DISJOINT [](PROJECT DISTINCT [$DocId](MagColl));
termDoc = PROJECT [dColl.$Term, dColl.$DocId](
   JOIN [dColl.$DocId = docSpace.$DocId](dColl, docSpace));
df_t = PROJECT SUM [$Term](termDoc);
```

(a) PRA expression for $df(t)$



(b) PRA operator tree

Plan Tree Node	*Generated SCX*	*Interpreted SCX*
R7 (df_t)	this.s = R6.w	R7.w = COUNT(DISTINCT R0.[$Term, $DocId | $Term]) / COUNT(DISTINCT R0.[$DocId])
R6 (PROJECT)	this.s = acc.[$Term].w + R5.p	R6.w = COUNT(DISTINCT R0.[$Term, $DocId | $Term]) / COUNT(DISTINCT R0.[$DocId])
R5 (JOIN)	this.s = R1.c $*$ R4.p	R5.p = R0.[$Term, $DocId].c[1.0] / COUNT(DISTINCT R0.[$DocId])
R4 (BAYES)	this.s = R3.c[1.0] / (acc.w + R3.c[1.0])	R4.p = R0.[$DocId].c[1.0] / COUNT(DISTINCT R0.[$DocId])
R3 (PROJECT)	this.s = acc.[$DocId].c[1.0] + R2.c[1.0]	R3.c = R0.[$DocId].c[1.0]
R2 (MagColl)	this.s = c[1.0]	R2.c = R0[MagColl].c[1.0]
R1 (PROJECT)	this.s = acc.[$*].c[1.0] + R0.c[1.0]	R1.c = R0.[$Term, $DocId].c[1.0]
R0 (MagColl)	this.s = c[1.0]	R0.c = R0[MagColl].c[1.0]

(c) Articulating scoring functions with SCX (read bottom-up)

Figure 3.24: Scoring expressions for Project-Join-Bayes PRA expressions for $df(t)$

$this.s = acc.[\$*].c[1.0] + R0.c[1.0]$ to indicate duplicate removal (but not aggregation); to interpret this case, the process would specify the projected attributes by taking all attribute names from the schema of R0, and then to re-formulize the body of the SCX by $R0.[\$Term, \$DocId].c[1.0]$, which means the projection yields unweighted intermediate result whose tuples originally come from R0 and are grouped by the specified attributes; in addition, the head of the SCX is replaced by $R1.c$ to announce the characteristic of scores.

And then, traversal algorithm moves articulating process to R2, which is again a table; but moreover, it is the same to R0, therefore, the interpreting process remarks the node with $R0[MagColl].c[1.0]$, which is the same interpreted SCX as node R0. Level up, the process visits node R3, which performs a distinct projection on attribute $\$DocId$. Because the articulation here is similar to node R1, hence we skip this node and move on to node R4.

Specially, node R4 contains a Bayes operation which is performed to produce a document space for computing $df(t)$. Initially, a generated SCX is created as $R3.c[1.0]/(acc.w + R3.c[1.0])$; and then the interpreter substitutes the scoring variable of $R3.c[1.0]$ by its interpreted SCX $R0.[\$DocId].c[1.0]$, so that the SCX becomes $R0.[\$DocId].c[1.0]/(acc.w + R0.[\$DocId].c[1.0])$; furthermore, the accumulative summation for denominator of the division can be replaced by a symbolic function; that is, according to Theorem 3.3.4, the body of the interpreted SCX is transformed into $R0.[\$DocId].c[1.0]/COUNT(DISTINCT\ R0.[\$DocId])$, while the head of the SCX is specified as $R4.p$.

After interpreting the branches of Join, the articulating process reaches node R5. Here the result of sub-query *dColl* (i.e. the result of R1) would be combined with the result of *docSpace* (i.e. the result of R4), which means on the one hand, the tuples of two intermediate results would be concatenated, and on the other hand, while given a probabilistic assumption as independent, the tuples scores of either intermediate results are to be combined by multiplication. Therefore, a generated SCX for R5 is given as $R1.c * R4.p$. To replace the variables by their interpreted SCXs, the SCX is transformed into $R0.[\$Term, \$DocId].c[1.0] * (R0.[\$DocId].c[1.0]/COUNT(DISTINCT\ R0.[\$DocId]))$, which matches the original form of Transformation 3.15. Let us leave the discussion for conflict free manipulation later, but apply the transformation by now. So by deploying an anti-clockwise rotation as suggested by Transformation 3.15, we get $(R0.[\$Term, \$DocId].c[1.0] * R0.[\$DocId].c[1.0])/COUNT(DISTINCT\ R0.[\$DocId])$. And then, the obtained SCX matches

another transformation, i.e. Transformation 3.4.2, where a judiciously chosen variable may replace the multiplication sub-tree. Again, let us choose $R0.[\$Term, \$DocId].c[1.0]$ and leave the explanation later. In the end, the interpreted SCX for R5 is given by $R5.p = R0.[\$Term, \$DocId].c[1.0]/COUNT(DISTINCT\ R0.[\$DocId])$.

Next, articulating process is moved up to node R6, which is an projection given a projected attribute $\$Term$ and aggregated assumption as sum. Hence, according to Table 3.4, we obtain a generated SCX as $acc.[\$Term].w + R5.p$, which means scores would be grouped by distinct values of attribute $\$Term$; and then, to replace $R5.p$ by its interpreted SCX, we get $acc.[\$Term].w + (R0.[\$Term, \$DocId].c[1.0]/COUNT(DISTINCT\ R0.[\$DocId]))$, which is accumulation of unit fractions and it matches the original form of Transformation 3.4.3. Again, let us leave the verification of conflict free later and accept a manipulation applying the transformation. So the SCX becomes $(acc.[\$Term].w + R0.[\$Term, \$DocId].c[1.0])/COUNT(DISTINCT\ R0.[\$DocId])$, while according to Theorem 3.3.5, the accumulation of constant-ones can be replace by $COUNT(DISTINCT\ R0.[\$Term, \$DocId \mid \$Term])$, so that the body of the interpreted SCX is finished, and the head is given as $R6.w$.

At last, the traversal process reaches the root-node of the df_D_t plan and finalise the articulation.

For now, let us verify if the transformations used above are conflict free manipulations. Similar settings are applied, where let R_X for relation, $|R|$ for cardinality, X_e for evidence attributes of Bayes, X_p for projected attributes of Project, Θ for Join predicate where $\Theta = \{=, \neq, <, \leq, > , \geq, \approx\}$, and $\mu : \overline{X}\Theta\overline{Y}$ for Join condition. The transformations articulate two kinds of PRA expressions: 1) Join-Bayes queries; and 2) Project-θ-Bayes queries, where θ could be any PRA operator except for aggregated projection and Bayes. Firstly, Theorem 3.4.2 is given with respect to the swapping of multiplication-division transformation.

Theorem 3.4.2. *SCX Transformation 3.4.1 is a conflict free manipulation for Join-Bayes queries of PRA while independence is given as a probabilistic assumption for Join.*

Proof. The cardinality of the result of Join-Bayes query is decided by the Join condition, while a Cartesian product instead of a Join is applied, the cardinality of result is given by $|R_X| \cdot |R_Y|$, where R_X is the first input and R_Y is the second input. Moreover, let R_Y also be the event space for Bayes operator, then the original SCX of Transformation 3.4.1 can be written in a specialised form as $R1.s * (R2.[Y_e].c[1.0]/R2.[Y_e].w)$, while the cardinality of the result of a Join-Bayes

query is $\left|R(X)^{\alpha}_{\mu:\overline{X}\Theta\overline{Y}}\right| \cdot \left|R(Y)^{\alpha'}\right|$, where the adornment of relation for both inputs are not restricted, which means they can be either based on set or multiset.

After Transformation 3.4.1 is applied, a specialised form of the original SCX becomes $(R1.s * R2.[Y_e].c[1.0])/R2.[Y_e].w$, which means a Join would be performed before a Bayes. In this case, the cardinality of the result of Join is still $\left|R(X)^{\alpha}_{\mu:X\Theta Y}\right| \cdot \left|R(Y)^{\alpha'}\right|$, while the cardinality of the result of Bayes depends on the number of scores (tuples) of numerator, which means the final cardinality of the result is the same to the original SCX of the transformation. Moreover, the mappings of scores are guaranteed by Join condition. Therefore, manipulations applying Transformation 3.4.1 for Join-Bayes queries of PRA are conflict free, and thus Theorem 3.4.2 is sound. □

In addition, there is another Theorem which is suitable and necessary for Join-Bayes queries. Theorem 3.4.3 specifies the applicability of Transformation 3.4.2, which is actually a further manipulation after Transformation 3.4.1, hence in many cases these two transformation would be applied together.

Theorem 3.4.3. *For multiplication applying SCX Transformation 3.4.2 where both inputs of the multiplication are scoring variables of constant-ones, the manipulation is conflict free for Join-Bayes queries of PRA if: 1) both inputs for Join are distinct relations; and 2) the scoring variable with more grouping attributes is chosen as the replacement.*

Proof. If the inputs of a Join are distinct relations, then the cardinality of the Join result fully depends on the input which has more arity, i.e. has more attributes (or columns); on the other hand, the arity of a distinct relation can be reflected from the grouping attributes of a scoring variable.

For Theorem 3.4.3, the original SCX of Transformation 3.4.2 can be specialised into the form of $(R1.[X].c[1.0] * R2.[Y_e].c[1.0])/R2.[Y_e].w$, which indicates the cardinality of the result is $|R(\beta)^{s}_{\mu:\beta\Theta\gamma}|$ where $\beta = \max(X, Y_e), \gamma = \min(X, Y_e)$. According to Theorem 3.4.3, the transformed SCX is $R.[\beta].c[1.0]/R2.[Y_e].w$, which indicates the same cardinality of the result as the original SCX. In addition, the mappings of scores can be guaranteed by Join condition. Therefore, manipulations applying Transformation 3.4.2 for Join-Bayes queries of PRA are conflict free, and thus Theorem 3.4.3 is sound. □

Moreover, Theorem 3.4.4 is given to support conflict free manipulations to be performed for PRA expressions in a form of Project-θ-Bayes query.

Theorem 3.4.4. *For PRA expressions in a form of Project-θ-Bayes query, where θ is any PRA operator except for aggregated projection and Bayes, and disjointness is given as probabilistic assumptions to Project and Bayes, a SCX manipulation applying Transformation 3.4.3 is conflict free.*

Proof. If SCXs are generated for PRA expressions of Project-θ-Bayes queries, the scoring function implied by the unspecified PRA operator θ must become a part of the numerator of a division (generated from Bayes) and it must indicate constant-ones. Because the cardinality of the final result is decided by the projection: let R_θ be the intermediate result yielded by operator θ, so that its cardinality is $|R_\theta|$; and let X_p be the projected attributes, so the cardinality of the projection result is $R_\theta(X_p)^s$.

Here, the original SCX of Transformation 3.4.3 can be specialised as $acc.[X_p].w + (R_\theta.[Y_e].c[1.0]/R.[Y_e].w)$, so that the transformed SCX is $(acc.[X_p].w + R_\theta.[Y_e].c[1.0])/R.[Y_e].w$, both SCXs indicate the number of scores are decided by the number of groups of accumulations, which is equal to $R_\theta(X_p)^s$, and the mappings of scores to tuples are the same. As a result, Theorem 3.4.4 is sound. □

So far, we have verified the manipulations on SCX for interpreting the df_D_t plan are conflict free according to Theorem 3.4.2, Theorem 3.4.3 and Theorem 3.4.4.

To summarise how scoring-driven optimization assists index selection, the interpreted SCXs for *tf*s and *df* articulate exactly how IR statistics (see Table 3.10) are modelled in PRA expressions, therefore, a query engine may exploit interpreted SCXs as one of the considerations while mapping logical PRA operators to physical implementations.

3.4.4.3 Aligning Scoring Function under Extensional Semantics

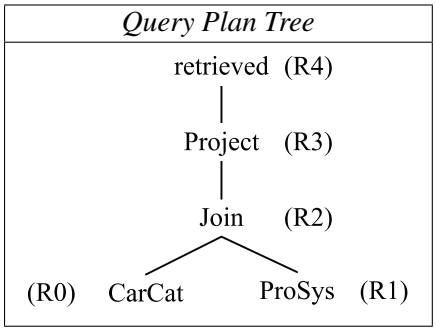
The query evaluation for conjunctive queries (or more precisely, for disjunctive-conjunctive combined queries) in PRA is an problem with respect to both efficiency and effectiveness. As previously discussed, because PRA was originally proposed under intensional semantics and probabilistic events are considered as independent events only (see [Fuhr and Rölleke, 1997], also see Section 2.4.2). However, it had been known that intensional evaluation of PRA expressions (or in general, for relational expressions on probabilistic databases) for conjunctive queries are

inefficient and impractical for processing large-scale data set.

Being aware of the problems, further work was completed by [Roelleke et al., 2008], which pushes the research of PRA a few steps forward and proposes several extensions on PRA: besides a Bayes operator for probability estimation, it also suggests using extensional evaluation for PRA expressions while given probabilistic assumptions. Such attempt is supported by the researches of IR theory on ranking models, where nearly all popular and well-performing (in terms of effectiveness) IR models suggest that terms in documents should not be viewed as independent events, in other words, either reoccurred terms or different terms exist some kind of dependencies. In addition, by employing probabilistic assumptions, [Roelleke et al., 2008] shows that the efficiency and scalability of a PRA query engine can be improved dramatically.

```
retrieved = PROJECT INDEPENDENT [$Class] (
   JOIN INDEPENDENT [$CarType = $VehType] (CarCat, ProSys));
```

(a) PRA expression for a conjunctive query



(b) PRA operator tree

Plan Tree Node	*Generated SCX*	*Interpreted SCX*
R4 (retrieved)	this.s = R3.p	R4.p = 1 − (acc.[$Class].p ∗ (R0.p ∗ (1 − (acc.[$VehType].p ∗ (1 − R1.p)))))
R3 (PROJECT)	this.s = ∨ R2.p	R3.p = 1 − (acc.[$Class].p ∗ (R0.p ∗ (1 − (acc.[$VehType].p ∗ (1 − R1.p)))))
R2 (JOIN)	this.s = R0.p ^ R1.p	R2.p = R0.p ∗ R1.p
R1 (CarProSys)	this.s = p	R1.p = R1[ProSys].p
R0 (CarCat)	this.s = p	R0.p = R0[CarCat].p

(c) Articulating scoring functions with SCX (read bottom-up)

Figure 3.25: Scoring expressions for conjunctive Project-Join PRA expression

From the DB community, [Dalvi and Suciu, 2004] proposed a safe-plan query evaluation technique on probabilistic database based on extensional semantics, in which tuples are viewed as independent events only. Moreover, they argued that unsafe query plans for conjunctive-disjunctive combined queries, i.e. queries include a join inside a projection, compute incorrect

probabilities.

Except for the safe-plan technique proposed by [Dalvi and Suciu, 2004] and the usage of lineage technique introduced by [Benjelloun et al., 2006a], another potential research direction is to study the differences of the rankings produced by approximate methods (such as [Roelleke et al., 2008]) and accurate functions (such as [Dalvi and Suciu, 2004]). However, the studies on ranking equivalence (see Section 3.3.2) would be a direction of future work. On the other hand, instead of studying ranking equivalence, SCX may also apply to adjust scoring functions for evaluating conjunctive queries in PRA.

For instance, an example is given in Figure 3.25, in which both *CarCat* and *ProSys* are the aliases of probabilistic relations in Table 2.2, where *CarCat* is an alias of table *CarCategory* and *ProSys* is an alias of table *CarPropulsionSystem*. The PRA expression of the query is shown in Figure 3.25a, while Figure 3.25b illustrates a query plan which is referred as the conjunctive plan, and a bottom-up articulating process is demonstrated in Figure 3.25c.

The articulating process follows the rules in Table 3.5 to generate initial SCXs for nodes R0 and R1, which both represents probabilistic relations. The next visiting node is R2 when following post-order traversal, while articulating process generates a logic-style SCX for the node because it is a Join operations for two probabilistic scores, which indicates a conjunctive query for joint probability, and the SCX is initialised as $R0.p \wedge R1.p$; furthermore, because an independence assumption is given to the Join, hence the logic-style SCX is interpreted by an arithmetic-style SCX as a multiplication of independent event probabilities, i.e. $R0.p * R1.p$.

And then the process moves up to node R3, where it encounters a projection for aggregation while given an independence assumption. Initially, an generated SCX is produced using logic-style SCX again as $\vee R2.p$. Specially, since the child-node of R3 is a conjunctive query, so that R3 indicates a disjunctive-conjunctive combined query. In order to reflect the intensional semantics of the scoring function of R3, the process replaces $R2.p$ by its initial logic-style SCX instead of interpreted SCX as usual, so that it obtains a SCX as $\vee(R0.p \wedge R1.p)$, which exactly presents the intensional semantics; in the next step, interpreting algorithm aligns the scoring function (based on intensional semantics) to extensional semantics by manipulating SCX operator tree applying Transformation 3.4.4; as the transformation suggests, an SCX tree can be rotated towards either clockwise or anticlockwise, here because the projected attribute is $\$Class$, so a clockwise rotation should be performed, and we will explain more details

later, while a transformed SCX is obtained as $R0.p \hat{} (\vee R1.p)$. For now, the interpreter algorithm is about to convert the logic-style SCX into arithmetic-style SCX, and aforementioned, a probability accumulator would be appended to the left-child node for a left-hand-side operand missing logical operator, and similarly an accumulator would be added into an arithmetic-style SCX as well. Because independence assumption is applied, an interpreted SCX is formulized as $1-(acc.[\$Class].p*(R0.p*(1-(acc.[\$VehType].p*(1-R1.p)))))$, and this is obtained in the following steps:

1. replace the logic-style SCX sub-tree $acc.p \vee R1.p$ by $1-(acc.[\$VehType].p*!R1.p)$;
2. replace $!R1.p$ by $1-R1.p$, which converts the previous SCX sub-tree into $1-(acc.[\$VehType].p*(1-R1.p))$;
3. replace the logical AND operator (i.e. $\hat{}$) in the previous SCX tree, i.e. $R0.p \hat{} (1-(acc.[\$VehType].p*(1-R1.p)))$, to multiplication (i.e. $*$), which transforms the SCX tree into $R0.p*(1-(acc.[\$VehType].p*(1-R1.p)))$;
4. create new arithmetic SCX tree for probability aggregation based on independence assumption, append the previous SCX to the new SCX tree, so that the final interpreted SCX tree is obtained.

3.4.4.4 Verifying Scoring Equivalence

Previously, in Section 3.3.2 we addressed the definitions of equivalence of PRA expressions, where for PRA expressions to be equivalent, they must satisfy relevance equivalence and one type of equivalence with respect to scoring semantics or ranking semantics equivalence. In this part, we discuss how to exploit SCX to verify whether PRA expressions are equivalent under strict scoring semantics.

Let us consider the following two PRA expressions: the first one has already been discussed extensively in previous paragraphs and section, which is a PRA modelling for within-document term frequency ($tf(t,d)$);

```
PROJECT DISJOINT [$Term, $DocId](
        BAYES DISJOINT [$DocId](MagColl));
```

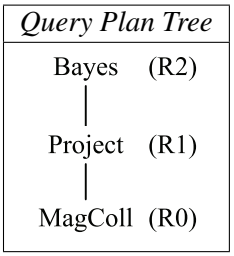
while the second one is very similar to the first expression, the only major change is the order of Bayes and Project has been swapped. A question is that are they equivalent PRA expressions?

```
BAYES DISJOINT [$DocId](
        PROJECT SUM [$Term, $DocId](MagColl));
```

A question is that are they equivalent PRA expressions? So let us articulate the second PRA expression with SCX and find out. The articulating procedure is illustrated in Figure 3.26.

```
BAYES DISJOINT [$DocId] (PROJECT SUM [$Term, $DocId](MagColl));
```

(a) PRA expression



(b) PRA operator tree

Plan Tree Node	*Generated SCX*	*Interpreted SCX*
R2 (BAYES)	this.s = R1.w / (acc.[$DocId].w + R1.w)	R2.p = COUNT(R0.[$* | $Term, $DocId]) / COUNT(R0.[$* | $DocId])
R1 (PROJECT)	this.s = acc.[$Term, $DocId].w + R0.c[1.0]	R1.w = COUNT(R0.[$* | $Term, $DocId])
R0 (MagColl)	this.s = c[1.0]	R0.c = R0[MagColl].c[1.0]

(c) Articulating scoring functions with SCX (read bottom-up)

Figure 3.26: Scoring expressions for Bayes-Project PRA expression of a $tf(t,d)$ equivalence

Let us quickly start from node R1, which is a projection given projected attributes $\$Term$ and $\$DocId$ for aggregation with assumption *sum*; according to SCX generating rule for unweighted relation (see Table 3.3), an initial SCX is created as $this.s = acc.[\$Term, \$DocId].w + R0.c[1.0]$; next, based on Theorem 3.3.3, the SCX is interpreted to be $R1.w = COUNT(R0.[\$* \mid \$Term, \$DocId])$ which uses of symbolic function.

And then, articulating procedure moves up to node R2 for Bayes operator, where a generated SCX is initialised as suggested in Table 3.4, which is $R1.w/(acc.[\$DocId].w + R1.w)$; then next, $R1.w$ is substituted by its interpreted SCX which transforms the previous SCX into $COUNT(R0.[\$* \mid \$Term, \$DocId])/(acc.[\$DocId].w + COUNT(R0.[\$* \mid \$Term, \$DocId])$. By now, we introduce a new theorem to support the interpretation, which is given by Theorem 3.4.5 as follow.

Theorem 3.4.5. *Let R_X be an unweighted relation, where X is the schema of R; let X_a be a list of attribute whose values are to be considered for counting; let both X_b and $\overline{X_b}$ be lists of grouping*

attributes, where $\overline{X_b} \subseteq X_b \subseteq X$; so that the result of counting the number of values of attribute X_a with given grouping attributes $\overline{X_b}$ is equal to the result of aggregate summation for grouped-and-accumulative scores for the result of counting the number of values of attribute X_a with given grouping attributes X_b.

$$\begin{aligned} COUNT(R.[X_a|\overline{X_b}]) \quad &= \quad acc.[\overline{X_b}].w + COUNT(R.[X_a|X_b]) \\ &=_{def} \quad \left(\begin{array}{l} acc.[\overline{X_b}].w = 0 \\ acc.[\overline{X_b}].w = acc.[\overline{X_b}].w + COUNT(R.[X_a|X_b]) \end{array} \right) \end{aligned}$$

Theorem 3.4.5 addresses the situation when a result of counting can be represented as an accumulative summation of another result of counting.

Proof. Let R_{X_b} and $R_{\overline{X_b}}$ be the results of aggregations based on projection, where X_b and $\overline{X_b}$ are projected attributes respectively. Let R_{X_b} to be split into n partitions, and in the i^{th} partition of R_{X_b}, denoted as $R^i_{X_b}$, there exists a many-to-one mapping from $R^i_{X_b}$ to $R_{\overline{X_b}}$, that is, for all tuple values in $R^i_{X_b}$ with respect to attributes $\overline{X_b}$ can only be mapped to one tuple with identical values in $R_{\overline{X_b}}$, i.e. $\sum_{i=1}^{m} 1 = \sum_{i=1}^{n} \sum_{j=1}^{k} 1$ where $m = \sum_{i=1}^{n} k$. Thus Theorem 3.4.5 is sound. □

Finally, an interpreted SCX is obtained which is identical to the one articulating the original PRA expression for modelling $tf(t,d)$.

3.5 Experiments and Results

In this section, we present the experiments for evaluating runtime performance of a IR+DB prototype called Birdie (see Appendix A), in which the proposed scoring-driven optimization mechanism was implemented and applied.

At the moment, we noticed that there are no existing benchmarks which are specific for measuring the performances of integrated IR and DB systems. Some previous researches chose to examine integrated IR and DB experimental systems by the TPC-H (e.g. see [TPC, 2005] (for decision support applications)) database benchmark, which might be suitable for assessing the efficiency of applied systems based on IR-on-DB or middleware architectures (see Section 2.5.1). On the the hand, traditional IR benchmarking methods tend to apply real-world data

(e.g. TREC[9] and INEX[10]) which better reflect the characteristics of textual data, which is in contrast to database convention that prefer synthetic data set. Hence, we conducted the experiments with self-defined but carefully considered measurements.

In addition, the query execution engine of Birdie is tuned to utilise relational inverted index (RIX) that will be discussed later in Chapter 5, while the experiments mainly demonstrated the retrieval efficiency of Birdie while applying scoring-driven optimization for assisting index selection (see Section 3.4.4.2). Therefore, though RIX plays an important role in speeding up retrieval performance, but the proposed scoring-driven optimization is a critical technique that automatically drives query execution engine to choose wisely physical operations and indexes.

3.5.1 Specifications and Setup

First of all, let us introduce the experimental specifications and setup.

Systems The testing bed is an IR+DB prototype named Birdie (see Appendix A). The computer hardware and software specifications of hosting Birdie are as follows:

- Hardware: Dell XPS M1330 Laptop, equipped Intel[11](R) Core[12](TM)2 Duo CPU T6400 at frequency 2.00GHz, 3.00 GB of RAM at frequency 1.20 GHz.
- Operating System: Windows XP Professional, version 2002, Service Pack 3.

Test Collection We used TREC-3 [Harman, 1994] document corpus as the testing collection. Its original data size is about 2.1 GB. In addition, the titles of TREC topics 151-200 were used as queries, where the average query length is 3.64 terms.

Setup The original text documents were pre-processed and stored in a relational table to be used by Birdie, and the schema of the table is given as follow:

```
CREATE TABLE trec3 (term VARCHAR, docid VARCHAR);
```

The data size of the table is about 4.02 GB, and the table consists of 202 254 542 tuples (i.e. over 202 million), in which includes 715 649 distinct terms (keys) and 741 647 documents (groups). A RIX index is built by given the attribute "term" as primary indexed key, while the index size is about 3.97 GB.

[9] http://trec.nist.gov/
[10] http://inex.is.informatik.uni-duisburg.de/
[11] Intel is a registered trademark of Intel Corporation.
[12] Core is a trademark of Intel Corporation.

To execute the queries, we implemented scoring strategies for $P_C(t|d)$, $P_C(t)$, *df*, *tf-idf* model (see Formula 2.1 in page 32), and language modelling (see Formula 2.13 in page 34) in PRA. For the actual PRA expressions implementing *tf-idf* and LM which were used in the experiments, interested readers can refer Section A.3.2 in Appendix A.

A table *qterm* was defined to store the query terms which schema is given as follow:

```
CREATE TABLE qterm (term VARCHAR, qid VARCHAR);
```

The 50 queries (each query consists of several terms) were inserted into table *qterm* during query time, and the scoring strategies and IR models were executed repeatedly for every queries.

3.5.2 Methodology

In the experiment, we investigated the effectiveness of scoring-driven optimization (SDO) based on SCX from two angles. Firstly, we measured the query processing time while utilising SDO in the query engine of Birdie, where 50 TREC queries were executed in a batch mode and the retrieval times while applying different scoring models were recorded. Secondly, in order to demonstrate how SDO may improve the efficiency of query processing, we disabled SDO from the query engine and re-executed the queries, and the retrieval runs without SDO is marked by Non-SDO. The query processing time of SDO runs and Non-SDO runs are compared based on the selectivity of queries.

Range of Selectivity	*Number of Queries*
$< 10\,000$	6
$10\,000 - 99\,999$	15
$1000\,000 - 199\,999$	11
$200\,000 - 299\,999$	10
$> 300\,000$	8

Table 3.11: Selectivity of the 50 TREC queries

Here selectivity is defined as following:

$$Sel(q) := \sum_{i=1}^{n} |\mathcal{R}(t_i)^m|,\ t_i \in q \tag{3.1}$$

where q stands for a query that consists of a number of terms t, and $|\mathcal{R}(t_i)^m|$ is the cardinality of term t_i in relation $\mathcal{R}$ (see Section 2.5.2.1), i.e. the total number of occurrences in the whole collection, so that the selectivity of a query, i.e. $Sel(q)$, is defined as the summation of the cardinality of all query terms.

While SDO is disallowed from Birdie, the query engine can still utilises TID index, however, the performance of query processing would decrease sharply. Therefore, to execute all 50 queries is impractical and unnecessary for demonstration purpose. As a result, we handpicked four queries from the query set that the selectivity of the chosen queries increments gradually. Some statistics of selectivity for the 50 TREC queries and the handpicked queries are demonstrated in Table 3.11 and Table 3.12 respectively. In particular, query 190 has the maximum selectivity 628 185.

Query ID	*Marked Selectivity*	*Actual Selectivity*
151	4 000	3 927
178	6 000	6 084
199	8 000	8 325
162	10 000	10 587

Table 3.12: Selectivity of the handpicked queries

Though efficiency is the main focus in the experiment, while the results were evaluated according to TREC[13] evaluation, where precision scores of the retrieval results are reported.

3.5.3 Results

The results of 50 TREC queries using title only is presented in Table 3.13. Although the selectivity of the 50 queries range from 3 927 to 628 185, but Birdie was able to process these queries efficiently while SDO was enabled. For instance, the average runtime for computing $P_C(t)$ and *df* are both 0.002 seconds only, while the average runtime for estimating $P_C(t|d)$, *tf-idf*, and LM are 2.669, 4.258 and 4.75 respectively, which are all in acceptable range while considering the size of data and computer hardware specifications. Moreover, the retrieval effectiveness using *tf-idf* model are MAP 0.1192 and P@10 0.212, while the effectiveness while applying LM are MAP 0.1873 and P@10 0.362.

In addition, the results of SDO runs versus Non-SDO runs with handpicked queries are given in Table 3.14 and Figure 3.27. The results demonstrate that scoring-driven optimization improved the query processing efficiency with a number of orders of magnitude for estimating popular IR models based on a very large data set.

Figure 3.27 illustrates the retrieval time for using relatively less selective queries. On the one hand, Figure 3.27a indicates Birdie is able to handle such queries in sub-seconds when SDO is

[13] http://trec.nist.gov/

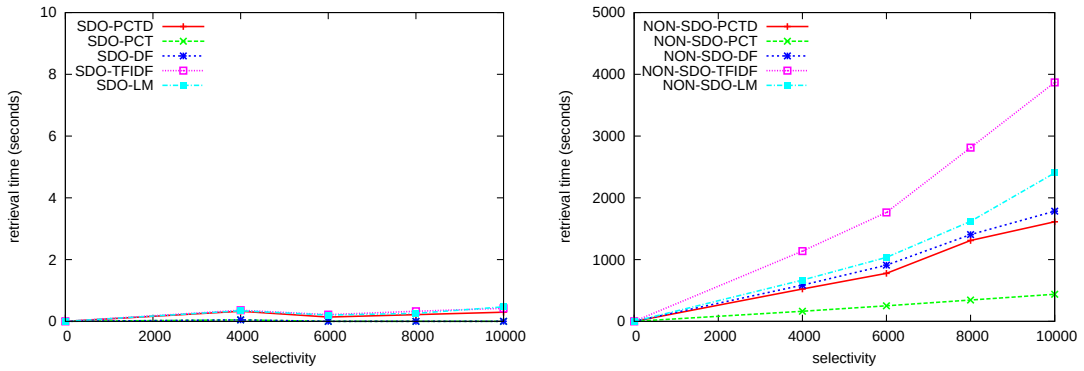
Scoring Model	*Retrieval Time (sec)*			*Effectiveness*	
	avg	*min*	*max*	*MAP*	*P@10*
$P_C(t\|d)$	2.669	0.141	9.344	–	–
$P_C(t)$	0.001	0	0.047	–	–
df	0.001	0	0.047	–	–
tf-idf	4.258	0.219	14.766	0.1192	0.212
LM	4.75	0.203	16.016	0.1873	0.362

Table 3.13: Retrieval time and effectiveness of Birdie with scoring-driven optimization, 50 queries, TREC topics 151-200 using title only

Scoring Model	*Retrieval Time (sec)*					
	SDO			*Non-SDO*		
	avg	*min*	*max*	*avg*	*min*	*max*
$P_C(t\|d)$	0.246	0.141	0.328	1056	524	1612
$P_C(t)$	0.012	0	0.047	299	162	439
df	0.012	0	0.047	1171	587	1784
tf-idf	0.332	0.219	0.422	2396	1138	3869
LM	0.319	0.203	0.469	1431	669	2403

Table 3.14: Retrieval time of Birdie, SDO vs. non-SDO, four handpicked queries, TREC topics 151, 178, 199 and 162 using title only

enabled. On the other hand, Figure 3.27b shows that retrieval time increases dramatically when SDO is switched off.



(a) Query Processing with SDO

(b) Query Processing without SDO

Figure 3.27: Retrieval performances based on selectivity, SDO vs. non-SDO

3.6 Summary

In summary, in this chapter we introduced a scoring-driven optimization technique based on scoring expression (SCX) for probabilistic relational algebra (PRA). Because PRA incorporates

probability estimation and aggregation capabilities along with relational operations (such as traditional relational algebra), hence in order to optimize logical PRA expressions, a practical optimization technique must consider the probabilistic or scoring semantics as well as relational semantics of PRA expressions.

Therefore, this chapter contributes in the following aspects which were neglected by the state-of-the-art optimization techniques for conventional non-probabilistic databases, these aspects are:

- In order to include scoring and ranking semantics to be considered by optimization techniques for PRA or similar variants, we proposed scoring equivalence and ranking equivalence, which extends traditional relational equivalence that is relied on by conventional algebraic optimization techniques for databases.

- We proposed scoring expression (SCX) which can be used for articulating scoring semantics of PRA expressions. We introduced a set of comprehensive syntax for SCX so that the semantics of SCX can be easily understood and handled by either machines or human users.

- We proposed a scoring-driven optimization technique based on SCX. Specifically, we not only introduced the methods for designing a rule-based and automatic scoring-driven optimizer for PRA, but also discussed the how to utilise the optimization to assist index selection, align implied scoring function of PRA between intensional and extensional semantics, and verify scoring equivalence of PRA expressions.

In addition, experiments were performed to evaluate the optimization technique, where respectable results showed that it is capable to speed up the runtime performance of processing PRA queries.

Chapter 4

TIP: Query Processing with Top-*k* Incorporated Pipeline

4.1 Introduction

Top-*k* query processing is a fundamental building block for information retrieval and databases, and it has been considered as a crucial technique for the integration of IR and DB. In short, top-*k* processing aims to provide an *early stop* or *early response* functionality for ranked retrieval with a little and acceptable retrieval effectiveness (quality) loss. In general, top-*k* methods are necessary for modern retrieval systems from two points of views. On the one hand, from a system-centric point of view, the size of data sets in general, e.g. the Web or corporate databases, have being grown geometrically; while handling very large data sets, retrieval systems are likely to return *too many answers* if there were not any effective "cut-off" methods. On the other hand, from a user-centric point of view, it has become a de facto standard for modern IR systems to be able to response within sub-second; especially, for those IR applications that involves human users' interactions, efficiency is the first criterion that decides whether a retrieval system would be competitive.

Therefore, various top-*k* methods or algorithms had been proposed by researchers from either IR community or DB community during past decades. Without losing generalisation, we may roughly categorise top-*k* mechanisms in two groups: 1) methods to be performed during indexing or preprocessing; and 2) algorithms to be applied during on-the-fly query processing. For the first category, methods usually perform (static) index pruning based on sophisticated settings, where tails of (very) long posting lists (see e.g. Chapter 5, Section 5.1.2) of indexes would be

deliberately discarded, and only those relatively likely relevant items (i.e. top-k items) would be kept. While for the second category, algorithms were designed to obtain weighted items that with the highest scores by aggregations or certain ranking functions, where processes could be terminated as long as top-k items were yielded.

In the chapter, we mainly focus on methods belong to the second category, in which we study how top-k algorithms could be incorporated into a pipelined query processing engine of IR+DB system. Specifically, we investigate whether a well-known family of algorithms, i.e. the family of *threshold algorithm* (TA) (see e.g. [Fagin et al., 2001, Nepal and Ramakrishna, 1999, Güntzer et al., 2000]), or its variants could be applied during pipelined query evaluation. Note that previous studies on TA or its variants were based on middleware systems (see Section 2.5.1), where TA (or other families) were implemented in an external layer of databases; while it is desirable to incorporate TA algorithms into a generic query processor of IR+DB system (see e.g. [Chaudhuri et al., 2005]).

Outline The remainder of the chapter is organised as follows: Section 4.2 reviews the backgrounds of top-k processing, which include the computational model with an example, and an introduction about TA and the variants, plus a range of other related work. In Section 4.3, we address a conceptual design of top-k incorporated pipeline, and an investigation on performance tradeoff with respect to efficiency and effectiveness. Moreover, the experiments and results will be presented in Section 4.4. Finally, the chapter is summarised in Section 4.5.

4.2 Background

4.2.1 Computational Model

Here we review a well-known family of top-k algorithms. As aforementioned, the threshold algorithm (TA) [Nepal and Ramakrishna, 1999, Güntzer et al., 2000, Fagin et al., 2001] is the best known general-purpose algorithm for evaluating top-k queries, and several variants of classical TA (under weak assumptions) were proposed for specific applications or retrieval under certain circumstances. To get started, we first introduce the original TA, and then our focus would be on the approximate top-k algorithms that were dedicated for IR applications.

First of all, the general settings while applying TA algorithm and its families are discussed, which include computational model, accessing modes, and a typical scenario.

Data Model Consider a *Cartesian product space* $D_1 \times \ldots \times D_m$ over domains $D_1, \ldots, D_m$, and a data set $\mathcal{D} \subseteq D_1 \times \ldots \times D_m$ of m-dimensional data items. Data items could be structured records, semistructured or text documents, which contain a set of attribute values or terms that produce an m-dimensional space. Each tuple of structured record or term-document pair is associated with a numeric score (i.e. weighted tuple) that represents the relevance of the data item with regards to the value or term; in other words, for each domain D_i there is a similarity function $f_i : D_i \times D_i \rightarrow [0, +\infty)$, i.e. scores are positive real numbers that could be either raw weights, (normalised) frequencies, or probabilities. Top-k queries are essentially partial-match queries on the m-dimensional Cartesian product space: queries are in the form of m-tuples $(q_1, \ldots, q_m)$ where $i = 1, \ldots, m$, $q_i \in D_i$ if the query matches the i^{th} dimension value or $q_i = *$ if the i^{th} dimension value is not considered; then let N be the total matches that can be found for a query, so a top-k query is to find at most k matches from the space where $1 < k \leq N$ (usually $k \ll N$). In this case, conditions are given for matching values from the space against queries, and approximate matching is allowed while certain mechanisms are available to guarantee the quality of query result. Moreover, the aggregations or ranking functions are often assumed to be monotonic.

Aggregate or ranking functions would be applied to compute the scores of result of conjunctive queries, which aggregate domain-specific similarities, i.e. $s : (D_1 \times D_m) \times (D_1 \times D_m) \rightarrow [0, +\infty)$, and with $s(x, y) = agg\{s_i(x_i, y_i) | i = 1, \ldots, m\}$. A aggregate function could be as simple as a summation, or applying sophisticated normalisation such as the saturation function in BM25 (e.g. see Section 2.2.2). For instance, while deploying summation then the aggregated similarity is given by $s(x, y) = \sum_{i=1}^{m} s_i(x_i, y_i)$.

Accessing Modes In [Fagin et al., 2001], the authors suggest two types of access to data, which are *random access* (RA) and *sorted access* (SA). The former mode requires an accessing key to be available, so that a tuple could be retrieved from a list of tuples instantly by probing the list using the key; whereas the latter performs a sequential scan on a sorted list, if there are more than one list, then SA would also be able to scan multiple lists in parallel. Moreover, [Fagin et al., 2003a] proposed *no random access* (NRA) for the settings of text retrieval where random accesses are unavailable or too expensive to be practical with respect to processing costs.

Query Processing The query processing of top-k algorithms may perform single accessing mode to data, for instance, to employ SA only. Alternatively, some algorithms may apply mixed accessing modes while sophisticated scheduling strategies could be used to switch among dif-

ferent modes. For example, the Fagin's Algorithm (FA) [Fagin, 1999] and TA schedule data accessing mode between SA and RA.

During top-k query processing, certain bookkeeping methods would be used to remember have seen yet items and possible candidates. Without loss of generality, let us consider values υ in domain $D_i, i \in 1,\ldots,m$ as indexed keys, there is an associated list of tuples τ that can be retrieved by each key. Usually, the following notations are used to denote scores for bookkeeping and termination of the algorithm.

- $s(\tau)$: denoting the score of tuple τ, which is the final score of τ by aggregating its different tuple weights in different lists;
- $E(\tau,\mathcal{L})$: denoting the tuple weights of τ that have been seen yet, where $\mathcal{L} = \{\ell_i\}$ and $i \in \{1,\ldots,m\}$ are the lists where the weights are seen, whereas $\overline{\mathcal{L}}$ denotes the lists in which the weights of τ have not yet been seen;
- $s(\tau,\ell)$: denoting the tuple weight of τ in list ℓ;
- $s_{worst}(\tau)$: denoting the *worst score* of tuple τ, which is the aggregated score of τ that obtained from the yet have seen tuple weights, i.e. $s_{worst}(\tau) = \sum_{\ell_i \in \mathcal{L}} s(\tau,\ell_i)$;
- $s_{high}(\ell,p)$: denoting the *highest* score in list ℓ at position p, which is obtained while performing NRA on list ℓ when a scanning iterator reaches position p;
- $s_{best}(\tau)$: denoting the *best score* of τ that can be reached, which is obtained by $s_{worst}(\tau) + \sum s_{high}(\ell_i,p)$ where $i \in \{1,\ldots,m\}$ and $\ell_i \in \overline{\mathcal{L}}$;
- $s_{unseen}(p)$: denoting the upper bound of aggregated score at position p of all lists, which is obtained by $\sum_{i=1}^{m} s_{high}(\ell_i,p)$.

4.2.2 Typical Scenario and Example

A typical scenario in IR is to aggregate the within-document *tf* of multiple query terms, which is addressed as follow:

Example 4.2.1. Given three query terms "hybrid", "car" and "fuel", where each term associates to an inverted document list, which contains a list of *DocIDs* representing the documents in which the term occurs; and for each *DocID*, there is a corresponding (normalised) score that indicates

the within-document *tf* of the term; moreover, the lists have been sorted in descending order by *tf*. The inverted document lists are illustrated in Figure 4.1. Assuming summation is applied as the aggregate function, try to retrieve the top k documents from the lists where $k = 2$, and stop the process as soon as possible.

Term = "hybrid"

pos	$P_C(t\|d)$	*DocID*
1	0.9	d78
2	0.8	d23
3	0.8	d10
4	0.4	d1
5	0.2	d88
6	0.2	d14
7	0.2	d25
8	0.1	d83
9	0.1	d17
	...	

(a) List One

Term = "car"

pos	$P_C(t\|d)$	*DocID*
1	0.8	d64
2	0.8	d23
3	0.7	d10
4	0.7	d1
5	0.3	d25
6	0.2	d45
7	0.2	d14
8	0.1	d12
9	0.1	d78
	...	

(b) List Two

Term = "fuel"

pos	$P_C(t\|d)$	*DocID*
1	0.9	d10
2	0.9	d78
3	0.8	d64
4	0.2	d99
5	0.1	d34
6	0.1	d22
7	0.05	d18
8	0.05	d35
9	0.02	d67
	...	

(c) List Three

Figure 4.1: Sorted lists of probabilities

Example 4.2.1 specifies a 2-dimensional space of two domains, i.e. $\mathcal{D} \subseteq D_t \times D_d$, where D_t represents a domain of terms (i.e. keywords) while D_d stands for a domains of documents. The result should contain two documents (*DocIDs*) with the highest similarity score of $sim(q,d)$.

A naive way to answer the query in the above example is to process all lists and obtain a list of documents with aggregated scores, then sort the list of documents in descending order according to their aggregated scores, and then select the two documents at the highest two positions from the list. Apparently, such naive method fails to meet the requirement "stop the process as soon as possible", and it is hardly scalable if the size of input lists are very large. Notice that the input lists have already been organised in certain order, therefore, sophisticated algorithms have been designed to take advantage of sorted input lists so that to achieve more efficient ways for processing top-k queries. In the next section, we review a successful family of algorithms for top-k processing, and demonstrate how these algorithms can handle queries such as the one in Example 4.2.1.

4.2.3 Family of Threshold Algorithms

In this subsection, we review some well-known algorithms for on-the-fly top-k processing. Some of the algorithms were originally proposed for database middleware systems for multi-media applications, for example, the FA and the TA (see e.g. [Nepal and Ramakrishna, 1999,

Güntzer et al., 2000, Fagin et al., 2001]); some were for cases where random accesses are either impossible or expensive relative to sorted access such as for text retrieval, while NRA and a combined algorithm (CA) of NRA and TA were introduced (see e.g. [Fagin et al., 2003b]); and some were designated for IR applications such as XML retrieval, for instance, approximate top-k with probabilistic guarantees and IO-Top-k (e.g. see [Theobald et al., 2004, Theobald et al., 2005b, Bast et al., 2006]).

Fagin's Algorithm First of all, the FA can be described as the following steps:

1. Do sorted assess in parallel to each of the m sorted lists ℓ_i, aggregate the tuple weights of the same items, continue the process until for each of the m lists there are at least k items have been seen, meanwhile bookkeeping for candidates would be performed;

2. For each of the k candidates, assuming an arbitrary item τ has been seen in lists $\mathcal{L} = \{\ell_i\}$, do random access to lists $\overline{\mathcal{L}} = \{\ell_j\}$ (where $\forall \ell_j \in E(\tau, \overline{\mathcal{L}}), \ell_j \notin E(\tau, \mathcal{L})$) to retrieve the tuple weights of item τ and aggregate the score for $s(\tau)$;

3. Return the top k items that their scores, and then FA halts;

In general, FA separates the process into two phases, the first phase does sorted access only, while the second phase does random access only.

Threshold Algorithm Different from FA, TA interleaves SA and RA, so that The algorithm is described as follows:

1. Do sorted assess in parallel to each of the m sorted lists ℓ_i where $i = 1, \ldots, m$. Once an item τ is seen under sorted access in a list, do random access to other lists and aggregate the tuple weights of the item for $s(\tau)$. Keep the item's identifier to bookkeeper;

2. If the number of top candidates is less than top k limit, then add the item to candidates; otherwise, if the item's final score is greater than the minimum score of candidates, then replace the minimum candidate by the item;

3. While sorted access is at position j of the m sorted lists, compute the upper bound score (threshold) of unseen items as $s_{unseen}(p_j) = \sum_{i=1}^{m} s_{high}(\ell_i, p_j)$. Stop sorted access if $s_{unseen}(p_j)$ is less than the minimum score of top k candidates;

4. Return the top k items and their score, and then TA halts;

To demonstrate the algorithm, we use Example 4.2.1 and explain the process. Because SA would be performed in parallel, in other words, TA visits each list in a round-robin manner and retrieves an item at the same relative position from a list. For explanation, the pseudo codes of TA is given in Figure 4.2.

Algorithm: TA
Input: Multiple sorted lists
Output: A list of top-*k* items with highest aggregated scores

```
1  begin
2    set iter = 0; /* initialise sorted access iterator */
3    set upper = 0; /* initialise upper bound */
4    set seen = new HashTable(); /* seen tuples */
5    set candidates = new PairList(tuple, score); /* top candidates */
6    foreach list in lists
7      /* do sorted access on list */
8      item = list[iter]->tuple;
9      upper += item->weight;
10     if seen->Contains(item) is false
11       seen->Add(item);
12       seen[item]->score = item->weight;
13     endif

15     /* do random access for item to all other lists */
16     foreach otherList in lists where otherList is not equal list
17       seen[item]->score += otherList[iter]->weight; /* aggregate */
18     endforeach

20     /* update the lower bound of top candidates */
21     if candidates->Count is less than limit
22       candidates->Add(item, seen[item]->score);
23     else if seen[item]->score is greater than candidates->MinScore
24       candidates->ReplaceMinItem(item);
25     endif

27     /* update sorted access properties or terminate TA */
28     if upper is less than candidates->MinScore
29       break; /* done */
30     else
31       upper = 0; /* reset upper bound of unseen items */
32       iter++; /* move down sorted access iterator */
33     endif
34   endforeach
35   return candidates;
36 end
```

Figure 4.2: Threshold Algorithm

Here query terms “hybrid” “car” and “fuel” correspond to lists one, two and three respectively. In the first round, the process sorted accesses items at position 1 of every lists. For instance, “d78” is retrieved from list one, and then random accesses would be issued to lists two

and three to retrieve the tuple weights of "d78" on these lists, so the aggregated score of "d78" is computed as $s(d78) = 0.9 + 0.1 + 0.9 = 1.9$. Similarly, "d64" and "d10" would be retrieved in the same manner and their scores could be obtained, which are $s(d64) = 0.8 + 0.8 = 1.6$ and $s(d10) = 0.8 + 0.7 + 0.9 = 2.4$. Moreover, the process also computes the upper bound of the achievable score at position 1, which is $s_{unseen}(1) = 0.9 + 0.8 + 0.9 = 2.6$. After the first round, the top 2 candidates are "d10" and "d78" so that they would be kept in memory; whereas for "d64", because its score cannot be greater than the minimum top item that has been seen so far, so that it can be discarded since now. Meanwhile, because the upper bound of unseen scores is greater than the lower bound of the current top 2 items, thus the process will be continued.

Similar procedures are repeated till the fourth round while SA is retrieving the lists at position 4, where the upper bound of unseen scores is $s_{unseen}(4) = 0.4 + 0.7 + 0.2 = 1.3$, which is less than the lower bound of the top candidates, i.e. $s(d78) = 1.9$. In other words, the aggregated scores of remaining items cannot exceed the lower bound of the current top candidates, so that the top 2 items have been obtained by now and TA halts.

No Random Access and Combined Algorithm In the cases where random accesses are impossible or too expensive to be performed, [Fagin et al., 2003b] introduced *no random access* (NRA) for such situations. In fact, this is a very common situation in dedicated IR systems, where indexed terms (keywords) may associate to very long posting lists (of inverted documents), in which RA could be either impossible or impractical.

Informally, a *no random access* algorithm can be described as follows[1]:

1. Do sorted assess in parallel to each of the m sorted lists ℓ_i where $i = 1, \ldots, m$. For each seen item τ, first remember the list ℓ_i where τ was discovered into $E(\tau, \mathcal{L})$, and then compute (or aggregate) two scores of τ, i.e. a) $s_{worst}(\tau)$, and b) $s_{best}(\tau)$ (see Section 4.2.1). In addition, also calculate the upper bound at position p for aggregated scores of unseen items $s_{unseen}(p)$;

2. Keep SA until the following situation appears: there are at least k items whose $s_{worst}(\tau)$ are the highest scores, and minimum $s_{worst}(\tau)$ is greater than the $s_{best}(\tau)$ of an item which has not yet been considered as the top k candidates but its $s_{best}(\tau)$ is the maximum score among the seen-but-not-candidate items. Stop the scanning process;

[1]Note that our description is based on our understanding and we considered its feasible for the settings of IR applications, but it may be slightly different from the original description that introduced in [Fagin et al., 2003b].

3. Return the top k candidates and their score, and then NRA halts;

In the worst case, NRA has to scan the entire lists in order to deliver the top-k items.

On the other hand, in cases while RA is not impossible but relatively expensive than SA, [Fagin et al., 2003b] suggested a combined algorithm that balances the usage of SA and RA. In short, cost models are applied to estimate the costs of sorted accesses and random accesses, for example, let c_S be the cost of SA and let c_R be the cost of RA, compute a ratio $h = \lfloor c_R/c_S \rfloor$, so that h is the number of rounds for sorted access to be performed, and then random accesses would be deployed for selective top k items to discovered required $s(\tau, \ell_i)$ where $\ell_i \notin E(\tau, \mathcal{L})$ before RA rounds.

Moreover, the pseudo codes of NRA algorithm is given in Figure 4.3.

Approximate Top-k Processing with Probabilistic Guarantees If we consider the previous algorithms (i.e. FA, TA, NRA and CA) as *exact top*-k algorithms, which means they would retrieve "exactly top-k" results for queries while certain k limits are specified. On the other hand, IR applications are usually less concern about the exact limits of k results, but they would prefer to retrieve approximate k (or k percent of all) results instead. That is because a basic idea of IR models is to estimate the similarities between information items and queries, hence the accuracy of retrieved results is heuristic anyway.

Based on similar consideration, [Theobald et al., 2004] suggested that the aforementioned TA and its variants are overly conservative with regards to stopping criteria. On the contrary, [Theobald et al., 2004] proposed applying SA only algorithms while stopping top-k processes based on the distributions of tuple (data item) weights, in which the terminating point of a process would be estimated by probabilistic score prediction. In this method, the lower bound of top k candidates could be approximated by dynamic programming or histogram. In addition, an XML retrieval system named *TopX* [Theobald et al., 2005b] was developed by applying the approximate top-k mechanisms with probabilistic guarantees.

Top-k Query Optimizations Based on Scheduling Since the data accessing methods of TA related algorithms must involve sequential (sorted) accesses, and random accesses might be applied depending on availability or cost. Therefore, when RA is allowed, the optimization issues for top-k processing becomes a scheduling problem for balancing SA and RA.

In [Bast et al., 2006], the authors of the work took an integrated view of the scheduling issues for IR-style long posting lists, where (scheduling) strategies based on a Knapsack-related opti-

Algorithm: NRA
Input: Multiple sorted lists
Output: A list of top-*k* items with highest aggregated scores

```
1  begin
2    set iter = 0; /* initialise sorted access iterator */
3    set upper = 0; /* initialise upper bound */
4    set seen = new HashTable(); /* seen tuples */
5    set candidates = new PairList(tuple, score); /* top candidates */
6    foreach list in lists
7      /* do sorted access on list */
8      item = list[iter]->tuple;
9      upper += item->weight;
10     if seen->Contains(item) is false
11       seen->Add(item);
12       seen[item]->worstScore = item->weight;
13     else
14       seen[item]->worstScore += item->weight;
15     endif
16     seen[item]->seenFromList->Add(list);

18     /* compute the upper bound of current item */
19     seen[item]->bestScore = 0;
20     foreach otherList in lists where seen[item]->seenFromList->
           Contains(otherList) is false
21       seen[item]->bestScore += otherList[iter]->weight;
22     endforeach
23     seen[item]->bestScore += seen[item]->worstScore;

25     /* update the lower bound of top candidates */
26     if candidates->Count is less than limit
27       candidates->Add(item, seen[item]->worstScore);
28     else if seen[item]->worstScore is greater than candidates->
           MinScore
29       candidates->ReplaceMinItem(item);
30     endif

32     /* update sorted access properties or terminate NRA */
33     if upper is less than candidates->MinScore
34       break; /* done */
35     else if foreach candidate in candidates where candidate->
           seenFromList->count is equal lists->count
36       break; /* done */
37     else
38       upper = 0; /* reset upper bound of unseen items */
39       iter++; /* move down sorted access iterator */
40     endif
41   endforeach
42   return candidates;
43 end
```

Figure 4.3: No Random Access Algorithm

mization for sequential accesses and a cost model for random accesses were developed, which are shown to be outperformed than other less sophisticated scheduling methods.

The basic idea of the a scheduling scenario introduced by [Bast et al., 2006] may be summarise in short as follows:

1. Employ CA-like top-k algorithm, and start with sorted accesses to all lists at the beginning;
2. After a few rounds of SA and based on the yet have seen top candidates, stop sorted accessing on some lists but only continue SAs on the selected lists;
3. Continue dedicated SAs for the most likely top candidates while keep estimating the accessing costs of SA and RA based on a cost model and be ready to switch: for a top candidate, while the estimated cost of RA is less than SA, stop SA and perform RA to retrieve the remaining weights;
4. The process halts after RA rounds.

We may also employ Example 4.2.1 to demonstrate the scheduling.

The process is similar to a conventional combined algorithm, where SAs are performed to the posting lists of query terms "hybrid" "car" and "fuel" in parallel and a round-robin manner, while the lower and upper bounds of aggregated scores of every discovered items and the upper bound of unseen items are computed. For instances, after the first round of sorted accesses, we obtain $\langle s_{worst}(d78) = 0.9, s_{best}(d78) = 2.6\rangle$, $\langle s_{worst}(d64) = 0.8, s_{best}(d64) = 2.6\rangle$, $\langle s_{worst}(d10) = 0.9, s_{best}(d10)\rangle$, and the upper bound of unseen items at position 1 is $s_{unseen}(1) = 2.6$.

After four rounds of SA, the seen items and their worst and best scores are (in 3-tuple $\langle DocId, s_{worst}, s_{best}\rangle$): $\langle d10, 2.4, 2.4\rangle$, $\langle d78, 1.8, 2.5\rangle$, $\langle d64, 1.6, 2.0\rangle$, $\langle d23, 1.6, 1.8\rangle$, $\langle d1, 1.1, 1.3\rangle$, $\langle d99, 0.2, 1.3\rangle$; and the upper bound of unseen items are $s_{unseen}(4) = 1.3$. Because on the list three (i.e. the posting list of "fuel"), the tuple weight drops dramatically, hence a judiciously scheduling strategy may decide to stop scanning on this list. In addition, because the highest scored item is $d10$, which has been seen in all lists; and the second largest item is $d78$, which has not yet been seen in the list two only. Therefore, a scheduled SA may perform a few more steps on the list two; and if it still has not been seen after the additional SA, then an RA will be performed to retrieve the score. In the end, the process is able to find the top 2 items $\langle d10, 2.4\rangle$ and $\langle d78, 1.9\rangle$ while saving considerable costs on sorted accesses and random accesses.

4.2.4 Pipelined Top-*k* Operators in Relational Databases

Apart from the studies of algorithms, another line of research regarding top-*k* processing focus on *engine level* integration of top-*k* operators within relational databases. One of the purposes of such tight coupling approaches is to support pipelining in query processing (e.g. see []), which is a widely used technique in modern database systems for minimising retrieval response time. Previous research include new physical operators (e.g. Rank-Join [Ilyas et al., 2003] and ranking aggregate [Li et al., 2006]), rank-oriented logical algebra (e.g. [Li et al., 2005]), and rank-aware optimization (e.g. [Ilyas et al., 2004] and [Li et al., 2005]).

In [Ilyas et al., 2003], the authors introduced a Rank-Join algorithm and a physical implementation Hash Rank-Join Operator (HRJN) based on this algorithm. The Rank-Join algorithm is similar to NRA [Fagin et al., 2003b] in a way that both algorithms perform only sorted access to get tuples from each data sources, while Rank-Join is different from NRA that it maintains the scores of the completely seen join combinations only, whereas NRA also maintains incomplete scores of partially seen tuples. As a result, the Rank-Join algorithm reports exact scores, whereas the NRA reports bounds on tuples' scores. With regard to implementation, the HRJN is a variant of Ripple Join (e.g. see [Haas and Hellerstein, 1999]) that utilises two hash tables on both inputs and a priority queue, therefore it can be interleaved with other conventional operators in relational databases.

Other examples of pipelined top-*k* join operators include the NRA-RJ operator [Ilyas et al., 2002], and the J^* algorithm [Natsev et al., 2001].

In addition to physical join operators, aggregation is another operation that could be involved in rank-oriented execution plan. For example in [Li et al., 2006], the work studies rank-aware query operators work under top-*k* aggregation, in which introduces two fundamental principles: one is Group-Ranking, and the other is Tuple-Ranking. The Group-Ranking principle dictates the order in which groups are probed during top-*k* processing. In short, during a procedure of incremental consuming tuples from the groups, some groups that are prioritised if they are more likely to achieve possible maximum aggregate values, and some groups could be excluded at some points if their chances for being in top groups are unlikely. Moreover, the Tuple-Ranking principle dictates the order in which tuples should be accessed from each group, where tuples are chosen based on tuple orders.

To reflect the ranking requirements in a query algebra, the RankSQL is proposed in

[Li et al., 2005] that views the ranking of queries as a logical property, similar to the conventional membership property. RankSQL extends traditional relational algebra by interleaving ranking into relational operators such that the orders of intermediate relations are considered. In addition, the work introduces logical query optimization for the algebra, which extends conventional dynamic programming plan enumeration algorithm, where the order property is taken into account in addition to the membership property as enumeration dimensions.

While treating ranking requirements as a physical property of query execution plan, previous work in [Ilyas et al., 2006, Ilyas et al., 2004] study rank-aware query optimization, in which interesting physical order (such as join order) and cost estimation for execution plans using conventional operators or rank-aware operators are investigated.

It is worth mentioning that the TIP technique shares the same perspective with the above techniques, the difference is that the previous studies mainly focused on score aggregation, whereas TIP not only considers score/probability aggregation but also takes into account probability estimation.

4.2.5 Other Related Work

A recent survey [Ilyas et al., 2008] summarises a wide range of top-k processing techniques, which also introduces a taxonomy to classify related techniques based on multiple design dimensions including query model, data access methods, implementation level, data and query uncertainty, and ranking function. Accordingly, our previous discussions on generic algorithms and pipelined operators cover the mostly featured techniques of top-k processing that relate to the TIP technique, while there are some other related work are also interesting and worth mentioning.

While top-k processing methods are implemented at middleware level, the techniques can be classified into Filter-Restart methods and Indexes/Materialised Views methods.

Filter-Restart techniques limits the number of retrieved results which formulate top-k queries as range selections queries, where the limitation is indicated by an estimated cut-off threshold. Incorrect estimation of cut-off threshold leads to either insufficient answers or too many answers. A probabilistic approach to estimate cut-off threshold was proposed in [Donjerkovic and Ramakrishnan, 1999], where a top-k query based on an attribute X is mapped into a selection predicate $\sigma_{X>T}$, where T is the estimated cut-off threshold. Another example [Xin et al., 2006b] introduces a ranking cube approach.

Different from the DB's approaches that apply dynamic estimation, the problem of cut-off

threshold is studied as static index pruning techniques in information retrieval systems. For example, see [Soffer et al., 2001, Büttcher and Clarke, 2006, de Moura et al., 2005].

On the other hand, techniques to utilise specialised indexes and materialised views were proposed to improve query response time where extra storage are relatively inexpensive. Examples of specialised top-k indexes such as the Onion indices [Chang et al., 2000, Xin et al., 2006a] and the Ranked Join indices [Tsaparas et al., 2003]. Moreover, research studied deploying materialised views for top-k processing, which provide efficient assess to scoring and ordering information that is expensive to gather at runtime, for example in [Hristidis et al., 2001, Hristidis and Papakonstantinou, 2004], and in [Das et al., 2006].

In addition, top-k processing methods have been studied in the contexts of specific applying fields.

For instance, the Upper and Pick algorithms [Bruno et al., 2002, Marian et al., 2004] are proposed in the context of Web-accessible sources, such processing methods belong to the category applying sorted access with controlled probes, which assume that at least one source provides sorted access, while random accesses are performed only when necessary. Moreover, controlling the number of random accesses is necessary for efficient query processing, which can be achieved by optimizing the number of times the ranking predicates are invoked. Related efforts have been discussed in [Chang and won Hwang, 2002] and [won Hwang and Chang, 2007], which introduced a Minimal Probing (MPro) algorithm that adopts a concept of "necessary probes" to minimise the predicate evaluation cost.

For another instance, top-k processing in XML data has gained more attention from both DB and IR communities, examples include the TopX system [Theobald et al., 2005b], the XRank system [Guo et al., 2003], and [Marian et al., 2005].

Furthermore, applying parallel and distributed computing for very large data to improve efficiency and scalability of query processing becomes more and more popular, hence the methods for calculating top-k queries over distributed networks have been also studied, for example, see [Cao and Wang, 2004, Yu et al., 2005].

Finally, most earlier work focus on minimising the processing costs to reach a target result quality, while a recent work [Shmueli-Scheuer et al., 2009] tried to tackle a different top-k problem where query processing is given a limited budget in terms of time or access costs. Such consideration could become more and more popular while mobile applications and real-time

analytics keep increasing fast.

4.3 Top-*k* Incorporated Pipeline

In this section, we discuss and investigate the issues relate to integrated top-*k* mechanisms into a pipelined query execution engine of IR+DB system. First of all, we introduce the preliminaries of query evaluation and execution with respects to common forms of queries, designs of physical operators, and pipelined execution plans. And then, we address a conceptual design of top-*k* incorporated pipeline (TIP) that intends to incorporate top-*k* algorithms into query execution plan. Then next, we propose a method for estimating the performances tradeoff with regards to efficiency and effectiveness and we investigate several strategies for NRA-style top-*k* methods.

4.3.1 Preliminary of Execution Plan in Databases and IR+DB Systems

4.3.1.1 Common Query Block

In conventional databases, a typical query block usually consists of selections, projections and joins, which is called *Select-Project-Join (SPJ)* queries; while a complex DB query may be composed by multiple SPJ queries.

Let $\mathcal{R}_{\mathrm{X}}$ and $\mathcal{R}_{\mathrm{Y}}$ be two relations whose schema (i.e. a list of attribute names) are X and Y respectively; and let $\overline{\mathrm{X}}$ be an arbitrary header (i.e. a list of attributes) for conditions or projections, where $\overline{\mathrm{X}} \subseteq \mathrm{X}$. Then a SPJ query is given in a form as follow:

$$\Pi_{\overline{\mathrm{X}} \cup \overline{\mathrm{Y}}}(\mathcal{R}_{\mathrm{X}} \bowtie_{\overline{\mathrm{X}} \Theta \overline{\mathrm{Y}}} \mathcal{R}_{\mathrm{Y}})$$

Because selections (come from join conditions) and projections could be pushed into a join, so that a SPJ query can be actually written as the following algebraic expression:

$$\Pi_{\overline{\mathrm{X}}}(\sigma_{x \Theta \overline{\mathrm{X}}}(\mathcal{R}_{\mathrm{X}})) \bowtie_{\overline{\mathrm{X}} \Theta \overline{\mathrm{Y}}} \Pi_{\overline{\mathrm{Y}}}(\sigma_{y \Theta \overline{\mathrm{Y}}}(\mathcal{R}_{\mathrm{Y}}))$$

Moreover, multiple SPJ query blocks can be further combined by join operations.

Similarly, while writing queries in PRA (see Chapter 2) for modelling scoring functions or ranking models, there is also a common form of query blocks: because probability estimation and aggregation are two basic operations for yielding ranked results with scores (including weights and probabilities), hence we call each PRA query block as a *Select-Estimate-Aggregate (SEA)*

query.

Both probability estimation and aggregation are composed operations: usually, a PRA expression for probability estimation can be formulated in a patterned expression *Project-Bayes*, while for probability estimation one can be written in a pattern using *Project-Join*.

Let ε be a probability estimation operator, for instance, the Bayes operator in PRA, and let the other notations and settings the same as those for SPJ queries, then a SEA is given as follow:

$$\Pi_{\overline{\mathrm{X}}\cup\overline{\mathrm{Y}}}(\varepsilon_{\overline{\mathrm{X}}}(\mathcal{R}_{\mathrm{X}}) \bowtie_{\overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}} \varepsilon_{\overline{\mathrm{Y}}}(\mathcal{R}_{\mathrm{Y}}))$$

Similarly to SPJ query, an alternative formulation of a SEA query can be also written by:

$$\Pi_{\overline{\mathrm{X}}}(\varepsilon_{\overline{\mathrm{X}}}(\sigma_{x\Theta\overline{\mathrm{X}}}(\mathcal{R}_{\mathrm{X}}))) \bowtie_{\overline{\mathrm{X}}\Theta\overline{\mathrm{Y}}} \Pi_{\overline{\mathrm{Y}}}(\varepsilon_{\overline{\mathrm{Y}}}(\sigma_{y\Theta\overline{\mathrm{Y}}}(\mathcal{R}_{\mathrm{Y}})))$$

To combine or aggregate SEA queries, we may employ multiple join or union operations.

Since complex queries can be always divided into sub-queries formed by common query blocks, therefore, with regards to queries we always mean SEA queries in our later discussions.

4.3.1.2 Physical Operators and Pipelined Execution Plan

In traditional databases and IR+DB systems built from scratch, physical operators are the implementations of logical operators (of logical algebra), where the relationships between physical and logical operators are relationships of many-to-one mappings (e.g. see [Graefe, 1993]).

An execution plan is a sequence of physical operators, where a physical operator consists of one or more elementary operations. If we denote an elementary operation as *EOP* (i.e. Elementary Operator), then an operation in an execution plan can be categorised into one of the following three classes (see e.g. [Golfarelli et al., 2002]), which depends on its relationship with other operations:

- Starter: $EOP \rightarrow$, which delivers output to other physical operations;
- Linker: $\rightarrow EOP \rightarrow$, which receives and consumes input from other operations, and delivers output to other operations;
- Terminator: $\rightarrow EOP$, which receives input and consumes from other operations.

An operator that consists of more than one elementary operations is denoted by a set of *EOP*s,

i.e. $(EOP_1, EOP_2, \ldots, EOP_n)$, in which elementary operations would be executed in a sequential order.

Because intermediate results are passed from one operator to another one, hence an execution plan can be viewed as a pipeline of processes (i.e. operators) and intermediate or final results (i.e. data). Furthermore, some operators may be tuned to deliver (partial) intermediate results as soon as possible, and in such case an execution plan is called a pipelined plan.

4.3.2 Conceptual Design of TIP

For now let us discuss the design of top-k incorporated pipeline for query execution plan. Although to implement a practical TIP processor requires substantial engineering efforts, however, it is worthy of discussions on a conceptual and algorithmic level of TIP. Here we investigate two aspects: first, what physical operators are involved; and second, how top-k algorithms can be incorporated into an execution plan.

4.3.2.1 Physical Operators

Based on the previous discussions about *SEA* query (see Section 4.3.1), we propose three (classes of) physical operators: *index access* for selection, *probability estimator* for probability estimation, and *probability aggregator* for probability aggregation.

Index Access An index could be accessed in two modes. Operator *XS* stands for *index scan* and corresponds to sequential access mode, which could be deployed for retrieving tuples (or data items) in the posting lists of an index; in addition, if the posting lists of an index are sorted by certain weights, then *XS* is the operation should be called to perform sorted access to a lists of tuples. This is a starter operator in an execution plan, and it can be formulated as follow:

$$XS(index) \rightarrow \{(\tau, weight)\}$$

which means by given a certain index, *XS* retrieves a set of weighted tuples of τ.

In addition, operator *XRA* is proposed to allow *random access* functionality. Different from *XS*, *XRA* requires an accessing key in order to retrieve a weighted tuple from a given index. As a result, this is a linker operator in an execution plan, and the operator is given as follow:

$$key \rightarrow XRA(index) \rightarrow (\tau, weight)$$

Note that a given index must support random access so that *XRA* can be deployed.

Probability Estimator Probabilistic tuples are generated from unweighted tuples or tuples with different kind of weightings, while the process is performed by physical probability estimators denoted by *PE*, in which certain estimation model is applied. A general form of an operator-data pipeline of *PE* is given as follow:

$$\{(\tau_{\mathrm{X}}, weight)\} \rightarrow PE(model) \rightarrow \{(\tau_{\mathrm{X}}, score)\}$$

For instance, an *PE* could be an assembly of the physical implementations of Bayes and Project operators such as follow:

$$\{(\tau_{\mathrm{X}}, weight)\} \rightarrow (Bayes, Project) \rightarrow \{(\tau_{\mathrm{X}}, score)\}$$

In addition, an implementation of *PE* may be designed to take more than one inputs data streams such as follow:

$$(\{(\tau_{\mathrm{X_1}}, weight)\}, \ldots, \{(\tau_{\mathrm{X_n}}, weight)\}) \rightarrow PE(model) \rightarrow \{(\tau_{\mathrm{X'}}, score)\}$$

where in the above case, $X^{'} \subseteq X_1 \cup \ldots \cup X_n$, and an probability estimation process could incorporate top-k mechanisms to delivery probabilistic results as soon as possible.

Probability Aggregator Different from probability estimator which is used to assign initial probabilities to tuples, probability aggregator or *PA* removes duplicate tuples that contain the same (attribute) values and aggregate the probabilities of duplicate tuples in certain ways based on given probabilistic assumptions; while similarly to *PE*, an aggregation model (or method) should be given to a *PA*. The operator-data pipeline is demonstrated as follow:

$$(\{(\tau_{\mathrm{X_1}}, weight)\}, \ldots, \{(\tau_{\mathrm{X_n}}, weight)\}) \rightarrow PA(model) \rightarrow (\tau_{\mathrm{X'}}, score)$$

where $X^{'} \subseteq X_1 \cup \ldots \cup X_n$. For instance, a *PA* could be assembled by a combined operations of Join and Project such as follow:

$$(\{(\tau_{\mathrm{X_1}}, \ldots, weight)\}, \ldots, \{(\tau_{\mathrm{X_n}}, weight)\}) \rightarrow (Join, Project) \rightarrow (\tau_{\mathrm{X'}}, score)$$

Without loss of generality, one may expect the implementation of join are based on one of the existing algorithms such as nested-loop-join or hash join while applying suitable top-k methods

4.3.2.2 Incorporating Top-k Algorithms

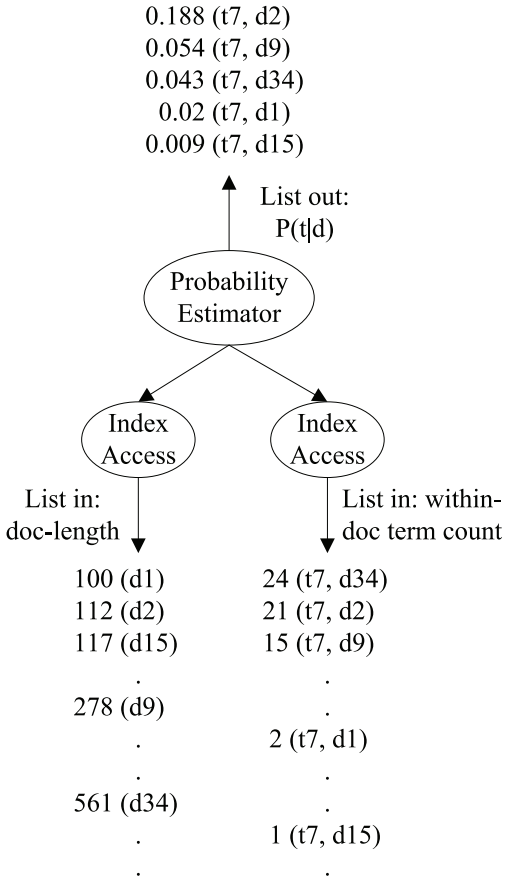
As we have mentioned (see Section 4.3.1), scoring functions or ranking models can be composed in a SEA query or a combination of multiple SEA queries in an IR+DB system. This is different from a conventional middleware architecture, where the entire middleware layer is indeed a complicated probability aggregator, while probability estimator could be either hard-coded as part of the probability aggregator or it is not necessary at all, because the probabilities of data items (e.g. tf and idf) have already been calculated materialised during indexing.

The basic idea to incorporate top-k algorithms into a pipelined procedures of query execution is to allow top-k mechanisms in SEA query blocks so that flexibility remains (because scoring functions can be still implemented in high-levelled abstraction languages such as PRA or probabilistic Datalog [Fuhr, 2000], also see Section 2.5.2) while improving the efficiency of an IR+DB system.

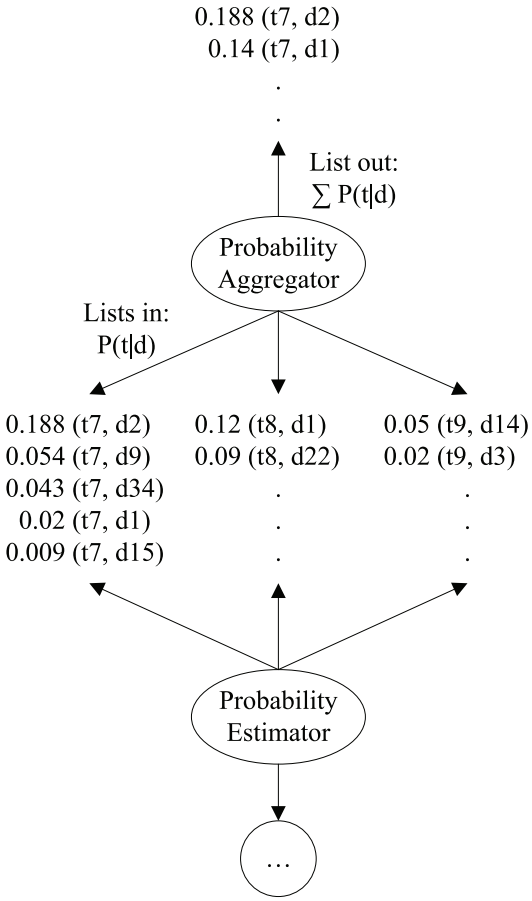
Let us use an example shown in Figure 4.4 to illustrate and explain the concept of top-k incorporated pipeline.

Figure 4.4a illustrates an execution sub-tree, in which a probability estimator (i.e. *PE*) takes two index accessing operators (i.e. *XA*) for retrievals from a posting list of document length and a posting list of within-document term count. While the output of *PE* are tuples with probabilistic weightings of $P(t|d)$, i.e. the within-document term frequency. Note that both posting lists store statistics based on raw term weight: the list of document length is sorted in ascending order, while the list of within-document term count is sorted in descending order. The *PE* estimates the probabilities of tuples by a given function, which is by Formula 2.28 (see page 55, also see Section 2.5.2.1) in this case.

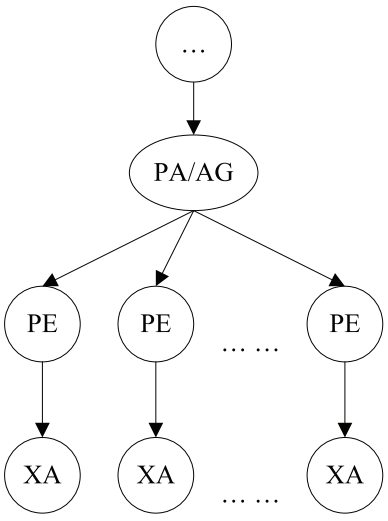
The main problem of top-k incorporated *PE* is that the accessed data have multiple dimensions while the tuple weights of a candidate must be able to be accessed by *PE* from all dimensions, otherwise the final score of a seen item cannot be computed. For instance, the weightings of document length and the weightings of within-document term count belong to two different dimensions, and the $P(t|d)$ score of an item can be yielded if and only if the tuple weights of the items are seen from both lists. As a result, if random accesses are available for both lists, then the original TA may be adapted to *PE*; however, if only sorted accesses are allowed, then an RNA



(a) Probability estimation with top-*k* adapted



(b) Probability aggregation with top-*k* adapted



(c) Pipelined execution plan for SEA query

Figure 4.4: Top-*k* incorporated pipeline

algorithm may still be adapted but the final scores of top candidates would be approximate.

Figure 4.4b illustrates a probability aggregator (i.e. *PA*) aggregates the intermediate results yielded by *PE* which are grouped by the attribute values of terms (i.e. keywords). In this case, a physical implementation of *PA* could perform the two steps:

1. For every terms (or keywords), request *PE* for a list of top candidates;
2. Aggregate the top candidates of every terms (or keywords) by the values of another attribute, for example, the document IDs (DocID).

Finally, figure 4.4c illustrates an example of how *XA*, *PE* and *PA* (or conventional aggregation *AG*) may form a SEA execution plan tree.

In a top-*k* incorporated pipeline, one crucial problem to be investigated is that when a pipelined top-*k* SEA query yields exact top-*k* results and when a query yields only approximate results. The same consideration have been addressed in [Ilyas et al., 2003] and [Theobald et al., 2005a]. In general, if a SEA query block is not embedded with another SEA query block, then performing top-*k* processing for the SEA query yields exact top-*k* results. This is because a *PE* delivers exact top-*k* results (note that its bound *XA* is supporting random access), and a onward *PA* implements similar algorithms as the Rank-Join [Ilyas et al., 2003] and the Rank-Aggregate [Li et al., 2006], which always computes scores/probabilities based on completely seen tuples' combinations. Therefore, top-*k* processing on non-nested SEA query blocks yields exact top-*k* results. On the other hand, if a SEA query is embedded with another SEA query, i.e. an outer SEA query uses the output of an inner SEA query, then performing top-*k* processing for the query yields only approximate results. This is because the probability estimator *PE* in the outer SEA cannot yield exact results based on a partial intermediate relation. A potential solution to improve retrieval precision for top-*k* processing on nested SEA queries could be increasing top-*k* budgets (see next Section 4.3.3) in the inner SEA query blocks.

4.3.3 An Investigation of Performances Tradeoff of NRA-Style Top-*k* Strategies

Here we introduce a concept of *ideal performances tradeoff measurement*, which helps to estimate the *tradeoff point* of efficiency versus effectiveness. In addition, we simulate NRA-style top-*k* strategies using PD and PRA, where we investigate the effect of different strategies which allot budget constraints (e.g. see [Shmueli-Scheuer et al., 2009]) for sorted accesses from posting lists.

4.3.3.1 Ideal Performances Tradeoff Measurement

While applying top-k mechanisms to speed-up query processing for IR applications, the tradeoff of efficiency versus effectiveness is an important measurement that is used to estimate and balance the losses of retrieval quality and the gains of response time. In previous work, for instance, [Hawking, 1998], graphs of time and effectiveness are plotted against optimization methods (e.g. top-k methods in used), and then by inspecting the graphs, an optimum tradeoff point can be selected. The method was used for measuring document retrieval, while similar studies had also been carried out for XML retrieval. For example, [Fuhr and Gövert, 2006] investigated the tradeoff for interactive XML retrieval while focusing on effectiveness, where retrieval qualities for those processes with and without top-k mechanisms were compared against the retrieval response time (measured or predicted).

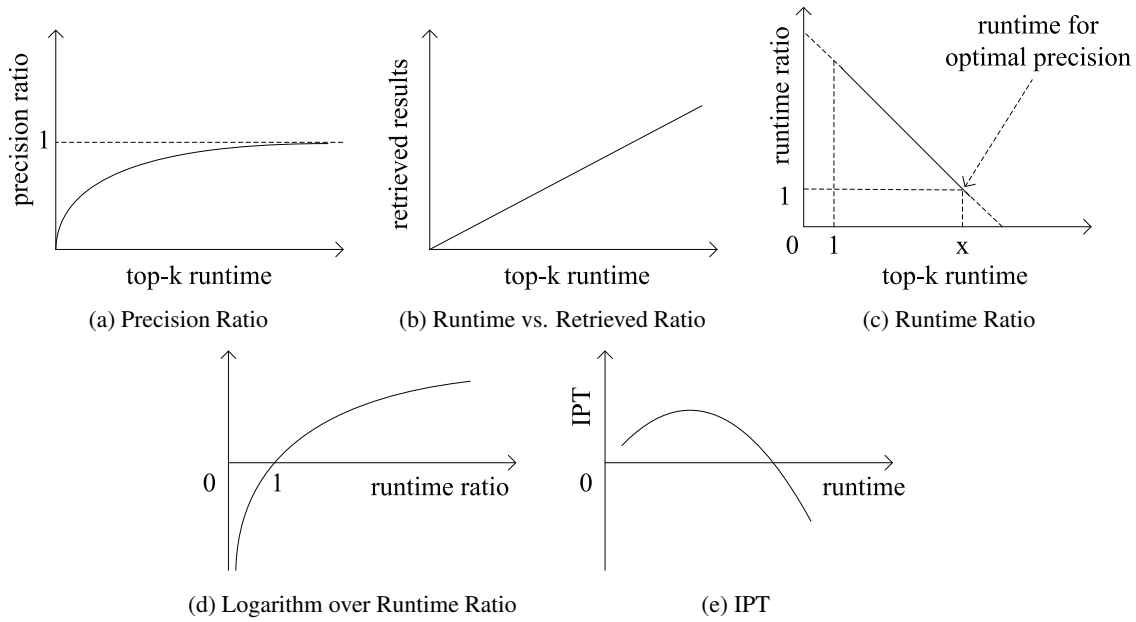
Notice that the tradeoff point of efficiency against effectiveness was estimated by "*inspection*" in the previous studies, where effectiveness was assessed and then plotted on certain time span. On the other hand, the studies about tradeoff in the previous work were based on empirical experiments without formal definitions, but it would be highly intriguing to have a comprehensive formulation for performance tradeoff.

Though we realise the difficulties of formally defining the efficiency and effectiveness tradeoff, but it is still reasonable to discuss a theoretically comprehensive formulation. Here, we propose a formulation defining the tradeoff of precision versus retrieval time for a given query, which is given as follow:

$$IPT(q) := \frac{\text{Precision}_{top}(q,k)}{\text{Precision}_{all}(q)} \cdot \log\left(\frac{\text{Time}_{all}(q)}{\text{Time}_{top}(q,k)}\right) \tag{4.1}$$

In Formula 4.1, $IPT(q)$ stands for *ideal performances tradeoff* of a query q, while the subscriptions top and all indicates retrieval modes; in particular, a parameter k represents the limit of the number of results to be retrieved in a top-k mode. The right hand side of the formula contains two ratios. One is the ratio of precisions $\frac{\text{Precision}_{top(q,k)}}{\text{Precision}_{all}(q)}$, where $\text{Precision}_{all}(q)$ is the optimal precision that can be achieved and $\text{Precision}_{top}(q,k)$ is the precision achieved in top-k processing, hence the ratio indicates the percentage of optimal precision. Similar measure has also been used in [Shmueli-Scheuer et al., 2009]. The other one is the ratio of runtime $\frac{\text{Time}_{all}(q)}{\text{Time}_{top}(q,k)}$, where $\text{Time}_{all}(q)$ is the runtime to achieve the optimal precision and $\text{Time}_{top}(q,k)$ is the runtime of top-k processing, thus the ratio represents the number of times that processing efficiency is speeded

up. In other words, the precision ratio indicates effectiveness loss and the runtime ratio indicates efficiency gains while running queries in top-k processing mode.



(a) Precision Ratio (b) Runtime vs. Retrieved Ratio (c) Runtime Ratio

(d) Logarithm over Runtime Ratio (e) IPT

Figure 4.5: Ratio and Ideal Performances Tradeoff

Moreover, let us discuss how the precision ratio and the runtime ratio should be interleaved in order to represent the performance tradeoff IPT. Let precision to be defined as the ratio of precision $:= \frac{|R_a|}{|A|}$, where $|A|$ stands for the number of retrieved results in the answer set, while $|R_a|$ means the number of relevant results in the answer set. Assuming a monitoring program can compute the precision repeatedly at a constant interval during top-k retrieval. Let precision(t) be the real-time precision at time t, and let $\delta_p = \text{precision}(t_i) - \text{precision}(t_j)$, where $i - j = 1$ be the increment of a real-time precision in one interval. Empirical experiences show that precision tends to increase relatively fast during early runtime and δ_p drops quickly towards zero until precision$t$ reaches the optimal precision. Therefore, the precision ratio $\frac{\text{Precision}_{top(q,k)}}{\text{Precision}_{all(q)}}$ can be illustrated as Figure 4.5a. On the other hand, assuming the retrieval engine supports pipelined delivery of results, then theoretically the number of results is in proportional to runtime, such as it is shown in Figure 4.5b. In order words, the runtime ratio $\frac{\text{Time}_{all(q)}}{\text{Time}_{top(q,k)}}$ can be illustrated as Figure 4.5c. However, the real-time precision does not approach the optimal precision in a linear manner but in a logarithmic kind, thus such feature should be reflected from the runtime, so that applying logarithm over the runtime ratio would be reasonable, which is illustrated in Figure 4.5d, where the logarithm of runtime ratio decreases faster and faster when the ratio

get closer to 1. Finally, the IPT could be obtained by multiplying the precision ratio and the logarithmic runtime ratio. Note that while runtime increases, the precision ratio is a monotonic increasing function with saturation at an optimal precision, and the logarithmic runtime ratio is a monotonic decreasing function. As one can imagine, the IPT becomes negative when the top-k runtime exceeds the runtime needed for achieving the optimal precision. To demonstrate, the IPT could look like Figure 4.5e. In other words, the IPT illustrates the balance of precision gains and runtime expenditures.

4.3.3.2 Modelling NRA-Style Top-*k* in Declarative Languages

In the previous discusses we have introduced modelling retrieval strategies in declarative languages such as probabilistic Datalog (PD) and probabilistic relational algebra (PRA) (see e.g. Chapter 2, especially, Section 2.5.2). Here we discuss how to simulate NRA-style top-k queries that composed in PD and PRA, and we demonstrate the modelling for *tf-idf* model (see Section 2.2.2.1, also see Formula 2.1 in page 32).

```
wqterm(T,Q):-
      qterm(T,Q) & idf(T).
retrieve(D):-
      wqterm(T,Q) & tf(T,D).
```

(a) PD rules

```
wqterm=Project[$Term,$Qid](
  Join[$Term=$Term](qterm,idf));
retrieve=Project SUM[$DocId](
  Join[$Term=$Term](wqterm,tf));
```

(b) PRA expressions

Figure 4.6: PD and PRA for *tf-idf* model

A conventional modelling for *tf-idf* are illustrate in Figure 4.6b, where Figure 4.6a shows a modelling in PD, while Figure 4.6b shows an equivalent translation (of PD) in PRA. For the modelling in PRA, a join operation in the first expression assigns an $idf(t)$ score to each query term, and a projection is performed to yield a list of weighted query terms (i.e. *wqterm*). Assuming a probability estimation for $tf(t,d)$ has been done in advance and the intermediate results are stored in a view (intensional relation) called tf, so that in the second PRA expression, where a join for *wqterm* and tf not only combines the tuples of the two relations but also implies a multiplication of $idf(t) \cdot tf(t,d)$, and then finally a projection is used to aggregate *tf-idf* scores of different query terms.

To simulate NRA-style top-k strategy in PRA, an example is demonstrated in Figure 4.7. As they are shown, budgets are specified in selection statements for query terms, which indicates the maximum number of tuples (data items) is allowed to be retrieved from the associated posting lists of query terms. Moreover, the union statements (i.e. Unite) generate an intermediate relation

```
term0=Select[$Term='hybrid'](tf):1000;
term1=Select[$Term='car'](tf):1500;
term2=Select[$Term='fuel'](tf):1200;
topTf=term0;
topTf=Unite ALL(topTf,term1);
topTf=Unite ALL(topTf,term2);
retrieve=Project SUM[$DocId](Join[$Term=$Term](wqterm,topTf));
```

Figure 4.7: Simulating NRA-style top-k strategy in PRA

topTf, which contains the top candidates of relevant items of all query terms. And then finally, only the top items are joined with *wqterm* which carrying the $idf(t)$ scores, while an aggregation for final scores will be performed as usual.

4.3.3.3 Allotting Strategies for Budgets

In previous subsection (see Section 4.3.2) we introduced SEA query and discussed the concept of incorporating top-k mechanisms into this type of queries. In practice, a strategy developer (who model scoring strategies in PD or PRA) wants to deploy top-k functionality in s/he's queries, s/he only needs to specify (explicitly or implicitly) a budget constraint for the final results; on the other hand, to process a pipelined and top-k incorporated execution plan, the internal engine should determine the local budget for sub-query plans.

Now let n_{budget} be a budget constraint for the final result, and let N_{qterm} be the number query terms; then a naive strategy is to deploy directly the budget to every sub-queries, for example, to the *PE* operations; hence, the actual total number of tuples to be retrieved is denoted as N_{budget}, which is called global budget, and could be estimated as follow:

$$N_{budget} = n_{budget} \cdot N_{qterm} \tag{4.2}$$

Note that N_{budget} is also the maximum number of tuples that need to be retrieved from storage, while the actual number of tuples to be fetched could be less than that.

In fact, different strategies can be applied to allot local budgets of sub-queries, here we introduce three allotment strategies based on either intuition or heuristic.

- *Uniform allotment of budgets*: intuitively, a global budget N_{budget} is to be allotted evenly among every query terms, where the allotment is given by Formula 4.3, in which $top(q)$ is the local budget of query term τ_q. This is equivalent to directly deploying the budget for final result (i.e. n_{budget}) to individual sub-queries.

$$top(\tau_q) = \frac{N_{budget}}{N_{qterm}} = n_{budget} \tag{4.3}$$

- *Static IDF-based allotment of budgets*: heuristically, a global budget N_{budget} can be allotted among query terms based on their *idf* values, where the allotment is formulated by Formula 4.4. As it is shown, the allotment for a query term τ_q is based on the proportion of its *idf* value against the sum of the *idf* values of all query terms.

$$top(\tau_q) = N_{budget} \cdot \frac{-\log df(\tau_q)}{\sum_{i=1}^{N_{qterm}} (-\log df(\tau_i))} \tag{4.4}$$

- *Dynamic IDF-based allotment of budgets*: also by heuristic, a dynamic version of the IDF-based allotment for local budgets not only considers the *idf* values of individual query terms, but also uses the up-to-date global budgets. In this strategy, assuming a local budget $top(\tau_q)$ is assigned to query term τ_q, but the actual retrieved number of tuples is $fetched(\tau_q)$ where $fetched(\tau_q) \leq top(\tau_q)$, so for the next query term, the global budget would reduced by removing the number of retrieved tuples m, i.e. $\sum_{i=1}^{m} fetched(\tau_i)$. The formulation of the allotment is given by Formula 4.5.

$$top(\tau_q) = \left(N_{budget} - \sum_{j=1}^{m} fetched(\tau_j) \right) \cdot \frac{-\log df(\tau_q)}{\sum_{i=m+1}^{N_{qterm}} (-\log df(\tau_i))} \tag{4.5}$$

Furthermore, we can view each selection statement as a data source (i.e. the documents retrieved for a term); and then, the top-k processing strategy can be related to database selection. e.g. see [Fuhr, 1999a, Fuhr, 1999b]. The idea of database selection is to consider more documents from data sources that are more likely to deliver relevant documents, since such strategies would minimise the costs to find and read relevant documents. On the other hand, a reasonable budgetary allotment strategy minimises the costs to achieve a retrieval quality close to what full retrieval would deliver. The uniform budgetary allotment (see Formula 4.3) reflects a baseline where no prior knowledge is available, while the IDF-based strategies (see Formula 4.4 and 4.5) exploit the global term statistics.

Finally, the proposed allotment strategies share a similar goal with those budget-aware scheduling strategies introduced in [Shmueli-Scheuer et al., 2009], while our work are different from theirs in two main aspects. Firstly, the implementation level of

[Shmueli-Scheuer et al., 2009] aims at middleware systems, whereas ours aims at query engine supporting pipelining. Secondly, also as a result of the first aspect, the scheduling methods in [Shmueli-Scheuer et al., 2009] consider both sorted access and random access, while random access is unsuitable for pipelined operators, hence the proposed allotment strategies consider sorted access only.

4.4 Experiments and Results

In this section, we present the experiments, which were designed for the following purposes:

- To demonstrate the tradeoff of efficiency versus effectiveness while applying NRA-style top-k mechanisms in an IR+DB system;
- To demonstrate using ideal performances tradeoff (IPT) measurement for estimating the tradeoff points of top-k processing;
- To evaluate the effectiveness of the introduced allotting strategies for internal budget assignments in pipelined query execution processes.

Next, we address the experimental specifications and results.

4.4.1 Specifications and Setup

Systems HySpirit [Fuhr and Roelleke, 1998, Fuhr et al., 1998, Rölleke et al., 2001] was used as the experimental platform, which was hosted on a Linux server with the following specifications: Fedora 8 Linux 64 bits operating system, eight AMD Opteron[2](TM) Dual-Core CPUs at frequency 3.00 GHz, 32 GB of RAM. Note that HySpirit was neither optimized towards 64 bits operating system nor parallel computing with multiple CPUs machine.

Test Collection TREC-3 were used as the testing collection and our experiments were run with 50 queries, i.e. topics 151-200, provided by TREC[3]. The original documents in TREC-3 were parsed and stored in a DB-style relational table named *TermDoc* with data schema `(Term, DocID)`, while the data size of relation *TermDoc* is 5.0 GB. In addition, another table named *TermDocTF* was produced that materialises the within-document term frequency weightings, i.e. $P_C(t|d)$, of tuples, and the data of table *TermDocTF* is 3.88 GB. Examples of the two tables are

[2]Opteron is a trademark of Advanced Micro Devices, Inc..
[3]http://trec.nist.gov/

illustrated in Figure 4.8 which shows the snapshots of the storage. In addition, indexes were built on attribute `Term`.

1(torch,“AP890101-0001”)
1(generation,“AP890101-0001”)
1(filmmakers,“AP890101-0001”)
1(grew,“AP890101-0001”)
1(platoon,“AP890101-0001”)
1(running,“AP890101-0001”)
1(empty,“AP890101-0001”)
1(mississippi,“AP890101-0001”)
1(burning,“AP890101-0001”)

(a) *TermDoc*

0.00153374(passed,“AP890101-0001”)
0.00920245(platoon,“AP890101-0001”)
0.00613497(running,“AP890101-0001”)
0.00460123(burning,“AP890101-0001”)
0.00306748(brought,“AP890101-0001”)
0.00153374(passage,“AP890101-0001”)
0.00306748(believe,“AP890101-0001”)
0.01380370(vietnam,“AP890101-0001”)
0.00613497(people,“AP890101-0001”)

(b) *TermDocTF*

Figure 4.8: Snapshot of HySpirit storage for TREC-3 collection

Setup Experiments were set to run in batch mode for the 50 queries with *tf-idf* strategy, which is modelled in PRA as it was discussed in Section 4.3.3.2. The baseline is a full retrieval run without deploying top-k mechanisms, in which the retrieval time of the 50 queries were recorded, and the retrieval results were assessed by TREC evaluation tool. In addition, Table 4.1 shows the results of the baseline runs, including average retrieval time, mean average precision (MAP), and precision at ten (P@10).

t (ms)	*MAP*	*P@10*
7003	0.1989	0.456

Table 4.1: Full run retrieval time vs. precision (baseline)

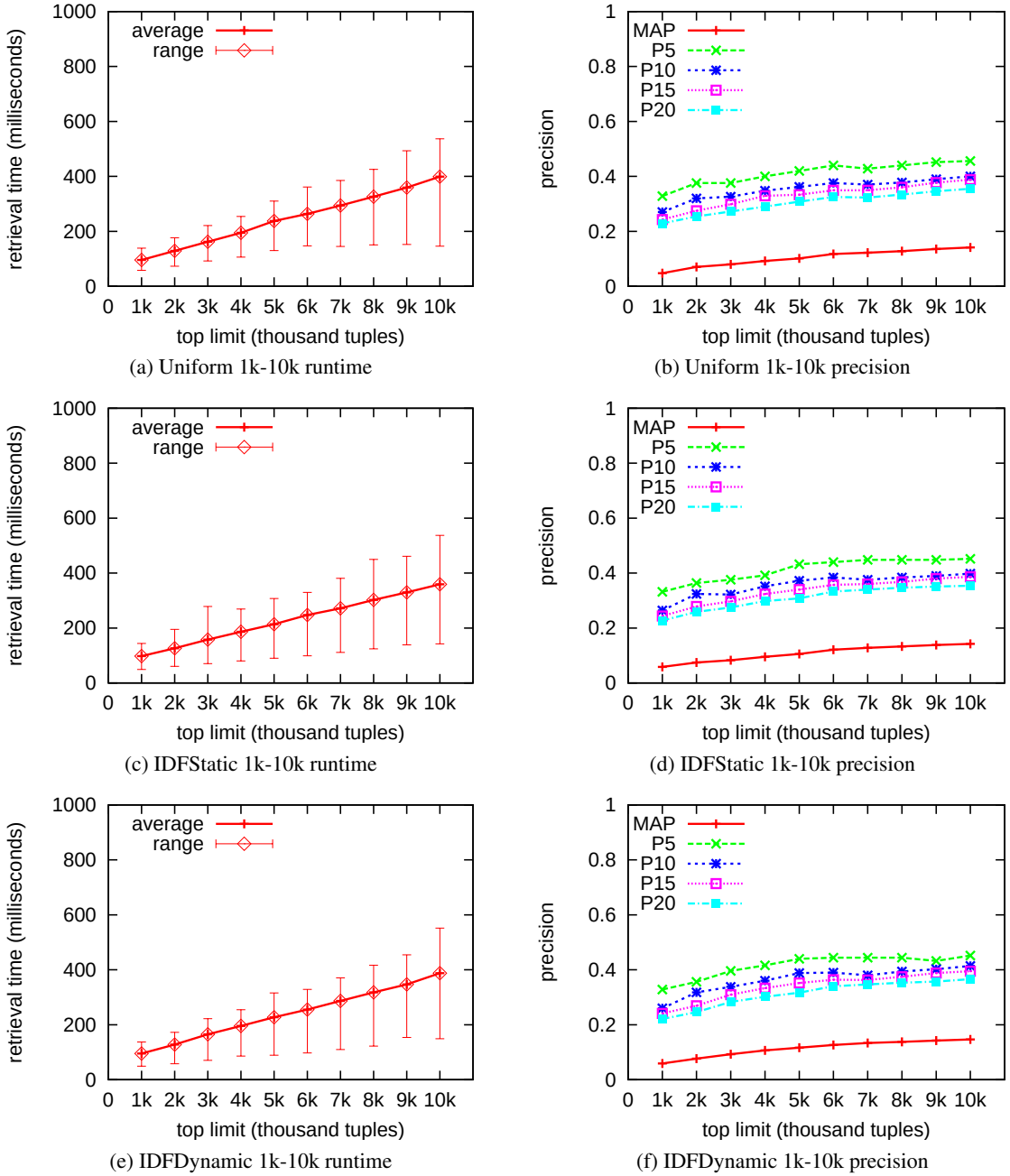
4.4.2 Methodology

For comparison, we investigate query processing in a TIP-style[4] manner retrieval with HySpirit, in which the simulating method is as described in Section 4.3.3.2, while the global budget allotting strategies discussed in Section 4.3.3.3 would be investigated. The ratio of comparing values recorded in a top-k run and a full retrieval run would be computed as $r_{top} := \frac{v_{top}}{v_{all}}$, so that the gain or loss (GOL_{top}) of a top-k measurement could be obtained by $GOL_{top} := 1 - r_{top}$.

Based on the settings of HySpirit, the posting lists of terms (keywords) were ordered by document IDs (but not term-based statistics), and the experiments were set off in two groups. In

[4]Precisely, we simulate the conceptual design (see Section 4.3.2) of TIP with HySpirit.

the first group, the global budgets were scaled from 1 000 to 10 000 tuples with one thousand tuples per increment, where runtime (every queries and the average), MAP and P@10 were recorded; meanwhile, the ideal performances tradeoff (IPT) (see Section 4.3.3.1) of each run would be computed as well. In the second group, the global budgets were scaled from 10 000 to 50 000 tuples with ten thousand tuples per increment while the other settings remained the same.



(a) Uniform 1k-10k runtime

(b) Uniform 1k-10k precision

(c) IDFStatic 1k-10k runtime

(d) IDFStatic 1k-10k precision

(e) IDFDynamic 1k-10k runtime

(f) IDFDynamic 1k-10k precision

Figure 4.9: Top-*k* retrieval time vs. precision (global budgets 1k - 10k)

4.4.3 Results

Now we present the experimental results. For the first experimental group, i.e. global budgets scaled from 1 000 to 10 000 tuples, the results of the three strategies are plotted in Figure 4.9 while the comparisons of average performances are given in Table 4.2.

Budget	*Uniform*			*IDFStatic*			*IDFDynamic—*		
	t (ms)	*MAP*	*P@10*	*t (ms)*	*MAP*	*P@10*	*t (ms)*	*MAP*	*P@10*
1k	96	0.0475	0.27	98	0.0583	0.264	95	0.059	0.26
2k	129	0.0704	0.32	126	0.0743	0.324	128	0.0767	0.318
3k	162	0.0799	0.326	158	0.0826	0.322	165	0.0927	0.338
4k	194	0.0919	0.348	186	0.095	0.352	196	0.1065	0.36
5k	238	0.1016	0.362	214	0.1056	0.372	227	0.1167	0.388
6k	263	0.1174	0.376	247	0.1209	0.384	255	0.1263	0.39
7k	294	0.122	0.37	272	0.1278	0.376	287	0.1337	0.38
8k	326	0.128	0.378	302	0.1329	0.384	318	0.1379	0.394
9k	359	0.1356	0.39	330	0.1382	0.39	347	0.1424	0.402
10k	399	**0.1411**	**0.4**	359	**0.1421**	**0.398**	388	**0.1466**	**0.414**

Table 4.2: Retrieval time vs. precision with global budgets 1k - 10k

In a runtime plot corresponding to efficiency, for instance, Figure 4.9a, the y-axis is retrieval time in milliseconds, while the x-axis is budget scale; the curve named "average" stands for the average runtime of 50 testing queries, while the error bar named "range" indicates the range between minimum and maximum runtime of the queries. In addition, there is a precision plot for each strategy as well, which represents the effectiveness against global budget scales. For instance, Figure 4.9b shows the precision of MAP, P@5, P@10, P@15 and P@20 for different global budgets. Numerical comparisons of average precision are shown in Table 4.2, though the findings are not very significant: it shows that with average runtime grows nearly linearly, the MAPs of the three strategies increase considerably from 1k to 2k and then gradually increase after that, and the P@10s appear to increase smoothly for all strategies. While comparing to the baseline run, the gain or loss (see Section 4.4.1) of the three strategies of budgets at 10k tuples are estimated using ratio as $\frac{\text{topk\_run\_value}}{\text{full\_run\_value}}$:

- Uniform at 10k: runtime ratio: 0.057 (i.e. 94.3% gain), MAP ratio: 0.709 (i.e. 29.1% loss), P@10 ratio: 0.877 (i.e. 12.3% loss)
- IDF static at 10k: runtime ratio: 0.051 (i.e. 94.9% gain), MAP ratio: 0.714 (i.e. 28.6% loss), P@10 ratio: 0.873 (i.e. 12.7% loss)
- IDF dynamic at 10k: runtime ratio: 0.055 (i.e. 94.5% gain), MAP ratio: 0.737 (i.e. 26.3% loss), P@10 ratio: 0.908 (i.e. 9.2% loss)

Moreover, the IPTs of the strategies were calculated and given in Table 4.3. As they were shown, the general trends of the IPTs for all strategies appear to be growing, where higher values of IPTs achieved while more budgets were allowed; though there is an exception for static IDF strategies, but it does not contradict to the growing trends for later budget examination points. In fact, all strategies appear to achieve a relatively high IPTs at 5k or 6k check points while compared to their respective maximum IPTs in the experimental group.

Budget	*Uniform*		*IDFStatic*		*IDFDynamic*	
	IPT_MAP	*IPT_P@10*	*IPT_MAP*	*IPT_P@10*	*IPT_MAP*	*IPT_P@10*
1k	2.6742	6.6304	3.2761	6.4709	3.3232	6.3878
2k	3.8592	7.6514	4.0803	7.761	4.2082	7.6102
3k	4.2876	7.6305	4.4431	7.555	4.9663	7.8984
4k	4.8477	8.007	5.0321	8.1327	5.6144	8.278
5k	5.257	8.17	5.5195	8.4811	6.0642	8.7943
6k	6.014	8.4014	6.2312	**8.6328**	6.4897	8.7409
7k	6.1814	8.177	6.5265	8.3755	6.7918	8.4199
8k	6.4196	8.2691	6.7159	8.464	6.933	8.6402
9k	6.7346	8.4487	6.9225	8.521	7.0971	8.7392
10k	**6.933**	**8.5728**	**7.0573**	8.6217	**7.2246**	**8.8992**

Table 4.3: Ideal performances tradeoff with global budgets 1k - 10k

For the second experimental group, i.e. global budgets scaled from 10 000 to 50 000 tuples, the results are plotted in Figure 4.10 and the average performances are given in Table 4.4. From the plots in Figure 4.10 we can see that the runtime increase almost linearly while budgets increased, but precision values have been already very close to their highest achievable values at 20k and hardly increase any more (some even appear to be decreasing after 20k). Though most strategies appear to achieve the highest precision values at 50k (with one exception only), but when we compare the gain and loss values at 20k and 50k to the baseline to illustrate the differences. First, the 20k checkpoints are obtained as follows:

- Uniform at 20k: runtime ratio: 0.096 (i.e. 90.4% gain), MAP ratio: 0.859 (i.e. 14.1% loss), P@10 ratio: 1.0 (i.e. 0% loss)
- IDF static at 20k: runtime ratio: 0.085 (i.e. 91.5% gain), MAP ratio: 0.881 (i.e. 11.9% loss), P@10 ratio: 0.987 (i.e. 1.3% loss)
- IDF dynamic at 20k: runtime ratio: 0.09 (i.e. 91% gain), MAP ratio: 0.889 (i.e. 11.1% loss), P@10 ratio: 0.982 (i.e. 1.8% loss)

And then, the 50k checkpoints are given as follows:

- Uniform at 50k: runtime ratio: 0.193 (i.e. 80.7% gain), MAP ratio: 0.914 (i.e. 8.6% loss), P@10 ratio: 0.996 (i.e. 0.4% loss)

- IDF static at 50k: runtime ratio: 0.17 (i.e. 83% gain), MAP ratio: 0.932 (i.e. 6.8% loss), P@10 ratio: 1.018 (i.e. 101.8% gain)
- IDF dynamic at 50k: runtime ratio: 0.2 (i.e. 80% gain), MAP ratio: 0.932 (i.e. 6.8% loss), P@10 ratio: 1.004 (i.e. 100.4% gain)

Budget	*Uniform*			*IDFStatic*			*IDFDynamic*		
	t (ms)	*MAP*	*P@10*	*t (ms)*	*MAP*	*P@10*	*t (ms)*	*MAP*	*P@10*
10k	399	0.1411	0.4	359	0.1421	0.398	388	0.1466	0.414
20k	673	0.1709	0.456	597	0.1752	0.45	632	0.1768	0.448
30k	911	0.1789	0.456	769	0.1811	0.46	879	0.1810	0.45
40k	1137	0.1814	**0.464**	945	0.183	0.456	1169	0.1847	0.456
50k	1350	**0.1818**	0.454	1189	**0.1854**	**0.464**	1400	**0.1854**	**0.458**

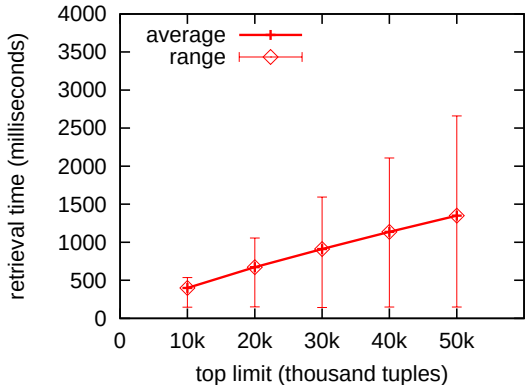
Table 4.4: Retrieval time vs. precision with global budgets 10k - 50k

Budget	*Uniform*		*IDFStatic*		*IDFDynamic*	
	IPT_MAP	*IPT_P@10*	*IPT_MAP*	*IPT_P@10*	*IPT_MAP*	*IPT_P@10*
10k	6.933	8.5728	7.0573	8.6217	7.2246	8.8992
20k	7.9485	**9.2507**	8.2541	**9.2473**	**8.2778**	**9.149**
30k	**8.0475**	8.9472	**8.3012**	9.1971	8.1742	8.8644
40k	7.9578	8.8786	8.198	8.9105	8.0771	8.6981
50k	7.8187	8.5166	8.0922	8.8337	7.9392	8.5546

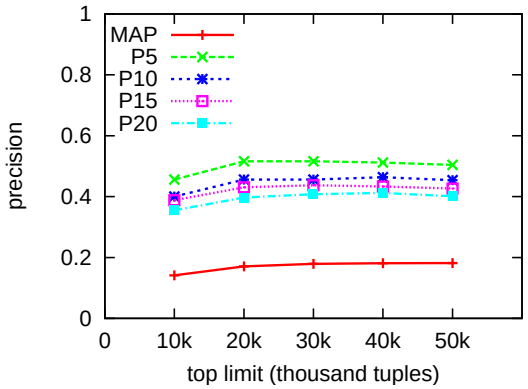
Table 4.5: Ideal performance tradeoff of with global budgets 10k - 50k

Here we can observe some interesting phenomena. First, the retrieval times at 50k are about twice as much as the runtime at 20k, but the improvements of precision are relatively small while more tuples are retrieved. Second, some strategies even achieve as good results as the full retrieval run baseline for precision P@10 while global budget at 20k. Third, some strategies achieve a little better results than the baseline for precision P@10 while global budget at 50k. In other words, these observations suggest that top-k processing may not always lead to losses in terms of retrieval quality.

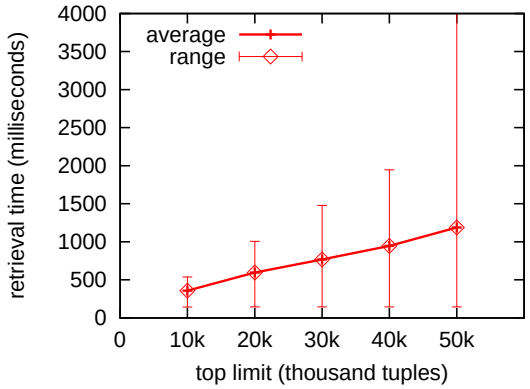
While considering the possibly best tradeoff points, we compute the IPTs for the comparing strategies at the checkpoints, which are given in Table 4.5. As they were highlighted, most strategies reach the best tradeoff point for P@10 at 20k checkpoint; Though only IDF dynamic reaches the best tradeoff point for MAP at 20k, but the IPTs of the other two at 20k are actually not far from the turning points (which appear at 30k) at all.



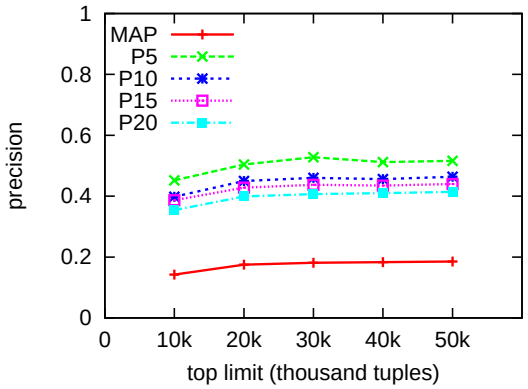
(a) Uniform 10k-50k runtime



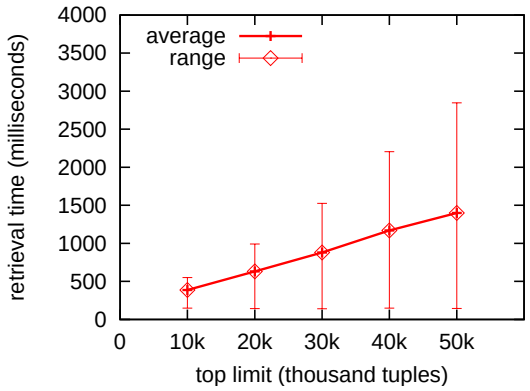
(b) Uniform 10k-50k precision



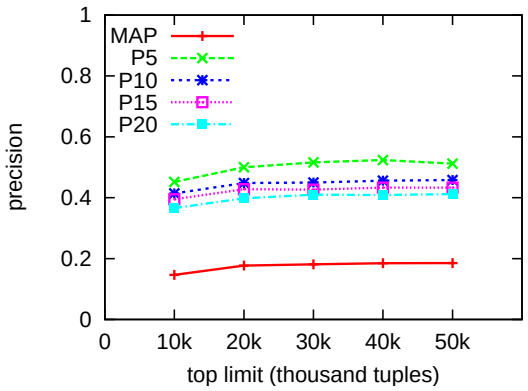
(c) IDFStatic 10k-50k runtime



(d) IDFStatic 10k-50k precision



(e) IDFDynamic 10k-50k runtime



(f) IDFDynamic 10k-50k precision

Figure 4.10: Top-*k* retrieval time vs. precision (global budgets 10k - 50k)

4.5 Summary

To summarise this chapter, we have discussed and investigated the usage of top-*k* processing in an IR+DB environment. The main contributions are three folds. Firstly, we introduced (conceptually) a physical query execution mechanism named top-*k* incorporated pipeline (TIP), which aims to employ and adapt popular top-*k* algorithms such as TA families into a generic IR+DB processing engine, this is essentially different from the previous studies, which applied TA or its variants in a middleware architecture of database applications. Secondly, we attempted defining mathematically a measurement for performances tradeoff with respect to efficiency versus effectiveness, where we proposed ideal performances tradeoff (IPT) which can be used to estimate the tradeoff points of top-*k* processing. Third, to perform (or simulate) query executions in a TIP-style manner, we discussed three allotting strategies for global budgets; in addition, we investigated the allotting strategies with a TREC collection in experiments, in which we also applied IPT measurement to estimate the tradeoff points of different scales of global budgets.

The intent of this study is not to propose brand new top-*k* algorithms, but to adapt and incorporate originally external top-*k* methods into a generic query execution engine of IR+DB systems. In other words, we developed an algebraic version of top-*k* processing mechanisms while compared to conventional IR search engines, hence certain flexibility would be allowed in IR applications while efficiency requirements can still be met. On the other hand, similar demands of developing top-*k* incorporated query execution engine had already been called out by researchers from DB community (e.g. see [Chaudhuri et al., 2005]), so that our work can be viewed as one step forward comparing to previous studies focused on middleware architectures.

Chapter 5

RIX: Indexing with Relational Inverted Index

5.1 Introduction

In this chapter, we will discuss an indexing technique for IR+DB systems. Various accessing methods have been developed to support efficient search and retrieval on IR systems and databases. In IR, indexing methods over text collections include *suffix arrays* [Manber and Myers, 1990], *inverted files* or *inverted indexes* [Moffat and Zobel, 1996, Witten et al., 1994], and *signature files* [Faloutsos and Christodoulakis, 1984]. In particular, inverted index variants have been widely used as a default structure in modern IR applications. For example, Web IR [Brin and Page, 1998, Arasu et al., 2001], XML retrieval [Weigel et al., 2004], and Book Search [Wu et al., 2008b]. Furthermore, commercial search engine had achieved subsecond query response times by using an inverted index. In DB, indexes are chosen between *tuple-based index* or *TID-lists index* (or tuple-Id index) and *Bitmap* index. DB indexes are incorporated into query optimization to support efficient query evaluation. For example, for typical Select-Project-Join (SPJ) queries, aggregations (with GROUP BY clause), and sorting (with ORDER BY clause).

5.1.1 Motivation

Our intention on developing an indexing method is to support efficient probability estimation and context augmentation on IR+DB systems. Because both operations are expensive in terms of I/O intensive and compute intensive.

On estimating probabilities To estimate probability for a raw table is to assign initial probabilities (see Section 2.5.2.1 for different types of initial probability) to its tuples. In short, an estimation without utilising index involves table scanning and on-the-fly value aggregation: an entire table scan is inevitable in order to determine the size of event space(s), where grouping and counting are performed to separate sub event spaces (e.g. documents or segments) as well as to obtain the occurrences of distinct events. For very large data set, temporary files might be needed to store intermediate results while main memory is not big enough to hold all data.

In IR, grouping and counting are saved by inverted indexes, because they are part of index constructing procedure and statistic values are materialised in the index. Therefore, the cost to obtain initial probabilities is mainly bounded by I/O. On the other hand in databases, one can exploit TID index to speed up aggregations by explicitly including indexed (key) attributes in a GROUP BY clause in SQL. In this case, intensive on-the-fly grouping is replaced by relatively less expensive I/O access, but in-memory counting has to be performed and its computation cost is still considerable.

On context augmentation Context augmentation [Lalmas and Roelleke, 2002, Roelleke, 1999] is an IR concept that is applicable to many IR tasks. For example, for link-based retrieval [Craswell et al., 2001] or XML retrieval [Fuhr and Großjohann, 2004]. Its underlying principle is term weight propagation toward associated items. In general, augmentation consists of a combination of comparisons and aggregations, which could be also implemented by running SQL queries with Select-Project-Join or Select-Project-Unite in databases.

It is known in both communities that such augmentation is extremely expensive to compute on-the-fly, therefore either IR or DB came out with their own but in principle similar solutions. IR would build multiple inverted indexes in which each index represents one context. For example to apply tf-boosting on the Web search [Craswell et al., 2001], building one index for web pages in the form of $\langle Term, Doc \rangle$ and another index for anchor text in the form of $\langle Term, RefDoc \rangle$, thus while propagating term weight from anchor text to referenced web pages (so-call tf-boosting) becomes a task of fetching postings from two inverted indexes and summing up term weights of matching documents. On the other hand, database introduced materialised view to store intermediate results of SQL queries, so that term weights under different contexts are instantiated in materialised views or auxiliary tables, e.g. see [Theobald et al., 2005b]. As a result, SQL queries for augmentation could be processed quite efficiently.

In general, TID index sustains maximum flexibility and provides feasible support to relational operations, while considering the total costs (I/O and compute) for running IR ranking models, inverted index is superior than TID index. Therefore, we intend to adapt and integrate inverted index and TID index for IR+DB systems to allow efficient aggregation for ranking and retain flexibility for relational manipulation.

5.1.2 Inverted Indexes

Traditionally in IR, an inverted index was usually referred to inverted *document* index. In general, documents contain terms (words), so that given a document we know what terms occur in documents. On the contrary, there is a pair-wise relationship $\langle term, document \rangle$ between terms and documents, so that if given a term then the term's occurring documents could be retrieved, and this is the basic principle of inverted document index. If the concept of document is to be extended to bag-of-words, then inverted index is applicable to all kinds of text retrieval with or without considering document structure. For example, for passage retrieval or for XML retrieval.

Basically, there are two main components in an inverted index: a keys (terms) lookup facility, which could be implemented by B-tree or hash table; and posting data (or just posting), which are inverted lists associated to indexed keys. An inverted list is a series of $\langle document, value \rangle$ pairs (*document-level*), where the value could be some statistic of the term against the document, for instance, within-document term frequency (*tf*). While building a full-text index for documents (*word-level*), the locations of a term in an document would be included as well, thus an inverted unit becomes $\langle document, value, locations \rangle$, where locations is a list of offsets of words[1] $\langle l_1, l_2, \ldots, l_{tf} \rangle$.

For illustrating, we demonstrate with the toy magazine corpus in Section 2.2.1 (see Figure 2.2). The inverted list for term "hybrid" in the trimmed corpus (see Figure 2.3) without including locations look like:

$$\langle doc1, 1 \rangle, \langle doc2, 3 \rangle$$

While including locations, then it looks like:

[1] Using word-offset but not character-offset is because in this way proximity between terms could be computed, thus phrase could be found out.

$$\langle doc1, 1, 4\rangle, \langle doc2, 3, 42, 43, 50\rangle$$

While including location information into inverted index, index size increased dramatically. Therefore, [Witten et al., 1994, Moffat and Zobel, 1996] discussed compression coding method using *d-gaps*. In this method, instead of including the absolute position in inverted lists, it records the incremental *gap* between locations, i.e. $\langle l_1, \delta_2, \ldots, \delta_{tf}\rangle$. For instance, the inverted list for "hybrid" using *d-gaps* coding becomes:

$$\langle doc1, 1, 4\rangle, \langle doc2, 3, 42, 1, 7\rangle$$

The advantage of *d-gaps* coding is that locations could be represented in small integer: normally, an integer uses four Bytes, but two Bytes would be sufficient while coding in *d-gaps*.

In recent years, ideas of using *payload* in indexing Web pages were addressed, e.g. see [Arasu et al., 2001, Melnik et al., 2001]. In Web IR, retrieval strategies may treat the same term differently, for example, displayed in large font size to be more important than displayed in normal font size, or in bold face to be more important than in other font faces. Therefore, payload is proposed to stored additional information regarding to different locations, i.e. location list becomes a list of $\langle location, payload\rangle$ pairs[2]. Similarly, compressed coding methods are employed to ensure storage and retrieval to be space and runtime efficient.

5.1.3 Outline

In fact, the data structure of *word-level* inverted index is very similar to TID-index, because the locations in posting lists of inverted index is comparable to the tuple IDs of TID-index. Therefore, it is possible to integrate these two types of indexes and adapt to an IR+DB environment.

In this chapter, we present a relational inverted index (RIX) that takes advantages of both inverted index and TID-index. RIX speeds up probabilities estimation and context augmentation for text retrieval and remains supportable to relational operations for data retrieval, which smoothly balances flexibility, scalability and efficiency required in IR+DB systems. In the following sections, the technical details of RIX will be discussed in Section 5.2, while experiments

[2]Note that payload is not restricted to one value but could be arbitrary number of values.

and experimental results will be presented in Section 5.3, and finally this chapter will be summarised in Section 5.4.

5.2 Relational Inverted Index

In this section, we introduce a relational inverted index (RIX). The purpose of proposing RIX is to support efficient query processing on IR+DB systems. As we discussed in the previous chapter (see Chapter 2), text is loose in structure and schema, so that the best way to represent text is to keep everything in a knowledge-base. However, in order to obtain information from known knowledge including text and structured data, schema is necessary to define relationships amongst text and attributes. Therefore, an index for integrated text and data retrieval systems such as IR+DB should consider flexibility as well as efficiency and scalability.

To satisfy query processing involving text and data, an intuitive solution is to implement both inverted index and TID index into an IR+DB system, but the disadvantages of this simply-put-together approach are twofold. First, it makes query optimization and index selection more difficult. Because a query engine has to manage different index structures, and the search space for query optimizer is getting bigger, so that the complexity of estimating costs of using different indexes is increased. Second, a less sophisticated integration does not optimize space usage. Consuming more space would cost more, but it also affects the scalability and efficiency of a retrieval system, because more I/O overhead means more index construction time as well as more data fetching time during search.

In next sections, we start from comparing the data structures of inverted index and TID index, and then move on to the details of RIX which inherits the advantages of both indexes. In our discussions, all examples are based on a toy table (see Table B.1) in Appendix B for a toy magazine corpus demonstrated in Section 2.2.1 (also see Figure 2.2 and Figure 2.4).

5.2.1 Logical Designs of Indexing Structures

5.2.1.1 Inverted Index versus TID Index

Figure 5.1 illustrates an inverted document index built on the attributes Term and DocID[3]. In addition to the key (term) lookup, usually there is a document lookup built on DocIDs in which con-

[3]In the illustrations, the attribute values of DocID and RefDocID have been added a prefix "doc", this is only for the convenience of narration, while in our RIX implementation these values are the same as they are in the associated table.

tains global document statistic such as document length. Note that documents are pre-processed one by one, therefore tuples in a table are automatically grouped by DocIDs. In the example inverted index, the posting data of a document includes the starting offset (TID) of the document and the length of the document (the number of tuples with the same DocID).

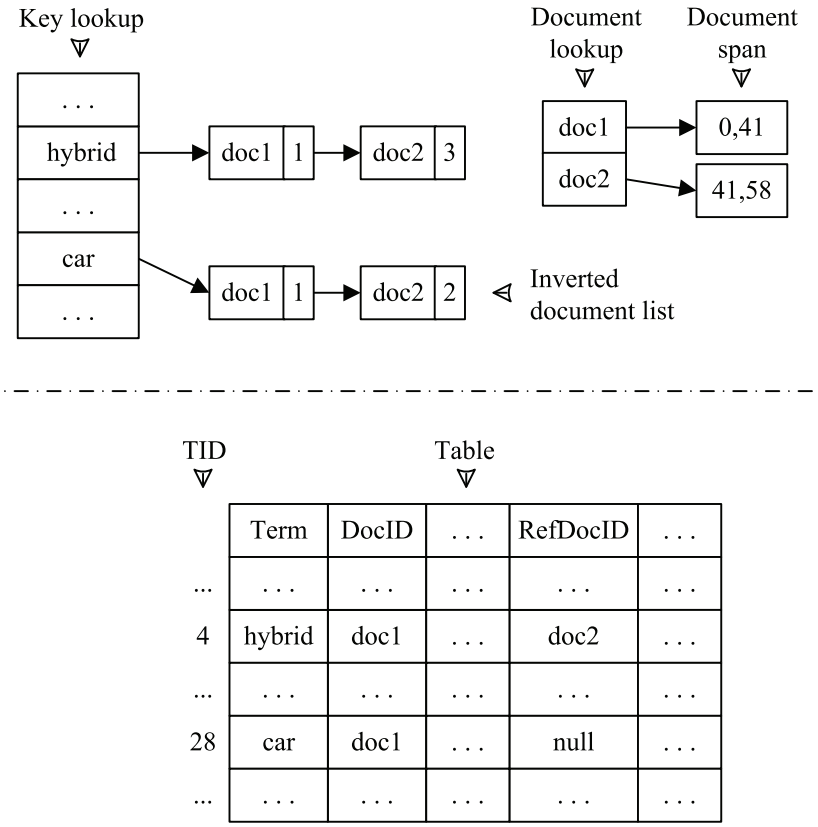


Figure 5.1: Data structure of traditional inverted document index

Figure 5.2 illustrates an TID index built on the attribute Term. The posting data of a key is a list of TIDs which can be used to directly access tuples in a table. Though additional statistic may be included, for example, the number of TIDs in a posting list. The advantage of this index is it provides shortcuts to access source data.

In principle, a document-level inverted index materialises aggregations group by Term and DocID, whereas a TID index can be viewed as a materialisation of aggregation group by Term only. If a key is allowed to have two posting lists, one for inverted documents and one for TIDs, then an index equivalent to a word-level inverted index could be built. Different from a word-level inverted index, inverted lists and TID lists (for locations information) are stored separately, which allows the index to be flexible; and only necessary posting list would be fetched for search also reduces I/O overhead. Allowing multiple posting lists associated to the same key becomes

the basic idea of RIX.

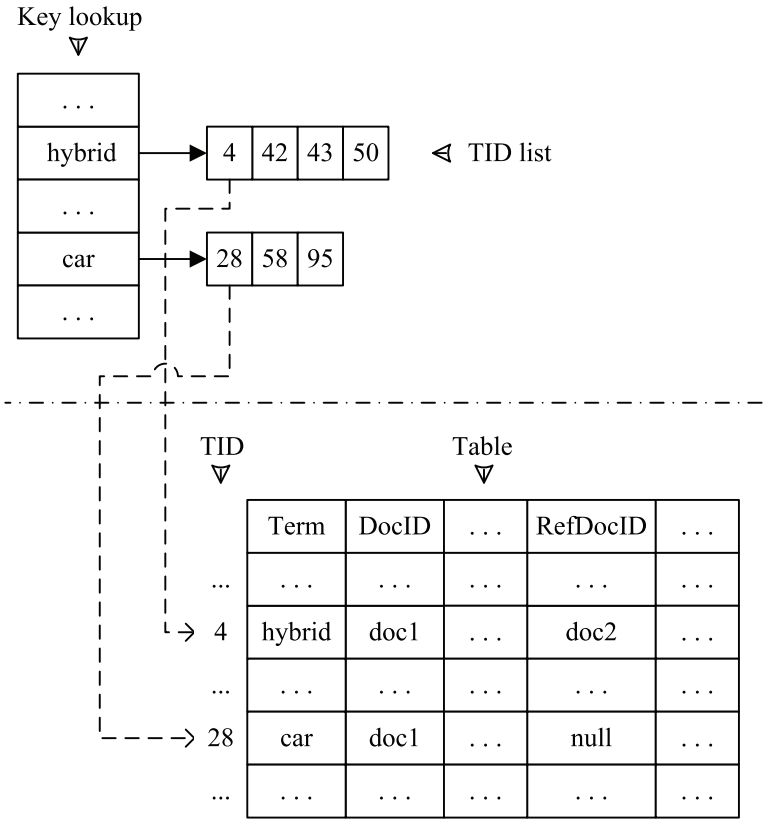


Figure 5.2: Data structure of traditional TID index

5.2.1.2 Structures of RIX

A RIX consists of two lookup facilities, one for primary key (on terms) and the other for primary group. The only requirement to be a primary group attribute is that tuples in the indexed table are grouped by the attribute. For example, in the case of document retrieval, the primary group would be document. In a light version of RIX (RixLite), as it is shown in Figure 5.3, each key associates with two posting lists, one for inverted groups and the other for TIDs. An inverted group unit is a pair $\langle groupID, in\text{-}groupTF \rangle$. Because a RixLite is a straightforward combination of traditional inverted index and TID index, hence there are not direct connections between inverted group list and TID list, so the inverted group list is standalone.

However, if connections could be made from inverted groups to TIDs, then the index would be very useful for supporting relational operations with aggregation. For example, while inverted group lists are sorted by in-group tuple frequencies, tuples in a table could be visited in group order. With this thought in mind, an additional field is added to each inverted group unit to

record a TID offset on the same key's TID list. This comes out to be the standard version of RIX (RixStd) which is illustrated in Figure 5.4. Similarly, RixStd has two posting lists, one called TID-mapped inverted group list, and another is TID list. Instead of containing pair values in an inverted group unit, a unit consists of a triple $\langle groupID, in\text{-}groupTF, TIDoffset \rangle$. Because the values in TID list are automatically organised in groups, so that the TIDs belonging to the same group are adjacent; and because the in-group tuple frequency is known, so in order to get all of the TIDs of a group, we only need to remember the position of the first TID in the TID list. Note that once posting lists were finished, performing sorting on TID-mapped inverted group lists would not impact their mapping against TID lists.

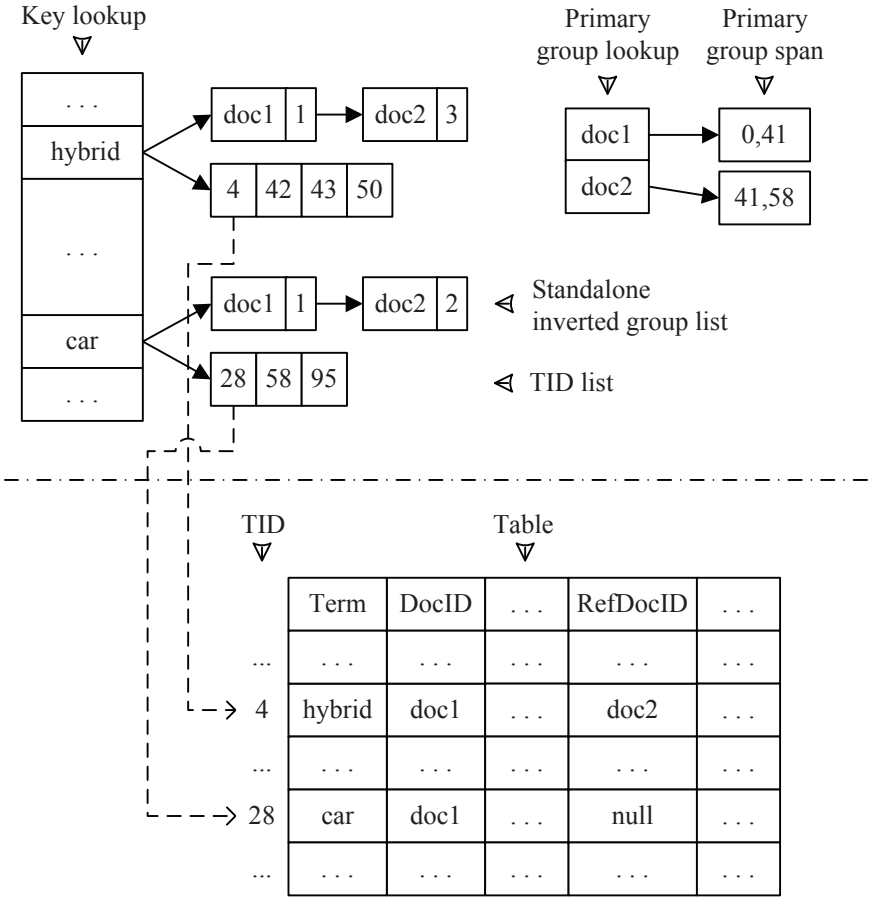


Figure 5.3: Data structure of light RIX: RixLite

For instance, the inverted group list of "hybrid" contains two inverted units, the group with ID "doc1" has one tuple, and its corresponding TID on the TID list is at position 0 in which value is 4. Similarly, the group "doc2" has three tuples, and the associating TIDs are the three adjacent values on the TID list starts from position 1, which are 42, 43, and 50.

Furthermore, a key may associate with more than one inverted group list. Although per RIX allows only one primary group, but additional inverted group lists could be added to satisfy aggregations on other attributes. Therefore, an extended RIX (RixExt) allows per key to have one TID-mapped inverted group list, one TID list, and several standalone inverted group lists, see Figure 5.5 for instance. In the figure, a standalone inverted group list is built on the attribute RefDocID (standing for referenced document)[4]. The reason to have several inverted group lists is because some IR strategies apply context augmentation on different groups, having multiple inverted group lists ready for augmentation would speed up query processing.

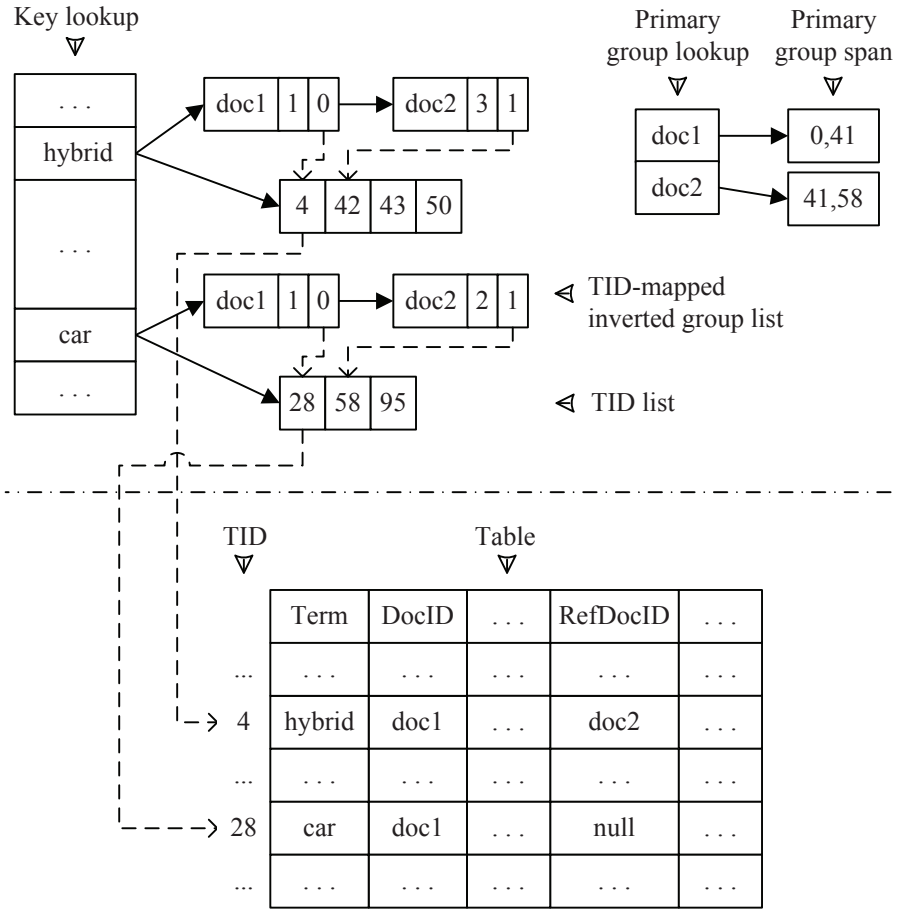


Figure 5.4: Data structure of standard RIX: RixStd

For example, while applying *tf*-boosting for link-based retrieval, terms appearing in anchor text are considered to have boosting effect to referenced documents, thus building an additional inverted group list for RefDocID would serve the purpose for augmentation. Because these additional inverted group lists do not map to the implicitly grouped TID lists, therefore they are

[4]The term "car" does not have any associated tuples falling into the RefDocID group, and an empty list was not shown in the figure.

standalone lists.

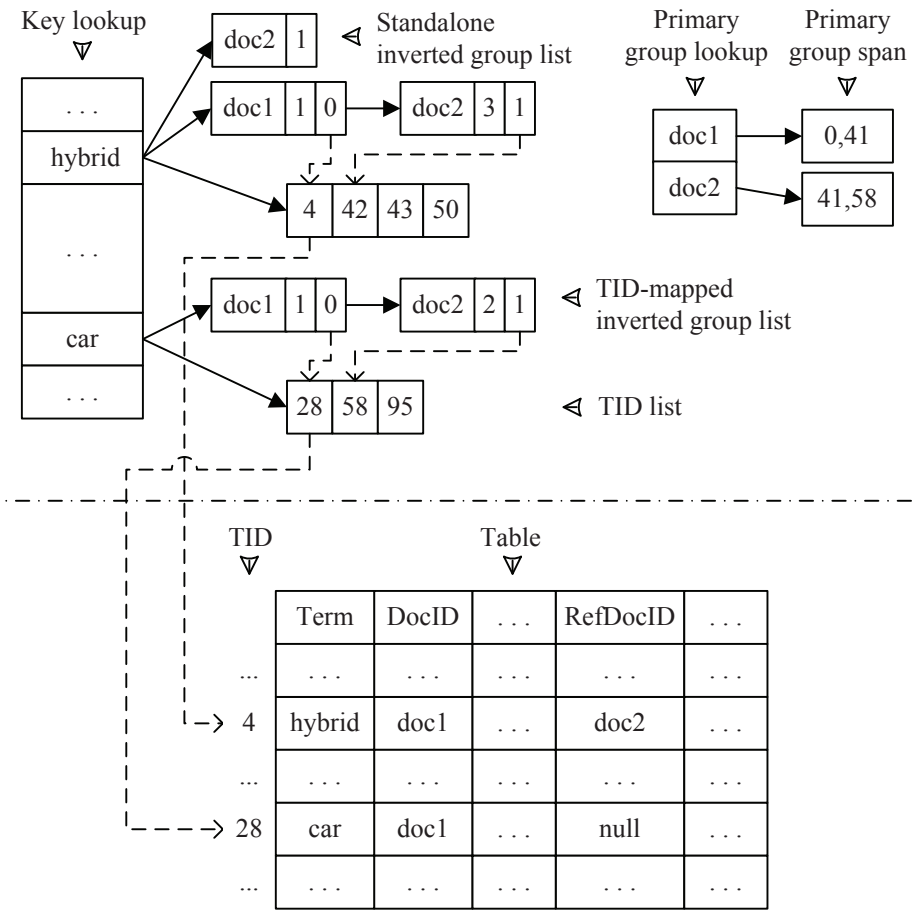


Figure 5.5: Data structure of extended RIX: RixExt

5.2.2 Architecture of RIX Indexer

Above all, the construction of RIX is performed by an indexer or index builder. In general, an indexer consists of several sub components, in which each of them in charge of a part of the indexing job while incorporating with each others.

For instance, Given a table to be indexed, tuples are read from the table by a fetcher under control of a scheduler, while the scheduler works based on rules or predefined cost models, so that sub-optimal constructing plans could be issued depending on different workload and resources. For example, given different table size or the number of tuples to be processed, and memory allowance. To conduct controls, a tuple validator is employed, and validation might also incorporate with traditional IR techniques such as stopwords removal. Validated tuples would be processed by other sub components, where accumulations to be performed, posting lists to be

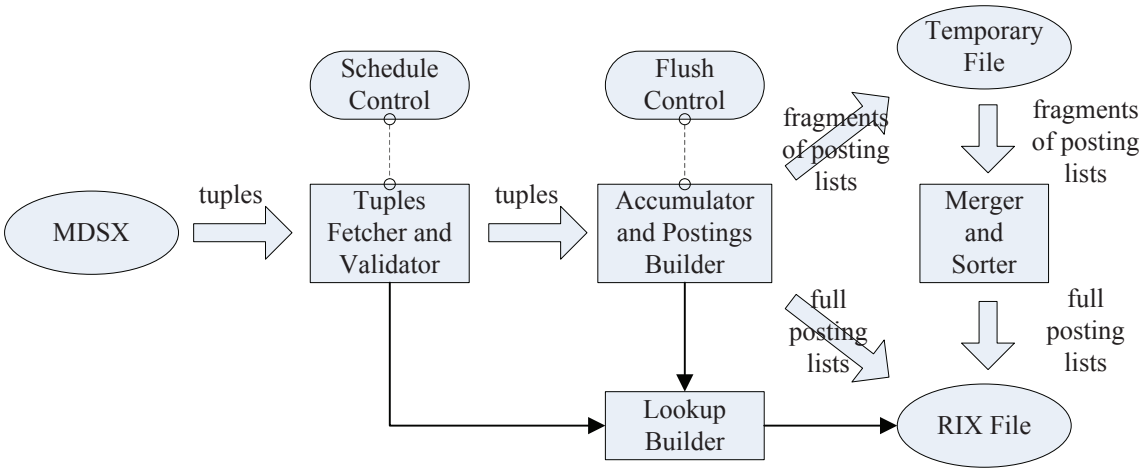


Figure 5.6: Architecture of RIX Indexer

constructed, and lookup facilities to be built. In particular, because memory is limited, therefore indexer need to flush partial built posting lists to disk so that memory could be released, and this is controlled by certain flushing policy. In addition, in order to provide better I/O performance for searching and retrieving on the index, partial lists should be merged and might be sorted at a later stage of indexing. An overview of the architecture of a RIX indexer is illustrated in Figure 5.6.

Because index size is often much bigger than main memory size, and the memory usage allowance of an indexer could be restricted by users as well, hence it is often unrealistic to construct a complete index in the memory for large-scale data. Therefore, an indexing job could be partitioned into several sub-jobs when it is necessary, where each sub-job completes a part of the index, and the entire index is obtained by merging all sub-parts. Previous researches on inverted index have studied different partitioning strategies for index construction (e.g. see [MacFarlane, 2000, Arasu et al., 2001, Melnik et al., 2001]). To build an inverted index for a very large text collection, partitioning could be based on term (i.e. key) or document (and i.e. primary group), while similar strategies are applicable to RIX. Some previous research (e.g. [MacFarlane, 2000]) on parallel and distributed building inverted indexes suggested that partitioning based on documents performs better than partitioning based on terms. However, because our current aim is to build RIX on standalone systems, and because the mapping between inverted group lists and TID lists requires indexer to trace down incremental offsets, therefore, algorithms applying key-based (terms) partitioning is more simple than group-based (documents) partitioning, and the algorithmic complexity of key-based strategy is also less than its group-based counterpart. Moreover, by exploiting a cost-based construction scheduling method, index

building process could be very efficient as well. More details will be discussed in the next a few subsections.

5.2.3 Abstract Data Types and Data Structures

Generally, when implementing indexes on external storage, because information could only to be stored in linear data structures, hence the abstract data types (ADTs) for external index need to be dedicated and optimized towards linear stored procedures and accessing methods. For RIX, the ADTs could be classified into either basic ADTs or operational ADTs. Basic ADTs are atomic data structures to store the entry of data or the exact data; whereas operational ADTs handle stored procedures and retrieved operations, in which each operational ADT contains a lists of certain basic ADT. Here we introduce these ADTs and their coding.

5.2.3.1 Basic ADTs

There are five basic ADTs to form RIX, which could be categorised into two classes: for data entries, or for posting data.

Key Entry A key entry contains a key (term), tuples count (TC) of the key (i.e. comparable to within-collection/global term frequency), groups count (GC) of the key (i.e. comparable to document frequency *df*), an address for posting TID list (PTA), an address for posting TID-mapped inverted group list (PMGA), and optionally a number of addresses for posting standalone inverted group lists (PSGA). A Key Entry looks like the follow:

$$\langle Key, TC, GC, PTA, PMGA\ [, PSGA_1, \ldots, PSGA_n] \rangle$$

The field for key may store original data, or store a numeric ID. For simplification, we take original data in our implementation. All other fields are numeric types, where TC and GC use double precision float (8 bytes), and addresses use long integer (8 bytes).

Group Entry A group entry is for storing the global statistic of primary group, which contains a primary group identifier (GID), an address of the first tuple in a table[5], and a tuple count of the group, i.e. group length (GL). The component is as the follow:

$$\langle GID, Address, GL \rangle$$

[5] We use absolute on-disk address in our implementation, another option could be using relative tuple offset.

Similarly, GID could be original data or numeric ID, and here we use original data as well. Address is long integer, and GL is double precision float.

Posting TID Unit A TID unit contains only one type of field, which is for on-disk address of a tuple, and it is using 8 bytes long integer.

$$\langle Address \rangle$$

Posting TID-mapped Inverted Group Unit A unit contains three fields, a group identifier (GID), an in-group tuple count (IGTC) of a key, and a TID offset on a posting TID list. The IGTC is a double precision float, and Offset is a long integer.

$$\langle GID, IGTC, Offset \rangle$$

Posting Standalone Inverted Group Unit It is similar to its TID-mapped counterpart, only without a field for TID offset, and the data types of its fields are the same as TID-mapped unit.

$$\langle GID, IGTC \rangle$$

5.2.3.2 Operational ADTs

Logically, there are two types of operational ADTs, which are hash lookup table and posting list. Hash based lookup allows the search for data entry to be completed in constant time (i.e. with computational complexity $O(1)$), therefore, hash lookup is usually preferred than B-tree on IR systems for text retrieval. The bucket collision problem of hash table is handled with a chain list strategy, where data entries share the same hash bucket would be chained together in a linear list, and a collided bucket stores the head of the list. To gain better I/O performance during construction, the hash lookup is partitioned and implemented with a hybrid hash algorithm [DeWitt et al., 1984]. The hash lookup and (chained) entry lists are stored in linear structures on disk respectively. In other words, search for the posting lists of a key needs two random accesses to obtain its data entries.

For example, assuming that hash value 2 was computed by both terms “hybrid” and “car”, which cause a collision on hash lookup table. The entries of the terms would be chained together during construction, and stored linearly in an entry list, as it is shown in Figure 5.7. At retrieval,

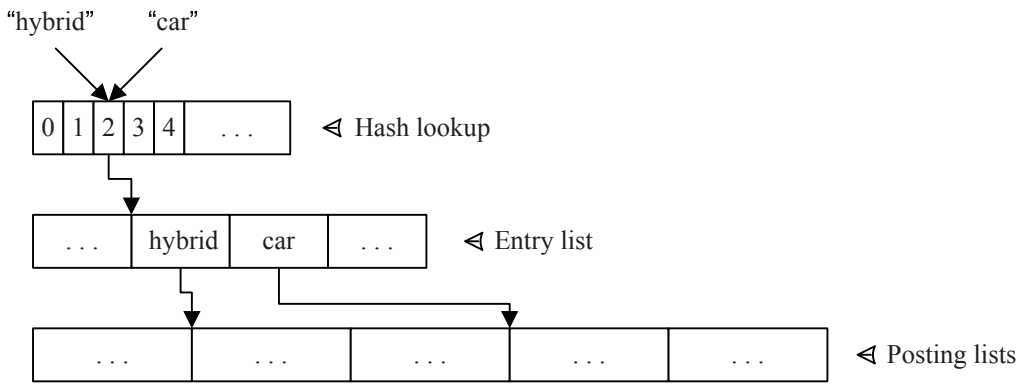


Figure 5.7: On-disk structure of operational components

while given a query term whose hash value was computed to be 2, we could obtain an address to access the entry list to get two entries, and then comparisons were made between the query term and the entries to determine which posting data should be retrieved.

Hybrid Hash Lookup A hash lookup table contains a list of buckets, and a bucket includes two fields: entry count (EC) and address. The entry count tells how many entries could be retrieved from this bucket, and the address gets access to associated chained entries, which is a segment of the entry list. A hash bucket looks as the follow:

$$\langle EC, Address \rangle$$

An EC is a small integer (2 bytes), and an address is a long integer (8 bytes). A hash lookup consists of a list of buckets.

$$\langle Bucket_1, \ldots, Bucket_n \rangle$$

Entry List An entry list contains a list of entry, where entry could be either Key Entry or Group Entry.

$$\langle Entry_1, \ldots, Entry_n \rangle$$

Posting List Similarly, a posting list a set of the same type of units, in which the type is either Posting TID Unit, or Posting TID-mapped Inverted Group Unit, or Posting Standalone Inverted Group Unit.

$$\langle Unit_1, \ldots, Unit_n \rangle$$

5.2.4 Theoretical Analysis

5.2.4.1 Overall Analysis

Here we analyse the theoretical performance of index construction of standard RIX (RixStd) on standalone machines. There are three kinds of overhead during indexing.

1. Pre-processing for source data. There are mainly two costs, which include I/O cost, e.g. read files from repository or read tuples from table; and compute cost, e.g. parsing source file or extracting values from tuples.

2. Internal computational cost of main constructing process, which includes building internal data structures, internal search and comparison, aggregation, merging, and sorting.

3. I/O cost of main constructing process, which include read and write of partial posting data, write of completed posting data, and write of external (on-disk) searching facilities.

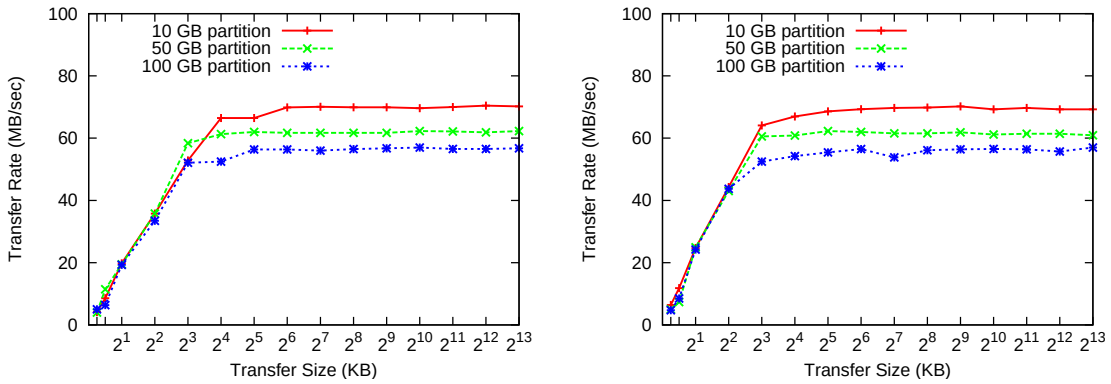
For RIX, the pre-processing is to read tuples from a table and extract values of attributes. In a table, tuples have already been organised (grouped) by primary group attribute, which is usually document ID (*DocID*). While indexer sequentially scans a given table, tuples would be read group by group, so that the indexer needs to extract the values of *DocID* to be aware of changes of groups. Because the cost of extracting tuples is too small so that it is ignorable, thus the cost of pre-processing is mainly disk read.

Similarly, the cost of the main constructing process is dominated by disk I/O. Considering internal process consists of the following sub-processes:

- Memory allocations for internal data structures and value assignments
- Internal search for KeyID based on hash-based lookup table
- Aggregation, mainly counting and accumulation
- Merging of partial posting data by concatenating sub-posting lists
- Internal sorting based on quick sort or radix sort, and external sorting based on merge sort

Except internal sorting, all other sub-processes could be implemented with algorithms having either constant complexity (i.e. $O(1)$) such as search over lookup tables, or linear complexity (i.e. $O(n)$) such as constructing internal data structures, aggregation and merging. Note that none of

the internal sub-processes need intensive computation, and quick sort or radix sort could be accomplished fast enough with today's CPU and main memory. On the other hand, disk transfer rate had been improved very slowly over past decades. Comparing to CPU and main memory, disk I/O rate is the biggest bottleneck of indexing for large scale data.



(a) Read (b) Write

Figure 5.8: Transfer rate against transfer size on varied sizes of partitions

5.2.4.2 Disk I/O Characteristics

To gain some comprehensive understanding about the features of disk I/O, we run a disk benchmark with a utility called ATTO Disk Benchmark (version 2.34). The testing hard disk was partitioned into varied sizes of partitions, and all partitions were formatted by the NTFS format with page (or cluster) size in 4.0 KB (4 096 bytes). The benchmark was performed by write/read a fixed length of data to/from a partition using different transfer (i.e. data block) sizes. Experiments were run in direct I/O (i.e. without buffering) and overlapped I/O (i.e. for example, two 2.0 KB data blocks would be filled into one 4.0 KB page). Table 5.1 shows the experimental results, while Figure 5.8 illustrates I/O rates of different transfer size over different size of partitions.

What can be learned from the benchmarking results is that a few characteristics of hard disk directly or potentially impact the efficiency of index construction and retrieval.

- *Transfer Size*: It had been known that transferred data size could directly affect I/O rate, especially when transfer size is smaller than page/cluster size then I/O rate drops dramatically, and the results show transfer rate is about in proportion to transfer size. Note that transfer rate has not reached maximum yet when transfer size is the same to page/cluster size, and it would continuously increase sub-linearly against transfer size until transfer size is about page/cluster size to the power of 2 or 3, and then transfer rate remains at a peak

Transferred Data Block Size (KB)	*Transfer Rate (MB/Sec)*					
	Drive Size 10 GB		*Drive Size 50 GB*		*Drive Size 100 GB*	
	Read	*Write*	*Read*	*Write*	*Read*	*Write*
0.5	4.151	6.400	3.968	5.007	4.992	4.724
1.0	8.575	11.825	11.434	7.349	6.352	8.491
2.0	19.770	24.717	19.456	24.901	19.311	24.212
4.0	35.750	44.281	35.750	43.008	33.457	43.704
8.0	52.851	64.093	58.370	60.532	52.076	52.461
16.0	66.444	66.941	61.286	60.831	52.461	54.226
32.0	66.444	68.611	61.972	62.266	56.357	55.404
64.0	69.868	69.288	61.680	61.972	56.375	56.496
128.0	70.089	69.722	61.680	61.536	56.013	53.828
256.0	69.905	69.813	61.680	61.536	56.496	56.133
512.0	69.905	70.179	61.709	61.851	56.751	56.394
1024.0	69.633	69.273	62.282	61.147	56.992	56.512
2048.0	69.996	69.723	62.137	61.426	56.512	56.394
4096.0	70.455	69.273	61.851	61.426	56.512	55.692
8192.0	70.179	69.273	62.282	60.928	56.751	56.992

Table 5.1: Disk I/O transfer rates on SATA Hard Disk (5400 rpm), Windows XP Professional OS, drive formatted by NTFS format, page/cluster size 4.0 KB (4 096 bytes), testing data length 256 MB, disk benchmarking utility ATTO Disk Benchmark v2.34, testing mode on Direct I/O and Overlapped I/O

value even transfer size keep increasing.

To gain sub-optimal transfer rate during RIX construction, it is important to keep the transfer size of each I/O operation (write or read) at proper amount. As a result, not only data entries and posting data should be transferred in batch mode, but also each (partial) posting list should be tried its best to pack into adjacent pages/clusters. Therefore, sophisticated buffering and flushing policies should be considered.

- *Drive Size*: It is interesting to see that drive size affects transfer rate while transfer size is bigger than page/cluster size. As a result, drive size potentially affects the maximum transfer rate that can be obtained during indexing, and it also directly restricts index size. This observation also implies parallelism and data distribution for building RIX is worth studying in future work.

- *Different Read/Write Rates*: Read rate is less than write rate unless transfer size is greater than two or three times of page/cluster size. This indicates that if partial posting lists are too short and too many then read operation is a considerable bottleneck while merging them; and if there are too many short posting lists then retrieval efficiency would be affected too.

5.2.4.3 I/O Cost Models for Building RIX

Regarding the construction cost of RIX we mean specifically the building time, but not the hardware or software costs of machine for running the program.

Variable	*Description*
B	page/cluster size (bytes)
V_θ	I/O transfer rate (bytes/sec) while transfer size equals to B
v_θ	estimated I/O transfer rate (bytes/sec)
t_θ	estimated I/O transfer time per page (sec)
λ_θ	levering coefficient for I/O operation

Table 5.2: System I/O variables

Variable	*Description*
N_t	the total number of tuples to be indexed
N_k	the total number of (distinct) keys
N_g	the total number of (distinct) groups
$n_{t,k}$	the number of tuples associated to a key
$n_{t,g}$	the number of tuples in a group (i.e. group length)
$n_{t,k,g}$	the number of tuples associated to a key in a group
$n_{g,k}$	the number of (distinct) groups that a key associates
S	transfer size (bytes)
s_t	average tuple size (bytes) of source table
s_k	(average) data size of a key entry (bytes)
s_g	(average) data size of a group entry (bytes)
s_b	data size of a hash bucket (bytes)
s_{pt}	data size of a posting TID unit (bytes)
s_{pm}	(average) data size of a posting TID-mapped inverted group unit (bytes)
s_{ps}	(average) data size of a posting standalone inverted group unit (bytes)
M	total memory buffer size (bytes)
M_t	tuples buffer size (bytes)
M_k	key entry buffer size (bytes)
M_g	group entry buffer size (bytes)
M_{pt}	posting TID list buffer size (bytes)
M_{pm}	posting TID-mapped inverted group list buffer size (bytes)
T	RIX construction time (sec)

Table 5.3: Data and indexing variables

The system I/O variables are introduced in Table 5.2, where page/cluster size B is usually 4 096 bytes in default. Though larger B allows better I/O rate for large data, but more disk space would be wasted if data are stored in lots of unpaged fragments. The connections of v, t and λ are defined in formula 5.1, in which if v is given then t can be obtained by inverse v and adjusted by λ.

$$t_\theta = \frac{1}{v_\theta} = \frac{\lambda_\theta}{V_\theta}, \text{ where } \theta \text{ is either read } (r) \text{ or write } (w). \tag{5.1}$$

Now let S be transfer size, so that the levering coefficient λ in equation 5.1 (also see Table 5.2) could be defined as follows:

$$\lambda = \begin{cases} 2^{(B/S)} & \text{if } 1 \leq S < B; \\ 1/\log_B S & \text{if } B \leq S \leq B^2; \\ 1/2 & \text{if } S > B^2. \end{cases} \tag{5.2}$$

As what equation 5.1 suggests, given transfer rate V_θ where $S = B$, transfer time per page t_θ for any size of S could be approximated by adjusting coefficient λ.

Some common data and indexing variables for I/O cost models are introduced in Table 5.3. In general, N indicates a total count of certain objects or contexts, n stands for a respective count of specified objects or contexts, s refers data size counted in bytes of components, M is for buffer size counted in bytes in main memory, and T is the cost in terms of building time. Different subscriptions are used for indicating specific objects or contexts. For instance, t for tuple, k for key, g for group, b for hash bucket, pt for posting TID unit, pm for posting TID-mapped inverted group unit, and ps for posting standalone inverted group unit.

Ideally, if main memory size could be unlimited then all data, including input and output, could be held in memory during construction, and the finally completed index would be materialised to disk with just one write operation. In this case, to obtain the I/O cost of index construction is to sum up the I/O costs of fetching tuples from source table (T_{fetch}), dumping posting lists ($T_{posting}$, including posting TID lists and posting inverted lists), dumping data entries (T_{entry}, including entry lists and lookups).

The I/O cost of fetching source tuples is given in equation 5.3. As aforementioned, t_r is average read time of one page size of data, s_t is average tuple size, N_t is the total number of tuples, and B is the page size. The ceiling of the division tells how many pages are there of source tuples, and the result of the multiplication of average read time per page and total pages gives the total read time of tuples, which is the fetching cost.

$$T_{fetch} = t_r \cdot \left\lceil \frac{s_t \cdot N_t}{B} \right\rceil \tag{5.3}$$

Next are dumping costs. To look at posting lists at first, we need to estimate how many pages

are used totally for posting lists, and the total size of postings could be computed by summing up posting lists size key by key. Hence for each key, the size of a posting TID unit is s_{pt} and there are $n_{t,k}$ TID units; similarly, the (average) size of a TID-mapped inverted group unit is s_{pm} and there are $n_{g,k}$ inverted units. Therefore, the I/O cost of dumping all posting data for N_k keys is to write all posting lists to disk, which is given in formula 5.4.

$$T_{posting} = t_w \cdot \left\lceil \sum_{i=1}^{N_k} \frac{s_{pt} \cdot n_{t,k_i} + s_{pm} \cdot n_{g,k_i}}{B} \right\rceil \tag{5.4}$$

The other dumping cost is for data entries. To recap, there are two kinds of instance of data entries, one for keys and the other for primary groups (see Section 5.2.1.2); each instance of data entries consists of a hash-based lookup and an entry list (see Section 5.2.3). Let the size of a hash bucket is s_b, the size of a key entry is s_k, and the size of a primary group entry is s_g, and it is known there are N_k keys and N_g primary groups, thus I/O cost of dumping data entries is calculated by formula 5.5.

$$T_{entry} = t_w \cdot \left\lceil \frac{(s_b + s_k) \cdot N_k + (s_b + s_g) \cdot N_g}{B} \right\rceil \tag{5.5}$$

Finally, the ideal total I/O cost could be obtained by formula 5.6.

$$\begin{aligned} T &= T_{fetch} + T_{posting} + T_{entry} \\ &= t_r \cdot \left\lceil \frac{s_t \cdot N_t}{B} \right\rceil + t_w \cdot \left\lceil \sum_{i=1}^{N_k} \frac{s_{pt} \cdot n_{t,k_i} + s_{pm} \cdot n_{g,k_i}}{B} + \frac{(s_b + s_k) \cdot N_k + (s_b + s_g) \cdot N_g}{B} \right\rceil \end{aligned} \tag{5.6}$$

However, main memory is limited in reality, and in most real-life applications it is impossible to construct an integrity of index in memory before materialising it to disk. Now assume an indexer has total M bytes memory usage allowance, and the allowance is allotted to different sub-processes, in which M_t for buffering fetched tuples, M_k for key entries, M_g for primary group entries, M_{pt} for posting TID lists, and M_{pm} for posting TID-mapped inverted lists. The buffer allocations is shown in formula 5.7.

$$M = M_t + M_k + M_g + M_{pt} + M_{pm} \tag{5.7}$$

In general, the indexer has to dump memory periodically to a temporary file, in which dumped data include key entry lists, group entry lists, fragments of posting TID lists and fragments of posting TID-mapped inverted lists. Note that internal data entry lookups would be released as

soon as entry lists were swapped out. Given buffering allowance M_t for tuples, the fetching cost is computed as formula 5.8.

$$T_{fetch} = t_r \cdot \left\lceil \frac{M_t}{B} \right\rceil \cdot \left\lceil \frac{s_t \cdot N_t}{M_t} \right\rceil \tag{5.8}$$

As a result of using temporary file for uncompleted posting data, extra I/O cost has to be paid. Let T_{tmp} be the cost of dumping temporary data, while the cost of loading these intermediate data would be taken into account later in $T_{posting}$. Since dumping fragments of posting lists are always performed in batch mode, where candidate lists for swap-out would be written to disk as big chunks of data in order to gain maximum write rate. Therefore, the cost of dumping posting data fragments to temporary file is similar to the cost of dumping completed posting lists in ideal situation, but the difference is the amount of data for swap-out are restricted to M_{pt} and M_{pm}. In other words, the upper bound of the number of pages used for posting TID fragments of every swap-out is $\left\lceil \frac{M_{pt}}{B} \right\rceil$, and $\left\lceil \frac{\sum_{i=1}^{N_k} \cdot s_{pt} \cdot n_{t,k_i}}{M_{pt}} \right\rceil$ number of swap-outs are needed for posting TID lists of all keys. Similarly, the counterparts of posting TID-mapped inverted lists could be measured in the same way. Combining both costs for TID lists and inverted lists, the final cost for dumping posting fragments is given by formula 5.9.

$$T_{tmp} = t_w \cdot \left(\left\lceil \frac{M_{pt}}{B} \right\rceil \cdot \left\lceil \frac{\sum_{i=1}^{N_k} \cdot s_{pt} \cdot n_{t,k_i}}{M_{pt}} \right\rceil + \left\lceil \frac{M_{pm}}{B} \right\rceil \cdot \left\lceil \frac{\sum_{i=1}^{N_k} s_{pm} \cdot n_{g,k_i}}{M_{pm}} \right\rceil \right) \tag{5.9}$$

Once all associated tuples of a key have been processed, then the posting data fragments (i.e. TID list fragments and inverted list fragments) of the key are ready to be merged into single posting lists. Generally, the I/O costs of merging posting fragments involves two parts: 1) cost of read posting fragments from temporary file; and 2) cost of write merged posting lists to index file. Since the lengths of posting lists of different keys are different[6], and the lengths of fragments of a posting list could be varied too, hence the read rates of posting fragments depend on individual cases. For the posting TID list of a key, let n'_{pt} be the number of fragments in temporary file, $m^{t,k}$ be the number of TID units of a fragment, t'_r be a read rate while loading a fragment, then $\left\lceil \frac{s_{pt} \cdot m^{t,k}}{B} \right\rceil$ is the number of pages used for the fragment, and by multiplying the read rate and the number of pages of the fragment we could obtain the read cost of the fragment. Summing up read costs of all fragments of the key then the total read cost is obtained. Let t'_w be the write rate while dumping final posting TID list of the key, and $\left\lceil \frac{s_{pt} \cdot n_{t,k}}{B} \right\rceil$ be the number of pages used for merged

[6]While comparing the same type of lists, i.e. either TID lists or TID-mapped inverted group lists.

TID list, and then the write cost of the posting TID list of the key could be also computed. While the total I/O cost of merging posting TID data is calculated by adding up the read cost of TID fragments and write cost of completed list. Let n''_{pm} be the number of fragments of inverted group list, $m^{g,k}$ be the number of inverted group units of a fragment, s_{pm} be the average size of inverted group unit, t''_r be a read rate of a fragment, t''_w be a write rate of final posting inverted list, and then the I/O cost of merging posting inverted group data could be computed similarly to the TID counterpart. Finally, the cost for all posting data is computed by formula 5.10.

$$T_{posting} = \sum_{i=1}^{N_k} \left(\sum_{j=1}^{n'_{pt}} t'_{r_j} \cdot \left\lceil \frac{s_{pt} \cdot m_j^{t,k_i}}{B} \right\rceil + t'_w \cdot \left\lceil \frac{s_{pt} \cdot n_{t,k_i}}{B} \right\rceil + \sum_{j=1}^{n''_{pm}} t''_{r_j} \cdot \left\lceil \frac{s_{pm} \cdot m_j^{g,k_i}}{B} \right\rceil + t''_w \cdot \left\lceil \frac{s_{pm} \cdot n_{g,k_i}}{B} \right\rceil \right) \tag{5.10}$$

Next, the costs for data entries are similar to the ideal situation, but swap-outs have to be taken into account. The I/O cost for posting entries is given in formula 5.11.

$$T_{entry} = t_w \cdot \left(\left\lceil \frac{M_k}{B} \right\rceil \cdot \left\lceil \frac{(s_b + s_k) \cdot N_k}{M_k} \right\rceil + \left\lceil \frac{M_g}{B} \right\rceil \cdot \left\lceil \frac{(s_b + s_g) \cdot N_g}{M_g} \right\rceil \right) \tag{5.11}$$

At last, the total I/O cost of constructing RIX with limited memory allowance is obtained by summing up all sub-costs, which is given in formula 5.12.

$$T = T_{fetch} + T_{tmp} + T_{posting} + T_{entry} \tag{5.12}$$

Although the total I/O cost T depends on all sub-costs, but as previously analysed, fetching source tuples, dumping batch posting fragments and entry lists involve sequential disk accesses, where maximum transfer rates could be obtained; whereas while merging posting fragments, transfer rates were bounded by frequent random I/O accesses, which becomes serious bottleneck during RIX construction. Therefore, to shorten indexing time it is important to reduce the $T_{posting}$ cost.

By reviewing formula 5.10, it is easily found that the less number of fragments of a posting list the more efficient merging of fragments could be performed. Thus the lower bound of merging cost of a posting list reaches when the list has only one fragment, i.e. the fragment is an entire list itself; whereas the upper bound of merging cost occurs when each fragment contains one only posting unit. As a result, we have the following inequalities:

$$t'_r \cdot \left\lceil \frac{s_{pt} \cdot n_{t,k}}{B} \right\rceil \leq \sum_{j=1}^{n'_{pt}} t'_{r_j} \cdot \left\lceil \frac{s_{pt} \cdot m_j^{t,k}}{B} \right\rceil \leq \sum_{j=1}^{n_{t,k}} t'_{r_j} \cdot \left\lceil \frac{s_{pt}}{B} \right\rceil \tag{5.13}$$

and

$$t''_r \cdot \left\lceil \frac{s_{pm} \cdot n_{g,k}}{B} \right\rceil \leq \sum_{j=1}^{n''_{pm}} t''_{r_j} \cdot \left\lceil \frac{s_{pm} \cdot m_j^{g,k}}{B} \right\rceil \leq \sum_{j=1}^{n_{g,k}} t''_{r_j} \cdot \left\lceil \frac{s_{pm}}{B} \right\rceil \tag{5.14}$$

In general, there are two options to reduce the number of fragments of posting lists, one is to increase memory size and buffer allowance for posting lists, and the other is to employ partitioning strategies. No doubt the first option is straightforward but less scientific value, and eventually there is an upper bound of memory, hence it is necessary to study the second option and design sophisticated constructing algorithms.

Note that usually there are two kinds of partitioning strategies, one is to partition by keys (e.g. *TermID*), and the other is by primary groups (e.g. *DocID*). Different from inverted (document) indexes, for standard RIX, the problem of group-based partitioning is that offset values of posting TID-mapped inverted group lists (see Section 5.2.3 for posting TID-mapped inverted group unit) have to be updated during merging. The disadvantages are multi-folds: on the one hand, it makes indexing algorithms more complicated and introduces extra computational cost; on the other hand and more important, it does not solve the *frequent random access* problem causing by too many posting fragments, because group-based partitioning produces short posting lists, which leads merging to confront the same problem as against fragmented posting lists.

5.2.5 Construction Procedures

5.2.5.1 Outline and Data Flow

In this subsection, we discuss the index construction of RIX. Among the three structures that have been addressed previously, since RixStd (see Figure 5.4) costs only one extra field more than RixLite (see Figure 5.3) but provides more functionality, therefore RixStd is the structure would be built in default. In addition, some address fields for marking posting list entries would be left empty in the key's look-up, so that RixStd could be extended to RixExt (see Figure 5.5) after initial construction run.

Currently, our methods only focus on building RIX with single machine, where all techniques and algorithms were designed to make RIX construction to be efficient and scalable in standalone environment. However, we were fully aware of other available techniques, such as parallel or dis-

tributed computing, and their capability of speeding up and scaling up index construction. Since parallelism and data distribution are mature techniques for constructing indexes, especially on building inverted indexes for Web IR, where commercial Web search engines have being utilised massive number of parallel commercial machines[7]. Previous studies on parallel index construction include, e.g. [MacFarlane, 2000, Arasu et al., 2001, Melnik et al., 2001]. Therefore, parallelism and distributed computing are enabling techniques to improve the efficiency and scalability of existing standalone algorithms, but a premise is that counterpart standalone techniques do exist. At our best knowledge, there are no other indexes act similarly as RIX, and this is the first time that a standalone construction technique of RIX would be discussed. Hence, the standalone methods that are going to be described here could be a baseline for possible parallel methods in the future.

Description of Data Size	*Amounts of Data*			
	Bytes	*Tuples*	*Keys*	*Primary Groups*
Small	$<$ 1 GB	$<$ 100 million	$<$ 50 thousand	$<$ 10 thousand
Medium	$<$ 5 GB	$<$ 1 billion	$<$ 100 thousand	$<$ 100 thousand
Large	$<$ 10 GB	$<$ 10 billion	$<$ 1 million	$<$ 1 million
Very Large	$\geq$ 10 GB	$\geq$ 10 billion	$\geq$ 1 million	$\geq$ 1 million

Table 5.4: Approximate description of data size

With regards to data size, rather than referring the size of original corpus (document collections) such as in IR, we mean the size of tables such as in DB, specifically, the size of tables to be indexed. In Table 5.4, some approximate descriptions about data size are clarified in terms of different measurements.

From the indexer architecture shown in Figure 5.6, we distil and illustrate the data flow in a Data Flow Diagram in Figure 5.9. Among these procedures, tuple fetching and validation are managed by building schedules, where three scheduling algorithms named Naive Build, Adaptive Build and Analytical Build will be discussed. Before that, there are several common processes of construction will be studied at first, which include accumulating for semi-finished postings, making posting lists, flushing policy during tuple insertion phase, and merging and sorting of fragments of posting lists during finalised phase. In general, construction could be divided into three phases: insertion phase, merging phase, and finalising phase. Posting lists are built in the

[7]Using low-cost commercial machines instead of expensive servers is to obtain the best cost-and-gain ratio when considering building parallel network contains tens of thousands of notes. As far as we know, a machine cluster at Google's data centre could contain at least 10 000 notes. For example, see [Dean and Ghemawat, 2004].

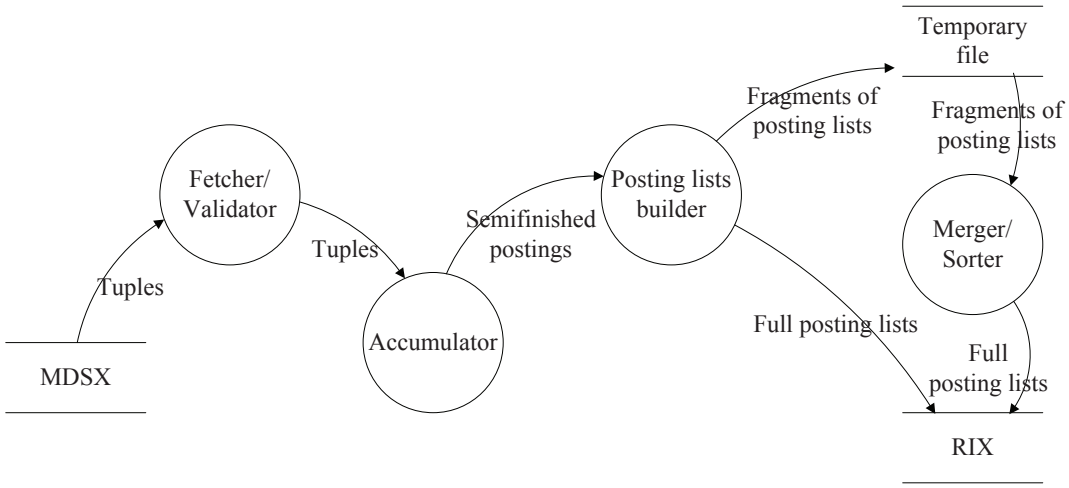


Figure 5.9: Data Flow Diagram of general procedures

former two phases, and external lookups is built in the last phase. In addition, a flowchart of the insertion phase is shown in Figure 5.10.

5.2.5.2 Building Algorithms

The common algorithms for the constructing procedures illustrated in Figure 5.9 and Figure 5.10 are discussed in detailed. Here we analyse algorithmic complexities of the construction procedures, in which we show internally constructing process has sublinear computational complexity. In addition, constructing algorithms were proposed while sophisticated scheduling methods were employed, for instance, to consider memory allowance and data size and to balance buffering and flushing. To get started, let us have a look at the algorithm for accumulating semi-finished posting data.

Accumulating semi-finished postings Before posting lists for TIDs and inverted groups could be made, statistics of keys based on primary groups are accumulated, in which keys' in-group tuple counts (i.e. comparable to within-document *tf*) are computed and associated TIDs are gathered. This procedure is on the left-hand-side of the flowchart in Figure 5.10. An accumulator has a small hash table[8] for fast lookup for accumulating items of keys. Accumulating items are semi-finished postings which would be ingredients of posting units for making posting lists.

The algorithm *Accumulate* is given in Figure 5.11, which includes a loop for processing received tuples. The first job is to extract Key (e.g. *TermID*), GroupID (e.g. *DocID*) and TID (i.e. TupleID, which could be on-disk address or offset of the tuple) from tuples (see line 3). If

[8]It is much smaller than the KeyLookup of RIX and it needs not to be materialised

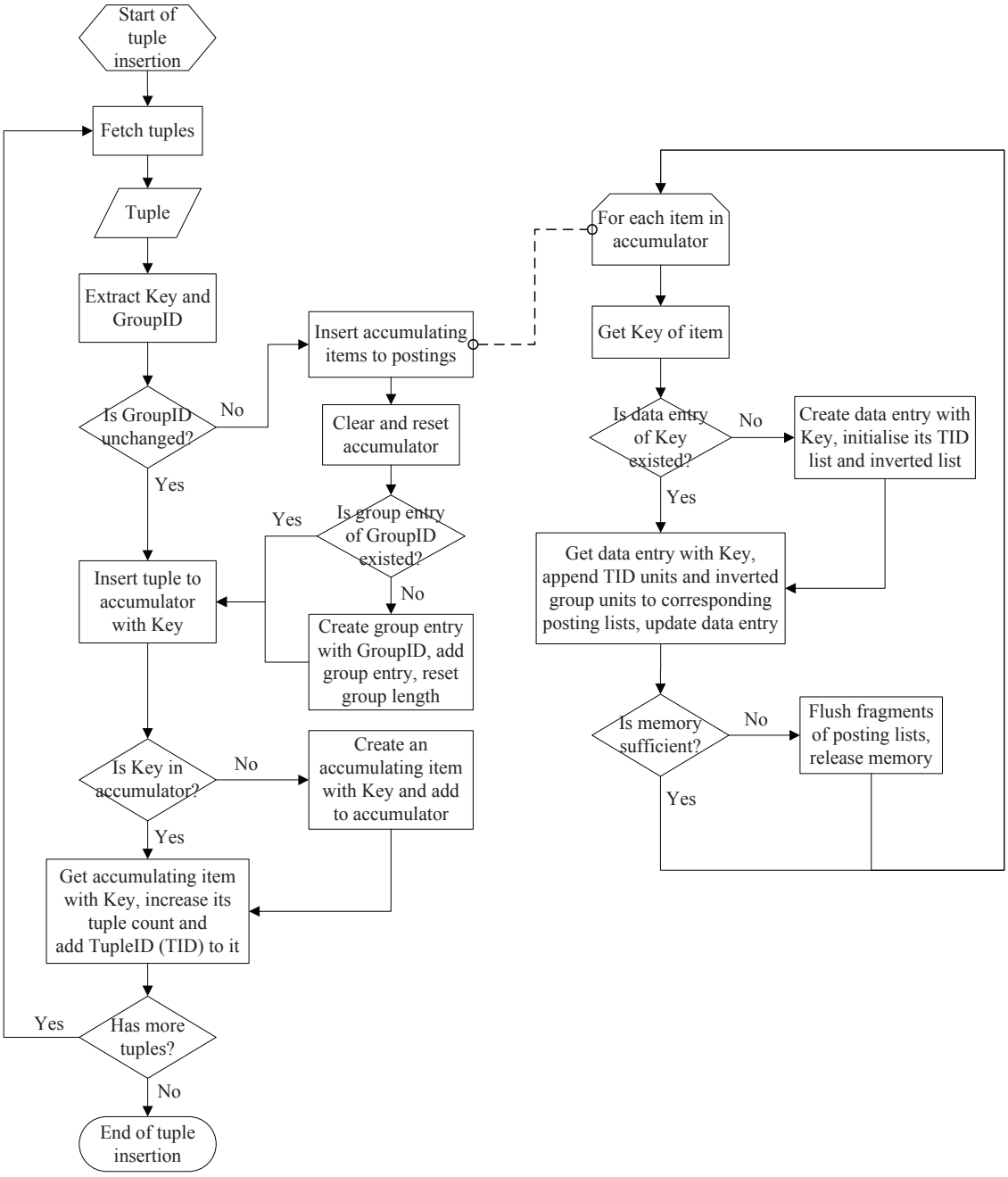


Figure 5.10: Flowchart of general procedures

Algorithm: Accumulate
Input: Tuples
Output: Semi-finished postings (accumulated Items in Accumulator)

```
1  receive Tuples;
2  foreach tuple in Tuples
3    extract Key, GroupID and TID from tuple;
4    if GroupID is not equal to LastGroupID
5      send GroupID and Accumulator->Items to Posting Lists Builder,
           reset Accumulator, LastGroupID = GroupID;
6    endif
7    if Accumulator->Items does not contain Key
8      TupleCount = 0, TIDArray = new Array();
9      Item = new Item(TupleCount, TIDArray);
10     Accumulator->Items->Add(Key, Item);
11   endif
12   Accumulator->Items[Key]->TupleCount++;
13   Accumulator->Items[Key]->TIDArray->Add(TID);
14 endforeach
```

Figure 5.11: Accumulating semi-finished postings

each tuple is stored in bytes array and a schema of tuple (same schema as a source table) is given, then extracting values from tuple needs to convert bytes to specified types of data. Next jobs include creating accumulated items for Keys (i.e. lines 7 to 11) and updating accumulated values of Keys (i.e. lines 12 and 13). All of these jobs should be processed and computed very fast on today's machines. The complexity of the algorithm is in proportion to the number of tuples to be processed. If let n be the size of the loop, then the complexity is expressed as $O(n)$.

Making Posting Lists Semi-finished postings or semi-postings based on primary groups are sent to posting lists builder to make posting lists. This is on the right-hand-side of the flowchart in Figure 5.10. Semi-postings of keys are processed one by one to make posting units to insert into corresponding posting lists. Meanwhile, work-in-progressed fragments of posting lists would be dumped to disk periodically in order to release main memory.

The algorithm is given in *MakePostingLists* as shown in Figure 5.12. For each item (represented as <Key, item> pair) in semi-posting items, if the Key is not in KeyLookup then a KeyEntry is created (see lines 3 to 7), otherwise corresponding KeyEntry will be found. To update TID-mapped inverted group list of a KeyEntry, the TID mapping offset would be assigned at first (see line 8), which is the tuple count (before updated), and then a TID-mapped inverted group unit will be created and appended to the end of the list (see lines 9 and 10). Next, TID units will be created and added to the end of TID list (i.e. lines 11 to 14). At last, global statistics of the KeyEntry will be updated (i.e. lines 15 and 16).

Algorithm: MakePostingLists
Input: GroupID and semi-finished postings
Output: Fragments of posting lists or full posting lists

```
1  receive GroupID and Items of semi-finished postings;
2  foreach <Key, Item> pair in Items
3    if KeyLookup does not contain Key
4      TIDList = new TIDList(), GroupList = new GroupList();
5      KeyEntry = new KeyEntry(Key, TIDList, GroupList);
6      KeyLookup->Add(Key, KeyEntry);
7    endif
8    Offset = KeyLookup[Key]->TupleCount;
9    GroupUnit = new GroupUnit(GroupID, Item->TupleCount, Offset);
10   KeyLookup[Key]->GroupList->Add(GroupUnit);
11   foreach TID in Item->TIDArray
12     TIDUnit = new TIDUnit(TID);
13     KeyLookup[Key]->TIDList->Add(TIDUnit);
14   endforeach
15   KeyLookup[Key]->TupleCount += Item->TupleCount;
16   KeyLookup[Key]->GroupCount++;
17 endforeach
18 write posting lists by FlushControl;
```

Figure 5.12: Make posting lists

In the algorithm, the size of the outer loop is equal to the number of keys in a primary group, whereas the sizes of the inner loop are varied to each key. However, the overall size of all inner loops (i.e. summation of all inner loop sizes) is equal to the number of tuples given primary group (i.e. the group length). Therefore, the complexity of the algorithm is a sum of complexities of making TID-mapped inverted group lists and TID lists. Let m be the number of keys and n be the number of TIDs, then the complexity for updating inverted lists is $O(m)$ and for TID lists is $O(n)$, and the overall algorithmic complexity is $O(m+n)$.

Flushing Control Here we discuss the flushing policy for posting lists while memory usage exceeding allowance. In general, there are two principles for flushing: 1) lists to be flushed should be as longer as possible, and 2) lists should be flushed as later as possible. The principles are based on a hypothesis that if swapping out a few longest lists may relieve memory pressure, then flushing of other lists could be postponed until those shorter lists become longer. Both principles aim to reduce the number of fragments of posting lists caused by memory swapping.

Memory dumping is triggered when buffer usage of posting lists exceeding allowance. Depending on real-time memory pressures so that different levels of actions are to be performed. Actions for three different memory pressure levels are:

1. If memory pressure is greater than or equal to level one but less than level two, then swap

out the longest *N* posting lists (applicable to both TID lists and inverted lists);

2. If memory pressure is greater than or equal to level two but less than level three, then swap out those lists that are longer than a threshold. The threshold is usually set to a multiple of page/cluster size of system;

3. If memory pressure is greater than or equal to level three (the maximal level), then swap out all posting lists in memory.

Figure 5.13 shows an algorithm *FlushControl* for conducting the flushing policy.

Algorithm: FlushControl

```
1  assess buffer usage;
2  if buffer usage is greater than allowance
3    if memory pressure is greater than or equal to level three
4      write all posting lists;
5    else if memory pressure is greater than or equal to level two
6      write posting lists longer than threshold;
7    else if memory pressure is greater than or equal to level one
8      write longest N posting lists;
9    endif
10   reassess buffer usage and update memory pressure level;
11 endif
```

Figure 5.13: Flushing policy

To find out the longest *N* posting lists for level one flushing, a small array is used to register the pointers (i.e. in-memory addresses) of candidate lists, and the array is kept updated while inserting posting units to posting lists. An algorithm for updating the register array is given in Figure 5.14. First of all, when the register array is not full, then pointers of posting lists will be directly added to the array (see lines 6 and 7); otherwise, if a posting list has been registered before and its length is greater than the list one position ahead of it, then swap the positions of those two pointers in the array (i.e. lines 10 to 16); else if a posting list has not been registered before but its length is greater than the list at the last position in the array, then reset the array's last value to the new pointer (i.e. lines 17 to 19).

Algorithm *RegisterTopLists* (Figure 5.14) does not maintain exact orders of longest lists, but it increases the chances of longer lists to stay in the register, whereas shorter lists would be soon replaced by longer lists. Since linear search is used for finding pointers within register, let m be the size of register, the average complexity for looking up a pointer is $O(\frac{m}{2})$; for total n keys where each key has two posting lists, the average complexity for updating the register is

Algorithm: RegisterTopLists

```
1  /* initialise Register when tuple insertion starts
2     Register = new Array[N];
3  */
4  receive Key and KeyEntry;
5  Pointer = &KeyEntry->PostingList; /* PostingList is either TIDList
       or GroupList */
6  if Register->Length is less than N
7    Register->Add(Pointer);
8  else
9    IsFound = false;
10   foreach position i in Register
11     if Register[i] is equal to Pointer and Register[i]->Length is
           greater than Register[i - 1]->Length
12       Swap(Register[i], Register[i - 1]);
13       IsFound = true;
14       break;
15     endif
16   endforeach
17   if IsFound is false and Register[N - 1]->Length is less than
         Pointer->Length
18     Register[N - 1] = Pointer;
19   endif
20 endif
```

Figure 5.14: Register longest N posting lists

$O(\frac{m}{2} \cdot n \cdot 2) = O(m \cdot n)$. Therefore, in order to minimise the overhead of updating register, it is important to keep register size very small, e.g. set m at most to one hundred.

Merging Fragments of Posting Lists Fragments of posting lists are merged at the end of a tuple insertion run[9]. In this phase, a merger reads posting fragments from a temporary file, and concatenates the fragments of a posting list into one whole list. The merger performs merging key by key until all posting lists of keys have been finished. To support top-k processing during retrieval, inverted group lists could be sorted by *IGTC* (i.e. in-group tuple count, see Section 5.2.3) before finalised and written to RIX file.

A merging algorithm *MergePostingLists* is given in Figure 5.15. For each key, fragments of TID-mapped inverted group list are merged at first (see lines 2 to 9), and then fragments of TID list are merged as well (see lines 10 to 17). Since each posting fragment contains an on-disk address of its previous fragment, so to read fragments one by one that an entire posting list could be restored, and merged lists will be materialised into RIX. With regards to computational complexity of merging, the size of the outer loop is decided by the number of keys in KeyLookup;

[9]Depending on building schedules, a RIX construction process may consist of one or several tuple insertion runs.

Algorithm: MergePostingLists
Input: Fragments of posting lists
Output: Full posting lists

```
1  foreach Key in KeyLookup
2    Address = KeyLookup[Key]->GroupList->Previous;
3    while Address is not empty
4      Fragment = Read(Address);
5      GroupFragment = new GroupList(Fragment);
6      KeyLookup[Key]->GroupList->MergeSort(GroupFragment);
7      Address = GroupFragment->Previous;
8    endwhile
9    Write(KeyLookup[Key]->GroupList);
10   Address = KeyLookup[Key]->TIDList->Previous;
11   while Address is not empty
12     Fragment = Read(Address);
13     TIDFragment = new TIDList(Fragment);
14     KeyLookup[Key]->TIDList->Merge(TIDFragment);
15     Address = TIDFragment->Previous;
16   endwhile
17   Write(KeyLookup[Key]->TIDList);
18 endforeach
```

Figure 5.15: Merge fragments of posting lists

whereas the sizes of the two inner loops are varied, which depend on the number of fragments produced in the insertion phase. Let n be the number of processed keys, and m be the number of processed tuples, then in the best case, each key has one group list fragment and one TID list fragment, thus the best complexity is $O(2n) = O(n)$; whereas in the worst case, each key has $\lceil \frac{m}{n} \rceil$ number of group list fragments and the same number of TID list fragments, so that the worst complexity $O\left(n \cdot \frac{m}{n} \cdot 2\right) = O(2m) = O(m)$. In summary, the computational complexity of merging is between in proportion to the number of keys and the number of processed tuples.

Build External Hash Lookup In the finalising phase, in-memory lookup for keys and lookup for groups would be materialised to disk; meanwhile, key entry list and group entry list would be reorganised when corresponding external lookup is built. An algorithm *BuildExternalLookup* for building external hash-based lookup is given in Figure 5.16.

In general, the on-disk linear structure of a hash lookup would be constructed in memory and then dumped to disk. Therefore, an external lookup would be stored in an array during construction. Collided entries (i.e. data entries sharing the same hash bucket) would be chained into lists. In algorithm *BuildExternalLookup*, an array is initialised for hash lookup at the beginning (i.e. line 3); then a given entry list is processed to fill up the hash lookup (see lines 4 to 8), in which direct accessing BucketIDs are computed from EntryIDs by a hash function (i.e. line 5), and

Algorithm: BuildExternalLookup
Input: EntryList
Output: Reorganised EntryList and external Lookup

```
1  begin
2  /* an EntryList is either a KeyEntryList or a GroupEntryList */
3    Lookup = new Array();
4    foreach Entry in EntryList
5      BucketID = HashFunction(Entry->ID);
6      Lookup[BucketID]->CollidedEntry->Add(Entry);
7      Lookup[BucketID]->EntryCount++;
8    endforeach
9    foreach position i in Lookup
10     Lookup[i]->Address = Write(Lookup[i]->CollidedEntry);
11   endforeach
12   foreach position i in Lookup
13     Write(Lookup[i]->EntryCount, Lookup[i]->Address);
14   endforeach
15 end
```

Figure 5.16: Build external hash lookup

entries are inserted to buckets with assigned BucketID (i.e. line 6), while an EntryCount variable of each bucket is updated (i.e. line 7) to record the number of entries sharing the bucket; in the next step, chains of entries in the lookup array are written to disk, while accessing addresses for entries in the lookup would be updated (see lines 9 to 11); in the last step, the lookup array is materialised to disk (i.e. lines 12 to 14). The computational complexity of the algorithm is in proportion to the length of given entry list, let n be the number of entries, so the complexity is $O(3n) = O(n)$.

5.2.5.3 Scheduling Algorithms

So far we have discussed the common building algorithms for sub-processes needed for constructing RIX. To enable RIX construction to be scalable, we investigate different building schedules, in which a naive algorithm is firstly introduced, and then an adaptive algorithm and a analytical algorithm are proposed.

Figure 5.17 illustrate the flowcharts of three scheduling procedures. Next, we discuss the details of scheduling algorithms.

Naive Build First of all, if the three building phases, i.e. insertion, merging and finalising, are run sequentially, then a constructing schedule of the base line is obtained, and this is the simplest building schedule and called the naive schedule. Figure 5.18 shows an algorithm *NaiveBuild* for naive scheduling.

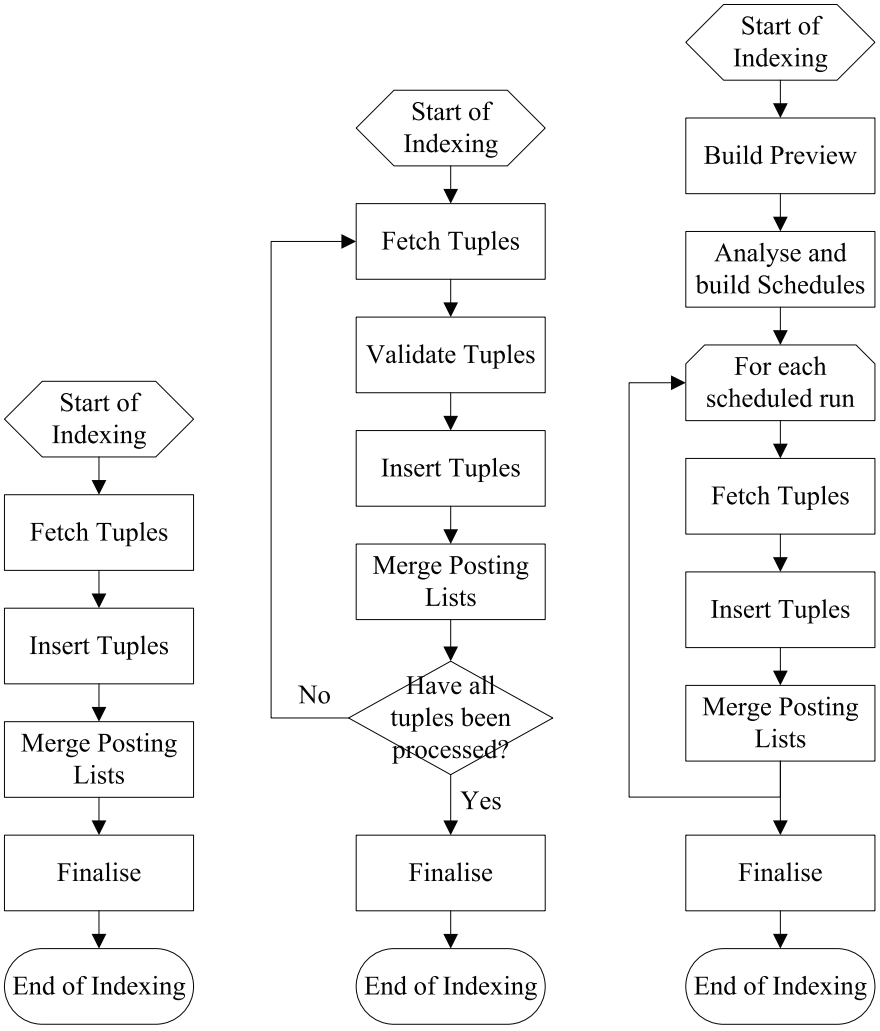


Figure 5.17: Flowcharts of building schedules

Algorithm: NaiveBuild
Input: Table
Output: RixStd

```
1  begin
2    Tuples = Read(Table);
3    while Tuples is not empty
4      ValidTuples = Validate(Tuples);
5      Accumulate(ValidTuples);
6      Tuples = ReadNext(Table);
7    endwhile
8    MergePostingLists();
9    BuildExternalLookup(KeyEntryList);
10   BuildExternalLookup(GroupEntryList);
11   delete temporary file;
12 end
```

Figure 5.18: Constructing RIX with naive scheduling

As demonstrated in Section 5.2.4, RIX construction by naive scheduling cannot be scalable for large data, this is due to I/O cost exponentially increases that led by frequent random accesses for small posting fragments.

Because posting fragments are caused by limited memory allowance for buffering, therefore, one of the solutions is to reduce indexed tuples. For example, stopwords removal could be performed by a tuple validator, and similar techniques have been applied in IR systems or search engines. Empirically, such techniques are pragmatic and useful.

Adaptive Build However, if the size of source data is very large and even applying stopwords removal cannot reduce the amount of posting fragments, additional techniques such as data partitioning or job partitioning could be employed.

Algorithm: AdaptiveBuild
Input: Table
Output: RixStd

```
1  begin
2    SkipPoint = 0, RunCount = 0;
3    while SkipPoint is set
4      RunCount++;
5      Tuples = Read(Table, SkipPoint);
6      unset SkipPoint;
7      while Tuples is not empty
8        foreach tuple in Tuples
9          if Validate(tuple) is true;
10           ValidTuples->Add(tuple);
11         else if SkipPoint is unset
12           SkipPoint = tuple->TID;
13         endif
14       endforeach
15       Accumulate(ValidTuples);
16       Tuples = ReadNext(Table);
17     endwhile
18     MergePostingLists();
19     Write(KeyEntryList), release memory and reset KeyLookup;
20     if RunCount is equal to 1
21       Write(GroupEntryList), release memory;
22     endif
23     delete temporary file;
24   endwhile
25   BuildExternalLookup(KeyEntryList);
26   BuildExternalLookup(GroupEntryList);
27 end
```

Figure 5.19: Constructing RIX with adaptive scheduling

As we have mentioned, data partitioning has been well studied in parallel and distributed indexing (e.g. see [MacFarlane, 2000, Arasu et al., 2001, Melnik et al., 2001]). On the other hand,

job partitioning is suitable for RIX construction on standalone machines. Hence, an adaptive scheduling method is to partition tuples insertion into several runs based on real-time memory usage, an algorithm *AdaptiveBuild* for adaptive scheduling is demonstrated in Figure 5.19.

In *AdaptiveBuild*, the Validate (i.e. line 9) function not only performs stopwords removal, but also rejects tuples when certain threshold has been exceeded. For example, a threshold could be set to the number of acceptable keys in a run, which could be a fixed value or an adaptive value. In addition, a variable SkipPoint is used to remember the address of a tuple that is firstly skipped in a run (see line 12), so that scanning on the source table in the next run could be started at the SkipPoint (see line 5). Note that constructing of GroupEntryList could be accomplished in the first run (i.e. lines 20 to 22), where the allotted buffer for GroupEntryList would be released after the list is dumped to disk, so that more memory would be available for other components in following runs.

In summary, adaptive scheduling relies on dynamic buffer management to reduce posting fragments, while it needs one or more scans on source table which might require extra sequential I/O cost. However, because sequential I/O rate is several times greater than random I/O rate, thus adaptive scheduling intends to balance sequential I/O and random I/O thus to achieve shorter overall construction time.

Analytical Build Different from adaptive scheduling that dynamically plans insertion runs, analytical scheduling decides insertion plans before the formal construction is started. Because posting lists of keys are not even lengths, hence dynamic scheduling may not use the best of available memory allowance but tend to request more sequential runs than it really needs. On the other hand, analytical scheduling performs a previewed scanning on source table to obtain statistics on the number of keys and the number of tuples that associated to keys, and then it estimates runtime costs of insertion runs by dynamic programming. By collecting statistical knowledge of source data, analytical scheduling could produce superior insertion plans than adaptive scheduling by better balancing sequential I/O and random I/O.

Previewed Scanning A previewed scanning on source table is performed beforehand to obtain knowledge about indexed keys, an algorithm *BuildPreview* for the process is given in Figure 5.20. The procedure produces a semi-finished KeyEntryList (i.e. SemiKeyEntryList) and a GroupEntryList. Different from a normal insertion phase, it only accumulates for each key the total tuple count and group count (see lines 14 to 21), but it would not produce semi-finished

postings for building posting lists; in addition, statistic of primary group is collected as well (see lines 7 to 13). Note that intermediate flushing is not required usually, because it is possible to hold all accumulated results in main memory. At the end of the process, the semi-finished key entries are sorted by the tuple counts of keys. Both group entry list and semi-finished key entry list are dumped to disk in the end.

Algorithm: BuildPreview
Input: Table
Output: GroupEntryList, sorted SemiKeyEntryList

```
1  begin
2    reset LastGroupID, GroupLength = 0;
3    Tuples = Read(Table);
4    while Tuples is not empty
5      foreach tuple in Tuples
6        extract Key and GroupID;
7        if LastGroupID is equal to GroupID
8          GroupLength++;
9        else
10         GroupEntry = new GroupEntry(GroupID, GroupLength);
11         GroupEntryList->Add(GroupEntry);
12         GroupLength = 1;
13       endif
14       if KeyLookup does not contain Key
15         KeyEntry = new KeyEntry(Key, NULL, NULL);
16         KeyLookup->Add(Key, KeyEntry);
17       endif
18       KeyLookup[Key]->KeyEntry->TupleCount++;
19       if KeyLookup[Key]->KeyEntry->GroupCount has not been updated
20         KeyLookup[Key]->KeyEntry->GroupCount++;
21       endif
22     endforeach
23     Tuples = ReadNext(Table);
24   endwhile
25   Write(GroupEntryList);
26   foreach KeyEntry in KeyLookup
27     SemiKeyEntryList->Add(KeyEntry);
28   endforeach
29   SemiKeyEntryList->Sort();  /* Sort by KeyEntry->TupleCount */
30   Write(SemiKeyEntryList);
31 end
```

Figure 5.20: Build preview

Let n be the number of source tuples, m be the number of keys, the complexity of accumulation is $O(n)$; if *Quicksort* [Hoare, 1961, Knuth, 1973] is applied for sorting semi-finished key entry list, then the average complexity of producing the list is $O(m + m \cdot \log m) = O(m \cdot \log m)$, and the total computational complexity of algorithm *BuildPreview* is $O(n + m \cdot \log m)$.

Algorithm: AnalyticalSchedule
Input: Sorted SemiKeyEntryList
Output: UpperBounds

```
1  begin
2    UpperBounds = new Array(), bound = 0;
3    reset Cost, LastCost and Variance;
4    RemainedTuples = SemiKeyEntryList->TupleCount, CurrentTuples = 0;
5    foreach KeyEntry in SemiKeyEntryList
6      CurrentTuples = CurrentTuples + KeyEntry->TupleCount;
7      RemainedTuples = RemainedTuples - KeyEntry->TupleCount;
8      Cost = CurrentCost(CurrentTuples) + RemainedCost(RemainedTuples);
9      Variance = ComputeVariance(Cost - LastCost);
10     if Cost is less than LastCost or Variance is less than threshold
11       bound++;
12       LastCost = Cost;
13     else
14       UpperBounds->Add(bound);
15       CurrentTuples = 0, reset Cost and LastCost;
16     endif
17   endforeach
18 end
```

Figure 5.21: Analytical scheduling

Analytical Scheduling and Cost Model Once a preview of source data is built, then insertion schedules could be generated based on the preview. In order to apply dynamic programming for computing schedules, a cost model is needed for estimating costs of planned runs. As it is given in formula 5.15, the estimated cost of RIX construction is indicated by I/O time, where the total I/O time T_{total} is summation of current I/O time $T_{current}$ and remained I/O time $T_{remained}$. Current I/O time is estimated by giving certain number of keys and tuples in a run, whereas remained I/O time is estimated by remained number of keys and tuples.

$$T_{total} = T_{current} + T_{remained} \tag{5.15}$$

Rather than using exact sizes of posting fragments (which is impossible to obtained) for computing, we use average sizes of posting TID fragments and posting inverted group fragments instead. Let S^{avg} denote average transfer size, then the average read transfer size of TID fragments and group fragments are given in formula 5.16, where n_k is the number of keys to be processed in a run, and other variables are as defined in Table 5.3.

$$S_{pt}^{avg} = \frac{s_{pt} \cdot \sum_{i=1}^{n_k} n_{t,k_i}}{n_k},\ S_{pm}^{avg} = \frac{s_{pm} \cdot \sum_{i=1}^{n_k} n_{g,k_i}}{n_k} \tag{5.16}$$

The I/O costs of current run and remained runs are estimated as formula 5.12, in which T_{fetch}, T_{tmp} and T_{entry} are computed as formulas 5.8, 5.9 and 5.11 respectively, whereas $T_{posting}$ is calculated by formula 5.17.

$$T_{posting} = t'_r \cdot \left\lceil \frac{S_{pt}^{avg}}{B} \right\rceil + t'_w \cdot \left\lceil \frac{s_{pt} \cdot \sum_{i=1}^{n_k} n_{t,k_i}}{B} \right\rceil + t''_r \cdot \left\lceil \frac{S_{pm}^{avg}}{B} \right\rceil + t''_w \cdot \left\lceil \frac{s_{pm} \cdot \sum_{i=1}^{n_k} n_{g,k_i}}{B} \right\rceil \quad (5.17)$$

A dynamic programming algorithm *AnalyticalSchedule* for scheduling insertion runs is shown in Figure 5.21. Let n be the total number of keys, so the complexity of the algorithm is $O(n)$.

Analytical Build Algorithm A building algorithm *AnalyticalBuild* that applies analytical scheduling is given in Figure 5.22.

Algorithm: AnalyticalBuild
Input: Table
Output: RixStd

```
1  begin
2    SemiKeyEntryList = BuildPreview(Table);
3    UpperBounds = AnalyticalSchedule(SemiKeyEntryList);
4    Iterator = 0;
5    foreach bound in UpperBounds
6      while Iterator is less than bound
7        Key = SemiKeyEntryList[Iterator]->Key;
8        KeyEntry = SemiKeyEntryList[Iterator]->KeyEntry;
9        KeyLookup->Add(Key, KeyEntry);
10       Iterator++;
11     endwhile
12     Tuples = Read(Table);
13     while Tuples is not empty
14       foreach tuple in Tuples
15         if KeyLookup contains tuple->Key
16           ValidTuples->Add(tuple);
17         endif
18       endforeach
19       Accumulate(ValidTuples);
20       Tuples = ReadNext(Table);
21     endwhile
22     MergePostingLists();
23     Write(KeyEntryList), release memory and reset KeyLookup;
24     delete temporary file;
25   endforeach
26   BuildExternalLookup(KeyEntryList);
27   BuildExternalLookup(GroupEntryList);
28 end
```

Figure 5.22: Constructing RIX with analytical scheduling

A seen drawback of the analytical building algorithm is that it introduces extra overheads in order to obtain statistics of source data and producing building schedules. However, experiments show that these additional overheads are worthy because overall building time will be considerably shorten comparing to naive and adaptive building algorithms. Note that an important gain from previewed statistics is that short posting lists could be directly written to RIX file since we have known those lists have been completed and do not need to be merged. In other words, a lot of I/O spent on temporary file are therefore saved.

Finally, an interesting question might be asked is that whether building schedules could be based on approximate statistics rather than based on thorough statistics? This is a topic worth studying but we will leave it to future work and do not discuss further in this thesis.

5.2.6 Retrieval Procedures

In this subsection, we address the retrieval procedures that are supported by RIX. In general, users may employ one of the implementations of RIX, i.e. RixLite, RixStd or RixExt, to support physical operators or operations for query processing. Specifically, a RIX instance could be used for the following purposes:

- To support conventional relational operations of database systems such as selection and indexed join
- To support special operations of IR+DB systems such as probability estimation and probability aggregation
- To support data accessing methods deployed by top-*k* algorithms such as sorted access and random access

Though varied RIX instances are slightly different with regards to the data structure of posting lists, nevertheless, they can be retrieved by similar searching and fetching methods. The rest of the section introduces the common methods for retrieving the instances of RIX, and describes in short how RIX can be used to support physical operators and operations.

5.2.6.1 Accessing Methods for Search and Fetch

To access the ADTs addressed in Section 5.2.3, there are three types of methods that can be used for search (i.e. looking-up) or fetch (i.e. retrieving) from RIX instances. In particular, *Contain* methods are applied for looking up existing data entries, while *Get* methods and *Next* methods are

employed for retrieving posting data. Implementations of a certain type of methods can access to RIX in a similar behaviour.

Contain Methods In general, a *Contain* method may perform two actions. First, it attempts to look for a data entry from a RIX instance while given an accessing key, and then it acknowledges the calling process whether a look-up is successful or not. Second, it reads a found data entry into memory so that to prepare for forthcoming fetch. Note that a read of data entry would be always performed after a look-up, and it would be only issued if the look-up is successful. With regards to associated ADTs, *Contain* methods are related to search facilities of RIX, which at the moment are hash tables (i.e. Hybrid Hash Lookup). Therefore, respective *Contain* methods are implemented for looking up indexed terms (keys) and indexed document IDs (GIDs).

Get Methods *Get* methods belong to a common type of accessing method for retrieving indexed data items and statistics, and there are implementations to access information from all kinds of ADTs including data entries and posting lists. After a data entry has been found (by a certain Contain method), then *Get* methods can be used (with a specified accessing key) not only to retrieve a list of indexed items such as a list of TID units or a list of inverted group units, but also to retrieve single statistics such as tuples count (TC), or groups count (GC), or group length (GL), etc..

While indexing documents (in IR applications), because the posting lists of RIX could be extremely long, hence the *Get* methods for retrieving posting lists (including TIDs and inverted groups) can be set to return a limited portion of an entire list, for instance, the first one hundred items of a list; while a stopping position is recorded for further fetching, which would be taken over by another type of fetching methods, i.e. *Next* methods.

Next Methods A *Next* method can be called to fetch more data items from a posting list if a *Get* counterpart was deployed to retrieve the first part of the same list. Similarly to *Get* methods, *Next* methods return a portion of a posting list with a pre-set number of items to be retrieved per fetch; at the mean time, the stopping position would be updated at the end of a fetch. Therefore, to retrieve an entire posting list, a calling process just needs to repeatedly deploy a *Next* method until no more data are returned.

5.2.6.2 Supporting Physical Operators and Operations

Here let us discuss what physical operators and operations can be supported by RIX in practice. Giving the three types of accessing methods as aforementioned, we relate specific methods to

the ADTs containing in RIX and introduce three situations where certain RIX instances can be applied, specifically, for supporting conventional relational operators, composed and special operations, and top-k related operations. To demonstrate, some available implementations of fetching methods are given in Table 5.5.

Get Methods	*Next Methods*	*Retrieved Data*	*Corresponding IR Concepts*
GetPTID	*NextPTID*	posting tuple-IDs	N/A
GetPTC	N/A	posting tuples count	within-collection *tf* of term
GetPGC	N/A	posting groups count	*df* of term
GetPIGTC	*NextPIGTC*	posting in-group tuples count	within-document *tf* of term
GetAvgPIGTC	N/A	average posting IGTC	average *tf*
GetMinPIGTC	N/A	minimum posting IGTC	minimum *tf*
GetMaxPIGTC	N/A	maximum posting IGTC	maximum *tf*
GetGL	N/A	group length	document length
GetAvgGL	N/A	average group length	average document length
GetMinGL	N/A	minimum group length	minimum document length
GetMaxGL	N/A	maximum group length	maximum document length
GetTC	N/A	global tuples count	total number of terms (i.e. N_t)
GetKC	N/A	global keys count	total number of distinct terms
GetGC	N/A	global groups count	total number of documents (i.e. N_d)

Table 5.5: Specific accessing methods of RIX

For Relational Operators First of all, all RIX instances including RixLite, RixStd and RixExt support TID-index-based relational operators such as selection and indexed join. Given a primary accessing key, e.g. an indexed term, *GetPTID* and *NextPTID* methods can be used to retrieve a list of corresponding tuple Identifiers, which give direct access to the tuples in a table where the term can be found.

For Composed IR+DB Operations Secondly, all RIX instances support composed operations for probability estimation and probability aggregation. In an IR+DB system, special inverted-index-based operations can be implemented as general purposed probability estimators and aggregators (e.g. see Section 4.3), which produce weighted tuples based on pre-defined scoring functions. Most of the accessing methods in Table 5.5 retrieve statistics associated to certain IR concepts, which are required for modelling IR ranking models.

Moreover, the accessing methods for statistics can be related to the scoring driven optimization technique that bases on scoring expressions (SCX) (see Chapter 3, also see e.g. Sec-

tion 3.4.4). An IR+DB query execution engine could exploit RIX to support queries formulating scoring functions such as those given in Table 3.10.

For Top-k Related Operations Thirdly, all RIX instances support top-k algorithms based on no random access (NRA). As posting inverted group lists can be sorted by in-group tuples count (IGTC) or document IDs (GID), the *GetPIGTC* and *NextPIGTC* methods together provide a sequential (sorted) access functionality to posting lists, which is the accessing mode required by NRA-style algorithms in a top-k incorporated pipeline (see Section 4.3).

On the other hand, with respect to random access to posting inverted group lists, current RIX instances (i.e. RixLite, RixStd and RixExt) do not provide accessing support for such purpose. However, it is possible to extend the current implementations so that to allow random accesses for posting data, while related studies will be left to future work.

5.2.7 Update Procedure

In general, index update is an important aspect for information management systems. Especially for databases, efficient methods for ad hoc update and incremental update are always popular research topics. However, data update is a relative expensive manipulation comparing to data insertion, therefore, when an index is out of date, many practical IR systems would rather reconstructing a new index than updating an existing one.

Because RIX is proposed for supporting text retrieval for applications involving both text and structured data, hence updating inverted group lists of RIX would be as expensive as conventional inverted document lists in traditional IR systems. Therefore, by following similar consideration of most IR systems, RIX instances allow very few update functionality. In RIX, ad hoc update would not be provided, while incremental update is allowed if indexed table grows as long as the original data in the table are not changed. In contrast, if tuples in a table are changed, then a corresponding RIX has to be rebuilt in order to keep the index up-to-date.

5.3 Experiments and Results

In this section, we present experiments that evaluate the performance of RIX with regards to index construction. While with respect to retrieval performance, which has been demonstrated in Section 3.5 of Chapter 3 where we investigated the runtime performances of processing PRA queries while utilising scoring-driven optimization and RIX.

5.3.1 Specifications and Setup

Above all, the specifications and experimental setup are given as follows.

Systems An IR+DB prototype Birdie A was used as the testing bed, in which the RIX instances have been implemented. The configurations of Birdie are:

- page size: 4 KB
- memory allowance for buffering data: 40 MB
- total memory allowance for indexing process: 512 MB (including the 40 MB for buffering)
- memory allowance for retrieval process: 512 MB

To measure the performances of index construction and retrieval, we used a standalone PC which specifications are as follows: Dell XPS M1330 Laptop. Intel[10](R) Core[11](TM)2 Duo CPU T6400 at frequency 2.00GHz, 3.00 GB of RAM at frequency 1.20 GHz. Windows XP Professional OS, version 2002, Service Pack 3. The capacity of disk partition where indexes are stored is 100 GB, and the page size is 4KB.

Test Collection Similarly to previous chapters, TREC-3[12] was used as the testing collection, and the original collection was pre-processed and loaded into a relational table in Birdie. The schema of the indexed table is as follow.

```
CREATE TABLE trec3 (term VARCHAR, docid VARCHAR);
```

Readers may refer to previous chapters and sections (e.g. see Section 3.5.1) for the specifications of the TREC collection.

Setup To create RIX indexes, Birdie employs a SQL-style data definition language (DDL), which can be used as follows:

```
CREATE INDEX trec3_rxl RXL ON trec3(term) GIVEN EVIDENCE (docid);
CREATE INDEX trec3_rxs RXS ON trec3(term) GIVEN EVIDENCE (docid);
```

For instance, the first line defines an RIX index named “trec3_rxl”, while “RXL” declares the index type to be RixLite; secondly, the RixLite index is built on table “trec3” on the attribute named “term”, which is the primary indexed key; and thirdly, the “GIVEN EVIDENCE” clause defines an “evidence” attribute for probability estimation, which is used as a secondary key for

[10]Intel is a registered trademark of Intel Corporation.
[11]Core is a trademark of Intel Corporation.
[12]http://trec.nist.gov/

grouping operation, and its values are used as the identifiers of the posting inverted groups, while in this case, the attribute "docid" is given as the evidence. In addition, the "RXS" keyword can be used for the declaration of building a RixStd index.

Since RixExt can be viewed as a mixture of RixLite and RixStd, therefore we skipped the investigations of RixExt but only measured RixLite and RixStd in the experiments.

The data size of the indexed table and the indexes are as follows: table size 4.02 GB (i.e. 4 216 908 bytes); the total number of tuples in the table is 202 254 542, the total number of keys (i.e. distinct terms) is 715 649, and the total number of groups (i.e. documents) is 741 647; the index size of RixLite is 3.97 GB (i.e. 4 168 713 bytes), while the index size of RixStd is 4.83 GB (i.e. 5 066 937 bytes).

5.3.2 Methodology

The experiments focused on investigating the efficiency of RIX construction and retrieval. First, we measured the performances of constructing RixLite and RixStd. Second, we conducted different fetching tasks in different conditions to evaluate the retrieval performances.

Despite efficiency is the main interest for us, we also measured the retrieval effectiveness when exploiting different indexes, so that to validate the correctness of the constructing and fetching algorithms.

Construction Performance To investigate the index construction performance, we conducted the *adaptive build* and the *analytical build* algorithms for constructing RixLite and RixStd, and the experimental methods are given as the follows:

1. *Adaptive Build*: Investigates the construction of RixLite and RixStd on a PC while applying adaptive building schedule.

   The three phases of adaptive build are insertion, merging and finalising. By applying adaptive scheduling, it adjusts the number of keys (e.g. distinct terms) to be accepted in the insertion phase, where the maximum limit of accepted keys is called *cap* or *keys cap*. Once the number of inserted keys (i.e. seen keys) reaches the cap, then indexer would only process the tuples with seen keys and skip those with unseen keys. In addition, if skips happen in a scanning run, the position in scanned table where the first skip occurs is called *skip start point* and would be recorded. Merging is performed after insertion. And then

indexer rewind table scan to the skip start point and restart insertion if skips happened in the previous scanning run, otherwise it finalises the construction.

This experiment demonstrates how keys cap is adjusted in scanning runs and the time consumption of different phases, where initial cap is set to 65 536 (i.e. 2^{16}) keys. and the same criteria are used to investigate the performances of building RixLite and RixStd.

2. *Analytical Build*: Investigates the construction of RixLite and RixStd on a PC while applying analytical building schedule.

   There are four phases during indexing while applying analytical build, which are analysis, insertion, merging and finalising. Different from adaptive build, in analytical build the keys cap would be estimated and pre-computed during analysis, and re-scans always restart from the beginning of indexed table. Apart from that, the other settings of analytical build are the same to adaptive build.

   This experiment demonstrates an analytical building schedule; and similarly, the time consumption of different phases of indexing for RixLite and RixStd.

3. *Time Measurement*: The indexing time are measured based on several constructing phases, and the overall indexing time are obtained and investigated.

   Two types of time measurements are considered, which are the elapsed time (denoted by *Elapsed*) and the process time (denoted by *Process*). The former is the observed indexing time, whereas the latter is the actual CPU time used by indexing process; the difference between the elapsed time and the process time indicates the amount of waiting time for I/O operations.

Retrieval Performance To investigate the index retrieval performance, we considered different fetching manners such as *sequential access* and *random access* while retrieving from different components of RixLite and RixStd. In addition, because the indexes would be integrated with query engine, hence we also studied the performances of different indexes while they are applied in a query engine.

1. *By Sequential Access*: Investigates the performances of retrieving posting lists such as TID List and Inverted Group List.

The TID lists and the Inverted Group list are the two main types of posting lists in RIX instances, which are retrieved through sequential access. The experiment investigates the fetching rates of different posting lists in different RIX instances, where the retrieval time were recorded incrementally for per 10 000 tuples (posting items) were retrieved. Considering the caching effect of operating system, we always performed the same run twice, so that the first run represents the performance from a "cold start", whereas the second run represents the performance from a "hot start". The performances of RixLite and RixStd were evaluated respectively.

2. *By Random Access*: Investigates the performances of retrieving entries such as Key Entry and Group Entry.

   The Key Entry and the Group Entry are based on external hash table so that can be retrieved by random access. The experiment investigates the lookup rates of different data entries in different RIX instances, where the lookup time were recorded incrementally for per 1 000 different keys (for Group Entries are group IDs) were issued and corresponding entries were retrieved. Similarly to the experiments of sequential access, the same retrieval run were always conducted twice with a "cold start" and a "hot start". The performances of RixLite and RixStd were evaluated respectively.

3. *By Integrating with Query Engine*: Investigates the performances while integrated with query engine for running different retrieval models.

   In practical, the indexes are utilised by the query engine where both sequential access and random access to the indexes would be applied. The experiment investigates the overall retrieval time when utilising different RIX instances, where the retrieval time for different retrieval models were recorded. The performances of RixLite and RixStd were evaluated respectively.

5.3.3 Results

5.3.3.1 Construction Performance

Here we present the experimental results of construction performances while building RixLite and RixStd by *adaptive build* and *analytical build* respectively.

Adaptive Build Respectively, the adaptive build result for RixLite is shown in Table 5.6, and the result for RixStd is given in Table 5.7, while the overall performances of RixLite and RixStd are compared in Table 5.8.

Scan Run	*Adjustment of Keys Cap*			*Final Keys Cap*	*Skip Start Point*
	Adjusted Cap	*Time Stamp*	*Scanned*		
1	40 041	00:00:34	1.36%	40 041	2 805 690
2	190 972	02:18:52	5.8%	174 231	30 962 032
	174 231	02:20:06	15.23%		
3	155 169	02:32:08	20.33%	197 674	94 917 350
	193 735	02:33:55	37.12%		
	196 226	02:35:04	46.64%		
	197 087	02:35:05	46.78%		
	197 674	02:35:06	46.93%		
4	138 985	02:41:54	54.53%	246 253	196 106 487
	206 021	02:43:12	67.55%		
	242 743	02:45:18	88.08%		
	246 810	02:46:10	96.05%		
	246 253	02:46:16	96.96%		
5	–	–	–	65 536	–

(a) Adaptive Scheduling for RixLite

Scan Run	*Insertion Time Stamp and Span (sec)*				*Merging Time Stamp and Span (sec)*			
	Elapsed		*Process*		*Elapsed*		*Process*	
	Stamp	*Span*	*Stamp*	*Span*	*Stamp*	*Span*	*Stamp*	*Span*
1	00:44:37	2 677	00:43:47	2 627	02:18:23	5 626	00:56:57	790
2	02:30:34	731	01:08:17	680	02:31:34	60	01:08:56	39
3	02:40:50	556	01:17:37	521	02:41:06	16	01:17:53	16
4	02:46:36	330	01:23:21	328	02:46:50	14	01:23:35	14
5	02:47:07	17	01:23:52	17	02:47:08	1	01:23:53	1
Total (sec)	–	4 311	–	4 173	–	5 717	–	860
Wait (sec)	138				4 857			

(b) Timing of Insertion and Merging for RixLite

Table 5.6: Scheduling and timing of RixLite construction with adaptive build

First of all, let us explain the adaptive scheduling for RixLite. In Table 5.6a, where the procedures of adaptive scheduling are shown. At the beginning of each scanning run, the keys cap is reset to initial value 65 536, while the cap would be adjusted during insertions. For instance, during the first scanning run, the cap was adjusted to 40 041 after 34 seconds when the indexing was started, and the adaptation happened while the process had scanned 1.36% of the indexed table. In addition, the skip start point records the first skipped tuple, which is at position 2 805 690 (this is a sequential position, i.e. the 2 805 690th tuple counted from 0) in the table. Moreover, indexer adjusted the cap only once in the first run, whereas it adjusted the cap twice in the second run, and five times in the third run, and so on so forth; while in the last run, i.e. the fifth run, there

is not a adjustment because scanning was completed before any adjustment might be required, hence the cap remained its initial value 65 536.

Next, let us have a look at the timing of insertion and merging for RixLite, which is given in Table 5.6b. As it is shown, each scan run consist f an insertion phase and a merging phase, and the timings of both phases in all runs were recorded, where a *time stamp* (or just *stamp*) is the time clocked, whereas a *time span* (or just *span*) is the duration. Since our actual interest is duration, therefore when we say "time" in our following discussions, we usually mean "time span" unless further clarify. For instance in the first run, the elapsed time of insertion is 2 677 seconds (i.e. 44 minutes and 37 seconds, or 44m 37s in short) and the process time is 2 627 seconds (43m 47s), whereas the elapsed time of merging is 5 626 seconds (1h 33m 46s) and the process time is 790 seconds (13m 10s). Similarly, the insertion and merging timing of subsequent runs are shown. Moreover, the total duration of insertion and merging are obtained by summing up respective timings in every runs, in which for insertion the total elapsed time is 4 311 seconds (1h 11m 51s) and the process time is 4 173 seconds (1h 9m 33s), whereas for merging the total elapsed time is 5 717 seconds (1h 35m 17s) and the process time is 860 seconds (14m 20s).

What can be found from Table 5.6 include:

- Though there is a very small portion of keys to be inserted in the first run than the subsequent runs, but those keys associate to very large amount of tuples; while reflecting in timing, the first run spent more than 80% of the overall indexing time. On the other hand, this observation also reflects the characteristics of text documents, i.e. a relatively small amount of common terms occurs in much larger amount of documents.

- With respect to the difference between elapsed time and process time, it can be seen that the gap is dramatic in the merging phase of the first run (4 857 sec, i.e. 1h 20m 57s), whereas the gaps are insignificant for all insertion sub-phases and the subsequent merging sub-phases. As aforementioned, the gap between the two timings represents the waiting duration for I/O operations, and the observation indicates a large amount of random disk I/O could be applied in the merging phase of the first scanning run.

Furthermore, we address the result of RixStd in Table 5.7. Overall, the procedure is similar to RixLite, though there are two casual events were observed.

First, it can be seen that the keys cap was adjusted to 35 165 during the first scanning run, which is a few thousand keys less than RixLite, but a skip start point was obtained exactly the

same to the one in RixLite, which seems to be impossible. Because each inverted group unit in RixStd contains one more field than a counterpart in RixLite (see Section 5.2.3.1), hence the adaptation function tends to tune the indexer to accept less keys so that to preserve memory allowance for posting lists. However, situations such as here can still happen because the cap is adjusted after inserting a group of tuples, while in our experiments a group corresponds to a document. As a result, although a smaller final cap was issued, but the actual accepted keys were the same to those in RixLite.

Scan Run	*Adjustment of Keys Cap*			*Final Keys Cap*	*Skip Start Point*
	Adjusted Cap	*Time Stamp*	*Scanned*		
1	35 165	00:00:35	1.36%	35 165	2 805 690
2	200 203 159 140	02:39:26 02:40:57	5.76% 15.92%	159 140	32 219 686
3	108 086 191 043 197 973 196 598	02:54:53 02:55:50 02:58:16 02:58:23	21.42% 29.04% 47.01% 47.79%	196 598	96 667 477
4	29 963	03:05:45	50.93%	29 963	103 012 546
5	133 841 200 979 233 863	03:11:16 03:12:49 03:14:44	60.82% 76.85% 95.46%	233 863	–

(a) Adaptive Scheduling for RixStd

Scan Run	*Insertion Time Stamp and Span (sec)*				*Merging Time Stamp and Span (sec)*			
	Elapsed		*Process*		*Elapsed*		*Process*	
	Stamp	*Span*	*Stamp*	*Span*	*Stamp*	*Span*	*Stamp*	*Span*
1	00:46:37	2 797	00:46:12	2 772	02:38:46	6 729	01:01:00	888
2	02:53:05	859	01:12:34	694	02:54:10	65	01:13:17	43
3	03:05:03	653	01:22:00	523	03:05:21	18	01:22:18	18
4	03:10:17	296	01:27:00	282	03:10:19	2	01:27:02	2
5	03:15:15	296	01:31:58	296	03:15:28	13	01:32:10	12
Total (sec)	–	4 901	–	4 567	–	6 827	–	963
Wait (sec)	334				5 864			

(b) Timing of Insertion and Merging for RixStd

Table 5.7: Scheduling and timing of RixStd construction with adaptive build

Second, an occasional situation was occurred in the fourth run, where the final cap is unusually small so that the first skip happened much earlier than expected. The indexing log indicates the actual memory usage does not match the indexer's internal record, where the actual usage was shown to be exceeding the allowance, but the indexer's record shows it did not used much memory. Considering a flushing operation had performed not long before, a possible explanation is that there was a lag somehow happened for memory recollection procedure[13], so that the

[13]In short, the memory management in C# depends on a mechanism called Garbage Collector (GC),

released memory had not been recollected in time. As a result, the indexer was forced to start skipping earlier than usual. Nevertheless, scheduling was back to normal in the fifth run, though a few more minutes had to spend to process the tuples which could have been done in the fourth run. Apart from that, other observations are similar to the constructing procedure of RixLite.

Similarly, merging posting lists in the first run for adaptive build RixStd appear to be suffering from a significant delay led by awaiting disk I/O.

Phase	*RixLite Build Time (sec)*		*RixStd Build Time (sec)*	
	Elapsed	*Process*	*Elapsed*	*Process*
Insertion	4 311	4 173	4 901	4 567
Merging	5 717	860	6 827	963
Finalising	37	29	37	29
Total (sec)	10 065	5 062	11 765	5 559
(hh:mm:ss)	02:47:45	01:24:22	03:16:05	01:32:39
Wait (sec)	5 003		6 206	
(hh:mm:ss)	01:23:23		01:43:26	

Table 5.8: Building time with adaptive build, RixLite vs. RixStd

In addition, the overall comparison for the indexing procedures of RixLite and RixStd using adaptive build is illustrated in Table 5.8. From the table we can see that during indexing over 99.5% time is spent on constructing posting data, while less than 0.5% time is used for constructing searching facility. Awaiting I/O is a considerable problem affecting indexing efficiency, for instance, it took the indexing process for RixLite more than one hour and twenty minutes in waiting, while for RixStd it took even twenty minutes more than RixLite.

Analytical Build Next, the results of analytical build are given in the following tables, where Table 5.9 demonstrates the results for building RixLite, while Table 5.10 illustrates the result for constructing RixStd, and the overall performances of both indexes are given in Table 5.11.

Let us first discuss the result of RixLite. As stated, indexer applying analytical build deploys a previewed scan on the indexed table, and then analyses the indexing workload based on a knapsack-style algorithm collaborating with a predefined cost model. While the scheduling result came out of analysis is given in Table 5.9a, in which shows the table scanning for insertion had been scheduled into three runs. For instance, the first run would process 3 421 keys which associate to over 150 million tuples, while the second run involves 82 873 keys, and so on. The analysing phase spent 13 minutes and 51 seconds, and then the insertion phase was started.

which collects released memory for reallocation. Similar mechanism is also utilised in other programming language such as Java.

Scan Run	*Analytical Schedule*	
	Keys Cap	*Tuples Scheduled*
1	3 421	150 552 295
2	82 873	49 156 770
3	629 355	2 545 477
Total	715 649	202 254 542

(a) Analytical Scheduling for RixLite

Scan Run	*Insertion Time Stamp and Span (sec)*				*Merging Time Stamp and Span (sec)*			
	Elapsed		*Process*		*Elapsed*		*Process*	
	Stamp	*Span*	*Stamp*	*Span*	*Stamp*	*Span*	*Stamp*	*Span*
1	00:44:46 (starts at 00:13:51)	1 855	00:42:55 (starts at 00:13:02)	1 793	01:30:40	2 754	00:52:14	559
2	01:53:45	1 385	01:14:28	1 334	02:14:03	1 218	01:18:14	226
3	02:27:25	802	01:30:50	756	02:27:31	6	01:30:55	5
Total (sec)	–	4 042	–	3 883	–	3 978	–	790
Wait (sec)	159				3 188			

(b) Timing of Insertion and Merging for RixLite

Table 5.9: Scheduling and timing of RixLite construction with analytical build

Furthermore, the processing time of insertion and merging are illustrated in Table 5.9b. For instance, the elapsed time of the first run insertion is 1 855 seconds (i.e. 30m 55s), and the elapsed time of the first run merging is 2 754 seconds (45m 54s). To sum the timing of all sub-phases up, we obtained the total elapsed and process time of insertion are 4 042 seconds (1h 7m 22s) and 3 883 seconds (1h 4m 43s) respectively, while the total elapsed and process time of merging are 3 978 seconds (1h 6m 18s) and 790 seconds (13m 10s).

There are at least two findings can be seen from Table 5.9:

- Although analytical build spends some extra overhead on analysing phase, but it earns worthy pay-back that the overall performance can be dramatically improved.
- Comparing to adaptive scheduling, analytical scheduling improves the elapsed time of insertion slightly by shortening the time for about four and a half minutes (4 042 sec vs. 4 311 sec), but it improves the elapsed time of merging dramatically by reducing the time by 42 minutes (3 188 sec vs. 5 717 sec). This can be achieved largely because analytical scheduling breaks down the waiting time during merging, for example in this case, 3 188 seconds for analytical versus 4 857 seconds for adaptive, which leads to a difference of over 27 minutes.

Now let us move on to the result of RixStd, which is addressed in Table 5.10. As expected,

Scan Run	*Analytical Schedule*	
	Keys Cap	*Tuples Scheduled*
1	2 564	140 274 631
2	58 146	58 235 571
3	654 939	3 744 340
Total	715 649	202 254 542

(a) Analytical Scheduling for RixStd

Scan Run	*Insertion Time Stamp and Span (sec)*				*Merging Time Stamp and Span (sec)*			
	Elapsed		*Process*		*Elapsed*		*Process*	
	Stamp	*Span*	*Stamp*	*Span*	*Stamp*	*Span*	*Stamp*	*Span*
1	00:43:58 (starts at 00:13:34)	1 824	00:42:09 (starts at 00:12:53)	1 756	01:20:00	2 162	00:51:29	560
2	01:45:00	1 500	01:15:43	1 454	02:14:35	1 775	01:20:34	291
3	02:28:30	835	01:33:40	786	02:29:02	32	01:34:06	26
Total (sec)	–	4 159	–	3 996	–	3 969	–	877
Wait (sec)	163				3 092			

(b) Timing of Insertion and Merging for RixStd

Table 5.10: Scheduling and timing of RixStd construction with analytical build

because the data structure of RixStd is different from RixLite, we obtained a slightly different scanning schedule for insertion as it is shown in Table 5.10a. Although there are still three allotted runs, but the keys caps had been even restricted for the first two runs comparing to the schedule for RixLite.

Moreover, the result of insertion and merging phases is given in Table 5.10b. Amazingly, the RixStd indexer handled the two phases very well that it can even match the counterpart for RixLite. Because the index size of RixStd is larger than RixLite, hence it is reasonable that the indexer would spend a little bit more time on insertion, for example, in this case it is 4 159 seconds (1h 9m 19s) for RixStd versus 4 042 seconds (1h 7m 22s) for RixLite. However, what is really impressive is that the elapsed time of merging for RixStd is as good as RixLite (actually, it is even 9 seconds better than RixLite), which means the analytical scheduling method works effectively to minimise the waiting duration of merging process.

Finally, the overall performances of constructing RixLite and RixStd with analytical scheduling is shown in Table 5.11. In which we can see that the total indexing time (i.e. elapsed time) of the two types of RIX are almost the same and both duration are less than two and a half hours. Bearing in mind the fact that the index size of RixStd is about 20% larger than RixLite, which reflects that reducing the cost of random accesses is an important principle for improving the construction efficiency of indexing. While comparing to adaptive scheduling, analytical build

Phase	*RixLite Build Time (sec)*		*RixStd Build Time (sec)*	
	Elapsed	*Process*	*Elapsed*	*Process*
Analysis	831	782	814	773
Insertion	4 042	3 883	4 159	3 996
Merging	3 978	790	3 969	877
Finalising	30	27	35	29
Total (sec)	8 881	5 482	8 977	5 675
(hh:mm:ss)	02:28:01	01:31:22	02:29:37	01:34:35
Wait (sec)	3 399		3 302	
(hh:mm:ss)	00:56:39		00:55:02	

Table 5.11: Building time with analytical build, RixLite vs. RixStd

appears to be superior to an adaptive counterpart in minimising the overhead of random accesses.

5.3.3.2 Retrieval Performance

Here we present the experimental results of retrieval performances while accessing RixLite and RixStd by *sequential access* and *random access*, as well as the runtime performances of query engine when utilising different RIX instances for processing retrieval models.

By Sequential Access In this set of experiments, the runs were performed by selecting two query terms, which are "be" and "from"[14], where the term "be" has the longest posting TID list containing 1 726 429 TID units, (there are 416 782 units in its Inverted Group list), and the other term "from" has the longest posting Inverted Group list including 442 777 units, (there are 1 369 818 units in the term's TID list). In the experiment, tuples (i.e. posting units) were retrieved from the list incrementally, and the retrieval time (including the incremental time and the accumulative time) of every 10 000 tuples were recorded. The results are presented in Table 5.12 and Table 5.13 respectively.

The results are organised according to the settings which can be easily found out from the headers. In particular, the header *inc* stands for *incremental time* and *total* means *accumulative time*. For instance, in Table 5.12, the retrieval time on RixLite for the first 10k (10 000) TID units are 31 milliseconds with a cold start and 15 milliseconds with a hot start. There are zeros in the table is because the elapsed time is too short to be recorded by a monitoring timer. Each table contains the first ten results out of one hundred. From Table 5.12 one can see that the retrieval from a posting TID list are very efficient.

Similarly, Table 5.13 demonstrates the first ten results of the retrieval from a posting Inverted Group list. The retrieval time with different settings show that the processes are very efficient on

[14]The stopwords had been deliberately reserved for experimental purposes such as these.

Number of Retrieved TID Units	*Retrieval Time (milliseconds)*							
	RixLite				*RixStd*			
	Cold Start		*Hot Start*		*Cold Start*		*Hot Start*	
	inc	*total*	*inc*	*total*	*inc*	*total*	*inc*	*total*
10k	31	31	15	15	15	15	0	0
20k	0	31	0	15	0	15	0	0
30k	0	31	0	15	15	31	15	15
40k	15	46	0	15	0	31	0	15
50k	0	46	15	31	0	31	0	15
60k	0	46	0	31	15	46	15	31
70k	15	62	0	31	0	46	0	31
80k	0	62	15	46	0	46	0	31
90k	0	62	0	46	15	62	0	31
100k	0	62	0	46	0	62	15	46

Table 5.12: Retrieval time of sequential accesses on posting TID lists of RixLite and RixStd

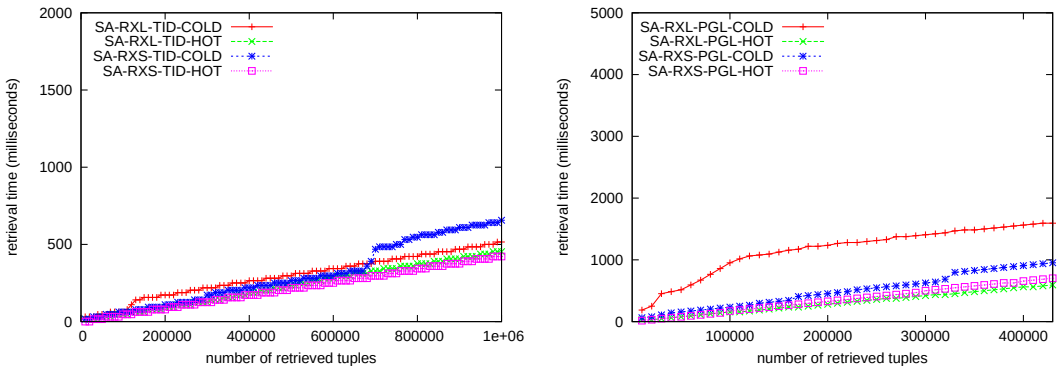
both RixLite and RixStd. However, to launch the retrieval from a cold start costs slightly more than a hot start at the beginning, which indicates the underlying caching function of operating system has shown positive effect to the retrieval.

Number of Retrieved Inverted Group Units	*Retrieval Time (milliseconds)*							
	RixLite				*RixStd*			
	Cold Start		*Hot Start*		*Cold Start*		*Hot Start*	
	inc	*total*	*inc*	*total*	*inc*	*total*	*inc*	*total*
10k	187	187	31	31	62	62	15	15
20k	62	250	0	31	15	78	15	31
30k	203	453	15	46	31	109	15	46
40k	31	484	15	62	31	140	15	62
50k	31	515	15	78	15	156	15	78
60k	78	593	15	93	15	171	15	93
70k	78	671	15	109	15	187	15	109
80k	93	765	15	125	15	203	15	125
90k	93	859	15	140	15	218	15	140
100k	93	953	15	156	15	234	31	171

Table 5.13: Retrieval time of sequential accesses on posting Inverted Group lists of RixLite and RixStd

In addition, the full results of retrieving posting lists are illustrated in Figure 5.23. The naming convention of the labels is $\langle AccessMode \rangle - \langle RixType \rangle - \langle Component \rangle - \langle LaunchMode \rangle$. For example, the label SA-RXL-TID-COLD stands for conducting sequential access on a TID list of RixLite by cold start. The following discussions comply to the same convention.

Overall, Figure 5.23a and Figure 5.23b show that sequential-access-based retrieval on the



(a) SA posting TID list

(b) SA posting Inverted Group list

Figure 5.23: Performance of sequential accesses for retrieving posting lists

posting lists of RixLite and RixStd has linear computational complexity, and the cost for retrieving a large quantity of posting data is very low. Therefore, both RixLite and RixStd can be competent for retrieval tasks that involve very long posting lists.

By Random Access Next, we investigate the performances of random access facilities in RIX indexes. In a RIX instance, Key Entries and Group Entries are stored in external hash tables so that can be retrieved by random accesses. Therefore, the experimental runs were to retrieve Key Entries and Group Entries by using random keys and group IDs.

In total, there are 715 649 distinct keys (terms) and 741 647 distinct group IDs (document IDs). Firstly, we measured the time for retrieving Key Entries, where the incremental time and accumulative time for every 1 000 entries were recorded. And then a similar experiment was repeated for retrieving Group Entries. The first ten results are demonstrated in Table 5.14 and Table 5.15, in which the retrieval performances of random accessing Key Entries and Group Entries were shown respectively.

The results confirmed that random accesses are expensive operations. For example, the total retrieval time for the first 1k (1 000) Key Entries on RixLite is 11 812 milliseconds (see Table 5.14), which is several orders of magnitude greater than sequential accesses. On the other hand, the caching mechanism of operating system could play a significant role and dramatically improve the lookup performances. For instance, the retrieval time for the first 1k Key Entries on RixLite from a hot start is only 31 milliseconds.

To compare the results in Table 5.14 and Table 5.15, one can see that the incremental retrieval time decrease along with more data entries were retrieved, while the incremental time of retrieving Group Entries always decrease faster than the counterpart of retrieving Key Entries. This

Number of Retrieved Key Entries	*Retrieval Time (milliseconds)*							
	RixLite				*RixStd*			
	Cold Start		*Hot Start*		*Cold Start*		*Hot Start*	
	inc	*total*	*inc*	*total*	*inc*	*total*	*inc*	*total*
1k	11 812	11 812	31	31	9 875	9 875	31	31
2k	7 671	19 484	15	46	6 828	16 703	31	62
3k	6 437	25 921	31	78	6 000	22 703	15	78
4k	4 515	30 437	15	93	4 406	27 109	31	109
5k	4 343	34 781	31	125	3 578	30 687	15	125
6k	3 156	37 953	15	140	2 781	33 468	31	156
7k	2 640	40 593	31	171	2 531	36 000	15	171
8k	2 187	42 781	15	187	2 140	38 140	31	203
9k	2 171	44 953	31	218	1 890	40 031	15	218
10k	1 812	46 765	31	250	1 765	41 796	31	250

Table 5.14: Retrieval time of random accesses for Key Entries of RixLite and RixStd

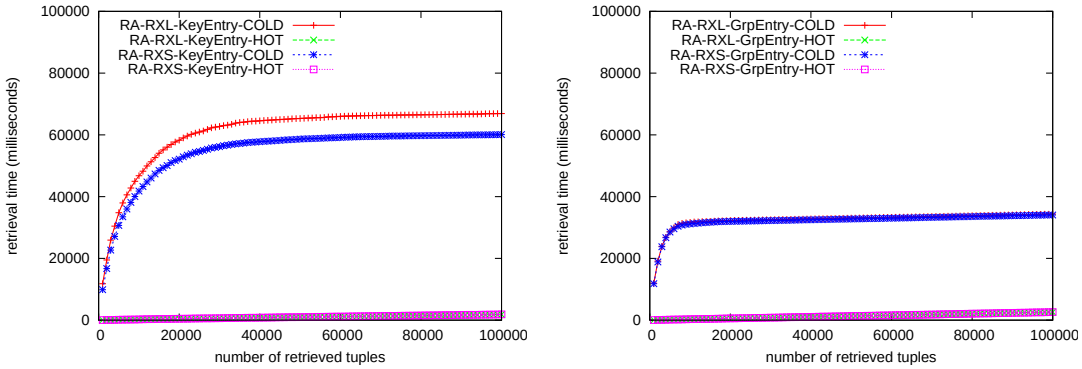
indicates that OS caching had showed positive effect earlier in batch random accesses for Group Entries than in batch random accesses for Key Entries. Considering the format of document IDs in TREC collection, one possible explanation for this observation could be there are more collisions in the external lookup table of Group Entries, because collided entries are chained and stored in the same block, and OS caching could reduce the accessing cost when the same block is accessed repeatedly.

Number of Retrieved Group Entries	*Retrieval Time (milliseconds)*							
	RixLite				*RixStd*			
	Cold Start		*Hot Start*		*Cold Start*		*Hot Start*	
	inc	*total*	*inc*	*total*	*inc*	*total*	*inc*	*total*
1k	11 968	11 968	46	46	11 796	11 796	46	46
2k	7 140	19 109	46	93	7 062	18 859	46	93
3k	4 812	23 921	31	125	4 906	23 765	15	109
4k	2 906	26 828	31	156	2 937	26 703	31	140
5k	1 812	28 640	31	187	1 843	28 546	31	171
6k	1 250	29 890	31	218	1 046	29 593	15	187
7k	1 093	30 984	15	234	828	30 421	31	218
8k	375	31 359	31	265	375	30 796	15	234
9k	187	31 546	31	296	234	31 031	31	265
10k	125	31 671	15	312	140	31 171	31	296

Table 5.15: Retrieval time of random accesses for Group Entries of RixLite and RixStd

Moreover, the full results of random accesses for data entries are illustrated in Figure 5.24. Both results (see Figure 5.24a and Figure 5.24b) show that when applying a large amount of random accesses on RixLite and RixStd could be very expensive at the beginning, while the

costs could be reduced dramatically later. On the other hand, the overall costs of retrievals from hot starts are much lower than the overall costs from cold starts, which suggests "warm-up" operations could be very helpful for retrieving data entries by random accesses.



(a) RA Key Entries (b) RA Group Entries

Figure 5.24: Performances of random accesses for retrieving entries

In summary, the results have shown that the random access facilities of RixLite and RixStd could effectively benefit from OS caching, which indicate the data structures of RixLite and RixStd are properly designed and are capable to handle random accesses efficiently in retrieval tasks.

By Integrating with Query Engine Finally, we measured the retrieval performances of query processing when utilising RIX indexes in a query engine. Though the experiment mainly focused on investigating the efficiency while effectiveness measurements in precision are also given. The experiment is similar to the previous experiment in Section 3.5, while here we studied the engine performance using different indexes for the scoring models.

In the experiment, five scoring models are applied to the 50 queries of TREC topic 151-200 using title only, retrieval time are measured respectively while utilising RixLite and RixStd. The results are given in Table 5.16.

The results show that the performances of retrieval using different RIX indexes with regards to efficiency are very similar, where both indexes can support efficient query processing for popular IR models. In addition, the MAP for *tf-idf* model is 0.1192 and P@10 is 0.212, and the MAP for LM model is 0.1873 and P@10 is 0.362.

Scoring Model	*RixLite*					*RixStd*				
	Retrieval Time (sec)			*Effectiveness*		*Retrieval Time (sec)*			*Effectiveness*	
	avg	*min*	*max*	*MAP*	*P@10*	*avg*	*min*	*max*	*MAP*	*P@10*
$P_C(t\|d)$	2.669	0.141	9.344	–	–	2.642	0.141	9.234	–	–
$P_C(t)$	0.001	0	0.047	–	–	0.001	0	0.047	–	–
df	0.001	0	0.047	–	–	0.001	0	0.047	–	–
tf-idf	4.258	0.219	14.766	0.1192	0.212	4.32	0.234	14.781	0.1192	0.212
LM	4.75	0.203	16.016	0.1873	0.362	4.8	0.219	16.11	0.1873	0.362

Table 5.16: Retrieval time for the queries of TREC topics 151-200, using title only

5.4 Summary

To summarise this chapter, we studied indexing methods that are suitable for IR+DB system, where the main contributions include: we proposed a relational inverted index (RIX) architecture and studied three types RIX instance, i.e. RixLite, RixStd and RixExt, and their constructing methods; in particular, we studied varied RIX constructing algorithms based on different scheduling methods, which include naive build, adaptive build and analytical build; moreover, we investigated the retrieval performances of the RIX indexes while considering several aspects.

To evaluate the indexing performances for building the proposed RIX instances with certain scheduling methods, we conducted experiments using a multi-gigabyte TREC collection on standalone PCs. Experimental results indicate that the proposed indexing methods are capable to construct RIX instances on standalone commodity machines efficiently, while the retrieval performances of different RIX instances are sufficient to be applied in a IR+DB query engine.

For future work, one of the interesting directions is to investigate the construction methods for building RIX instances in parallel or distributed environments, so that to scale up the indexing for very large data sets.

Chapter 6

Conclusion

6.1 Main Contributions

To conclude, this thesis presents three techniques for improving efficiency and scalability for IR+DB systems, which include scoring-driven optimization with scoring expression (SCX), top-k incorporated pipeline (TIP), and relational inverted index (RIX). The main contributions can be summarised as the following aspects:

- We discussed the criteria for PRA expressions to be equivalent, which is different from the traditional criterion required in conventional databases: to verify equivalence for PRA expressions, logical optimization methods not only need to prove relational equivalence, but also need to consider scoring or ranking factors.

- We proposed scoring expression for articulating scoring functions for PRA expressions. This is necessary in order to interpret the scoring semantics implied by PRA expressions, so that the scoring or ranking features of PRA expressions can be testified.

- We proposed scoring-driven optimization utilising scoring expressions, which aims to support efficient query processing from different angles, which include assisting index selection, aligning scoring functions with intensional semantics, and verifying algebra equivalence of PRA expressions. As a result, based on scoring or ranking equivalence of PRA expressions, judicious query (execution) plan could be generated for processing PRA queries efficiently.

- We proposed (conceptually) top-k incorporated pipeline for executing PRA queries. While developments for a full-fledged pipelined query execution engine is ongoing, we simulate TIP on HySpirit, which is a IR+DB prototype, where we investigated performances tradeoff with regards to efficiency versus effectiveness, and found sophisticatedly designed top-k mechanisms can dramatically speed up response time for queries but only cause tolerable losses in retrieval qualities.

- We proposed relational inverted index specifically for IR+DB systems. In particular, three RIX instances were designed to combine IR-style inverted index and DB-style TID-index (tuple-based index) in different degrees. As a result, RIX can be utilised not only for supporting efficient query processing for popular IR models and retrieval strategies that implemented in PRA, but also for supporting efficient execution for conventional relational operators in PRA that are used to model complex queries.

- We designed sophisticated constructing algorithms of RIX instances, in particular, we proposed three construction scheduling methods. Experiments showed that RIX indexer is capable to index several gigabytes of data efficiently on inexpensive commodity machines with limited memory allowance; especially, the adaptive and analytical scheduling algorithms may enable RIX indexer to be scalable for even larger data sets (tens of gigabytes).

In addition, we developed the proposed techniques (with a few work in progress) into an IR+DB prototype Birdie, which is not only a contribution of engineering efforts, but also an attempt of rethinking and redesign for suitable infrastructures (such as suggested in [Chaudhuri et al., 2005]) for future IR and DB integrated applications.

6.2 Statement on Research Questions

At the beginning of this thesis, we described three research questions that are interesting in the area of integrating IR and DB technologies, while here we discuss how the contributions in this work may satisfy these questions.

How to optimize probabilistic relational algebra expressions so that to generate logical or physical query plans that can be processed efficiently? The proposed scoring-driven optimization based on scoring expressions is one of the solutions for the problem. Although we did not study how to optimize PRA expressions directly such as it is suggested in the question, however, the

proposed technique satisfies the main aim of logical query optimization, which is to improve the processing efficiency of PRA expressions. In particular, one of the advantages of scoring-driven optimization is to generate efficient execution plans, which utilise sophisticated designed physical operators and special indexes such as RIX, thus the processing performances for probability estimation and aggregation can be improved.

How to incorporate top-k processing mechanisms into generic query execution engine of IR+DB systems or infrastructures. We studied from a conceptual point of view of how to integrate top-k incorporated pipeline into a query engine for processing PRA expressions. In addition, we implemented externally from the query processing engine a top-k processing method based on budgetary constraint. In general, the current work partly satisfies the above research question. It is worth mentioning that similar efforts have been made for integrating pipelined top-k operators into DBMS, the results of a number of previous work such as [Ilyas et al., 2003, Li et al., 2006, Li et al., 2005] can be inspiring for implementing physical TIP into IR+DB systems.

How to adapt IR-style indexing methods into an IR+DB platform, so that to provide efficient accessibility to statistics that are needed for flexible scoring and ranking, and how to design and implement such index that is scalable for large-scale data? The contribution on RIX indexing technique is a competent candidate for answering the above question. First of all, RIX has been designed to exploit IR-style indexing structures, and it also adapts conventional tuple-based indexing structures of databases, so that requirement of efficient accessibility can be fully satisfied. Secondly, a number of constructing algorithms for building RIX have been studied, in which sophisticated scheduling methods are proposed to index very large data set. Thirdly, the performances of different RIX instances have been carefully investigated, where experimental results have shown that the indexing technique is capable for Gigabytes of data on standalone PCs.

Research Hypothesis The hypothesis of this thesis states that the efficiency of IR and DB integrated systems can be improved by adapting and evolving state-of-the-art techniques in IR and DB, the contribution of this work have shown that the hypothesis is held. For instance, the work on scoring-driven optimization evolves traditional logical optimization for relational algebra, and proposed an optimization technique that is based on manipulates the scoring property of PRA, which can be seen as a step forward of conventional query optimizations in relational retrieval

systems. In addition, the top-k incorporated pipeline adapts the concept of top-k processing from both IR and DB communities, and the pipelined top-k operators is also a popular topic in databases. Moreover, the origin of relational inverted index is from the well-studied inverted files in IR, and note that indexed indexes had also been used in early years in databases. Therefore, integrating inverted index and tuple-based index has solid theoretical and practical backgrounds. In a word, for improving the efficiency and scalability of IR+DB systems, it is reasonable and effective to integrate related solutions from IR and DB.

6.3 Future Work

There are a number of potential areas in the field with respect to efficiency and scalability related technologies that would be interesting and worthy for further investigations and studies. However, there is one particular technology that we were especially interested, but have not been able to spend efforts on the topic because of limited resources and time. Therefore, we would like to outline this for the future work, which is parallel and distributed computing technology. Moreover, considering scoring-driven optimization could be another interesting orientation for optimizing probabilistic relational algebra or other similar variants in addition to algebraic optimization and cost-driven optimization, we would also like to state some notes for further research.

Exploiting Parallel and Distributed Computing Powers In this thesis, the techniques have been introduced should be able to handle several gigabytes (could be up to a few tens of gigabytes) of data on a single standalone PC, however, this is far not enough for real-world applications in nowadays which usually involves several terabytes or even petabytes of data. In order to develop IR+DB systems that could be competent enough for ever growing data, parallel and distributed computing is an enabling and well-established technology that should be exploited.

In both DB and IR community, parallel computing has been widely applied in practical systems, where parallel databases (e.g. see [DeWitt and Gray, 1992]) and IR search engines (e.g. see [Dean and Ghemawat, 2004]) provide useful examples for developing parallel IR+DB systems. Especially, when distributed computing being introduced to the more general public under the concept of "Cloud" in recent, parallelism became a popular topic "again", and recent developments of MapReduce [Dean and Ghemawat, 2004] also caught notices from the DB community

(e.g. see notes posted by David DeWitt and Michael Stonebraker [1]).

Some potential research directions could be parallel constructing algorithms for RIX, and parallel query processing technique incorporating top-k mechanisms. Another interesting and critical aspect should be carefully considered while applying parallelism relates to probability estimation and aggregation. For instance, whether probability estimation must to be based on global statistics? Shall aggregation to be performed for aggregating tuple frequency or tuple probability? These could be some potential questions when considering to apply parallelism under probabilistic paradigm, and more similar questions could be waiting for us to answer.

Scoring and Ranking Driven Optimization In this thesis, we discussed a scoring-driven optimization method based on strict scoring equivalence, but whether a PRA query could be processed efficiently fully depends on if certain special implementations for particular manners of probability (or score) estimation or aggregation are available. In other words, if a query engine has been implemented only generic operators, i.e. there are no special probability (or score) estimator and aggregator available, then it might not be able to benefit from the advantages provided by a scoring-driven optimizer.

Therefore, it is intrinsically interesting to investigate optimization methods that are based on soft scoring equivalence or even ranking equivalence. For example, given an complicated PRA expression which could be time consuming to execute, a optimizer could rewrite a PRA expression which is less expensive than the original expression, while the rewritten expression satisfies relational equivalence, and the tuples in the result are in similar order but not thoroughly identical order comparing to the original result. Similar optimization technique would greatly increase the chances for a generic query engine to execute any arbitrary PRA expressions efficiently.

6.4 Summary

This thesis enters the field of integrated information retrieval and database technologies with a broad view, while focuses on three specific techniques for improving the efficiency and scalability of IR+DB infrastructure. If this study can be viewed as a quest for the goal as it is stated in the title, then the adventure has not yet finished but rather just has begun.

As it has been addressed by many researchers in the area of integrated IR and DB technologies, IR and DB have been developed separately for decades, but both communities have started

[1] http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/ , and http://databasecolumn.vertica.com/database-innovation/mapreduce-ii/

to realise that the two fields have a number of aspects in common in terms of research interests and technologies. The proposed techniques in the thesis may be examples of how to bring technologies from either sides to tackle problems that are interested by both communities, while in our case, it is to provide efficient and scalable solutions for managing and searching structured and unstructured data for modern information applications.

Bibliography

[Abiteboul et al., 2003] Abiteboul, S., Agrawal, R., Bernstein, P. A., Carey, M. J., Ceri, S., Croft, W. B., DeWitt, D. J., Franklin, M. J., Garcia-Molina, H., Gawlick, D., Gray, J., Haas, L. M., Halevy, A. Y., Hellerstein, J. M., Ioannidis, Y. E., Kersten, M. L., Pazzani, M. J., Lesk, M., Maier, D., Naughton, J. F., Schek, H.-J., Sellis, T. K., Silberschatz, A., Stonebraker, M., Snodgrass, R. T., Ullman, J. D., Weikum, G., Widom, J., and Zdonik, S. B. (2003). The lowell database research self assessment. *CoRR*, cs.DB/0310006.

[Agrawal et al., 2008] Agrawal, R., Ailamaki, A., Bernstein, P. A., Brewer, E. A., Carey, M. J., Chaudhuri, S., Doan, A., Florescu, D., Franklin, M. J., Garcia-Molina, H., Gehrke, J., Gruenwald, L., Haas, L. M., Halevy, A. Y., Hellerstein, J. M., Ioannidis, Y. E., Korth, H. F., Kossmann, D., Madden, S., Magoulas, R., Ooi, B. C., O'Reilly, T., Ramakrishnan, R., Sarawagi, S., Stonebraker, M., Szalay, A. S., and Weikum, G. (2008). The claremont report on database research. *SIGMOD Record*, 37(3):9–19.

[Agrawal et al., 2002] Agrawal, S., Chaudhuri, S., and Das, G. (2002). DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16.

[Agrawal et al., 2003] Agrawal, S., Chaudhuri, S., Das, G., and Gionis, A. (2003). Automated ranking of database query results. In *CIDR*.

[Agrawal et al., 2000] Agrawal, S., Chaudhuri, S., and Narasayya, V. R. (2000). Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505.

[Amati and van Rijsbergen, 2002] Amati, G. and van Rijsbergen, C. J. (2002). Probabilistic models of information retrieval based on measuring the divergence from randomness. *ACM Transaction on Information Systems (TOIS)*, 20(4):357–389.

[Antova et al., 2007a] Antova, L., Jansen, T., Koch, C., and Olteanu, D. (2007a). Fast and simple relational processing of uncertain data. *CoRR*, abs/0707.1644.

[Antova et al., 2007b] Antova, L., Koch, C., and Olteanu, D. (2007b). From complete to incomplete information and back. In *SIGMOD Conference*, pages 713–724.

[Antova et al., 2007c] Antova, L., Koch, C., and Olteanu, D. (2007c). Query language support for incomplete information in the MayBMS system. In *VLDB*, pages 1422–1425.

[Arasu et al., 2001] Arasu, A., Cho, J., Garcia-Molina, H., Paepcke, A., and Raghavan, S. (2001). Searching the web. *ACM Trans. Internet Techn.*, 1(1):2–43.

[Baeza-Yates and Ribeiro-Neto, 1999a] Baeza-Yates, R. and Ribeiro-Neto, B. (1999a). *Modern Information Retrieval*. Addison Wesley.

[Baeza-Yates and Ribeiro-Neto, 1999b] Baeza-Yates, R. A. and Ribeiro-Neto, B. A. (1999b). *Modern Information Retrieval*. ACM Press / Addison-Wesley.

[Bast et al., 2006] Bast, H., Majumdar, D., Schenkel, R., Theobald, M., and Weikum, G. (2006). IO-Top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486.

[Beeri and Ramakrishnan, 1991] Beeri, C. and Ramakrishnan, R. (1991). On the power of magic. *J. Log. Program.*, 10(1/2/3&4):255–299.

[Benjelloun et al., 2006a] Benjelloun, O., Sarma, A. D., Halevy, A. Y., and Widom, J. (2006a). ULDBs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964.

[Benjelloun et al., 2006b] Benjelloun, O., Sarma, A. D., Hayworth, C., and Widom, J. (2006b). An introduction to ULDBs and the trio system. *IEEE Data Eng. Bull.*, 29(1):5–16.

[Berger and Lafferty, 1999] Berger, A. and Lafferty, J. (1999). Information retrieval as statistical translation. In SIGIR, editor, *SIGIR '99, Proceedings of the 22nd International Conference on Research and Development in Information Retrieval*, pages 222–229, New York. ACM.

[Blizard, 1989] Blizard, W. D. (1989). Multiset theory. *Notre Dame Journal of Formal Logic*, 30(1):36–66.

[Boncz et al., 2006] Boncz, P. A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., and Teubner, J. (2006). Monetdb/xquery: a fast xquery processor powered by a relational engine. In *SIGMOD Conference*, pages 479–490.

[Bosc et al., 1988] Bosc, P., Galibourg, M., and Hamon, G. (1988). Fuzzy querying with sql: extensions and implementation aspects. *Fuzzy Sets Syst.*, 28(3):333–349.

[Boulos et al., 2005] Boulos, J., Dalvi, N. N., Mandhani, B., Mathur, S., Re, C., and Suciu, D. (2005). Mystiq: a system for finding more answers by using probabilities. In *SIGMOD Conference*, pages 891–893.

[Brin and Page, 1998] Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. In *7th International WWW Conference, Brisbane, Australia*.

[Bruno et al., 2002] Bruno, N., Gravano, L., and Marian, A. (2002). Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–.

[Buneman et al., 1994] Buneman, P., Libkin, L., Suciu, D., Tannen, V., and Wong, L. (1994). Comprehension syntax. *SIGMOD Record*, 23(1):87–96.

[Büttcher and Clarke, 2006] Büttcher, S. and Clarke, C. L. A. (2006). A document-centric approach to static index pruning in text retrieval systems. In *CIKM*, pages 182–189.

[Cao and Wang, 2004] Cao, P. and Wang, Z. (2004). Efficient top-k query calculation in distributed networks. In *PODC*, pages 206–215.

[Carey and Kossmann, 1997] Carey, M. J. and Kossmann, D. (1997). On saying "enough already!" in sql. In *SIGMOD Conference*, pages 219–230.

[Carey and Kossmann, 1998] Carey, M. J. and Kossmann, D. (1998). Reducing the braking distance of an sql query engine. In *VLDB*, pages 158–169.

[Cavallo and Pittarelli, 1987] Cavallo, R. and Pittarelli, M. (1987). The theory of probabilistic databases. In *Proceedings of the Conference on Very Large Databases (VLDB)*.

[Chan, 2002] Chan, C. C. (2002). The state of the art of electric and hybrid vehicles. *Proceedings of the IEEE*, 90(2):247–275.

[Chang et al., 2006] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. (2006). Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI*, pages 205–218.

[Chang and won Hwang, 2002] Chang, K. C.-C. and won Hwang, S. (2002). Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357.

[Chang et al., 2000] Chang, Y.-C., Bergman, L. D., Castelli, V., Li, C.-S., Lo, M.-L., and Smith, J. R. (2000). The onion technique: Indexing for linear optimization queries. In *SIGMOD Conference*, pages 391–402.

[Chaudhuri, 1998] Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *PODS*, pages 34–43.

[Chaudhuri et al., 2004] Chaudhuri, S., Das, G., Hristidis, V., and Weikum, G. (2004). Probabilistic ranking of database query results. In *VLDB*, pages 888–899.

[Chaudhuri and Gravano, 1996] Chaudhuri, S. and Gravano, L. (1996). Optimizing queries over multimedia repositories. In *SIGMOD Conference*, pages 91–102.

[Chaudhuri et al., 2005] Chaudhuri, S., Ramakrishnan, R., and Weikum, G. (2005). Integrating db and ir technologies: What is the sound of one hand clapping? In *CIDR*, pages 1–12.

[Chaudhuri and Shim, 1995] Chaudhuri, S. and Shim, K. (1995). An overview of cost-based optimization of queries with aggregates. *IEEE Data Eng. Bull.*, 18(3):3–9.

[Chaudhuri and Shim, 1996] Chaudhuri, S. and Shim, K. (1996). Optimizing queries with aggregate views. In *EDBT*, pages 167–182.

[Cherniack and Zdonik, 1998] Cherniack, M. and Zdonik, S. B. (1998). Changing the rules: Transformations for rule-based optimizers. In *SIGMOD Conference*, pages 61–72.

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

[Cole and Graefe, 1994] Cole, R. L. and Graefe, G. (1994). Optimization of dynamic query evaluation plans. In *SIGMOD Conference*, pages 150–160.

[Cornacchia and de Vries, 2006] Cornacchia, R. and de Vries, A. P. (2006). A declarative DB-powered approach to IR. In *ECIR*, pages 543–547.

[Cornacchia and de Vries, 2007] Cornacchia, R. and de Vries, A. P. (2007). A parameterised search system. In *ECIR*, pages 4–15.

[Cornacchia et al., 2008] Cornacchia, R., Héman, S., Zukowski, M., de Vries, A. P., and Boncz, P. A. (2008). Flexible and efficient IR using array databases. *VLDB J.*, 17(1).

[Craswell et al., 2001] Craswell, N., Hawking, D., and Robertson, S. E. (2001). Effective site finding using link anchor information. In *SIGIR*, pages 250–257.

[Croft and Harper, 1979] Croft, W. and Harper, D. (1979). Using probabilistic models of document retrieval without relevance information. *Journal of Documentation*, 35:285–295.

[Croft et al., 1998] Croft, W. B., Moffat, A., van Rijsbergen, C. J., Wilkinson, R., and Zobel, J., editors (1998). *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York. ACM.

[Croft and Schek, 2008] Croft, W. B. and Schek, H.-J. (2008). Introduction to the special issue on database and information retrieval integration. *VLDB J.*, 17(1):1–3.

[Dalvi and Suciu, 2004] Dalvi, N. N. and Suciu, D. (2004). Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875.

[Dalvi and Suciu, 2005] Dalvi, N. N. and Suciu, D. (2005). Answering queries from statistics and probabilistic views. In *VLDB*, pages 805–816.

[Das et al., 2006] Das, G., Gunopulos, D., Koudas, N., and Tsirogiannis, D. (2006). Answering top-k queries using views. In *VLDB*, pages 451–462.

[de Moura et al., 2005] de Moura, E. S., dos Santos, C. F., Fernandes, D. R., da Silva, A. S., Calado, P., and Nascimento, M. A. (2005). Improving web search efficiency via a locality based static pruning method. In *WWW*, pages 235–244.

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150.

[DeWitt and Gray, 1992] DeWitt, D. J. and Gray, J. (1992). Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98.

[DeWitt et al., 1984] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M., and Wood, D. A. (1984). Implementation techniques for main memory database systems. In *SIGMOD Conference*, pages 1–8.

[Dittrich et al., 2002] Dittrich, J.-P., Seeger, B., Taylor, D. S., and Widmayer, P. (2002). Progressive merge Join: A generic and non-blocking sort-based join algorithm. In *VLDB*, pages 299–310.

[Donjerkovic and Ramakrishnan, 1999] Donjerkovic, D. and Ramakrishnan, R. (1999). Probabilistic optimization of top N queries. In *VLDB*, pages 411–422.

[Ercegovac et al., 2005] Ercegovac, V., DeWitt, D. J., and Ramakrishnan, R. (2005). The TEXTURE benchmark: Measuring performance of text queries on a relational DBMS. In *VLDB*, pages 313–324.

[Estivill-Castro and Wood, 1992] Estivill-Castro, V. and Wood, D. (1992). A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476.

[Fagin, 1996] Fagin, R. (1996). Combining fuzzy information from multiple systems. In *PODS*, pages 216–226.

[Fagin, 1999] Fagin, R. (1999). Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99.

[Fagin and Halpern, 1994] Fagin, R. and Halpern, J. Y. (1994). Reasoning about knowledge and probability. *J. ACM*, 41(2):340–367.

[Fagin et al., 2003a] Fagin, R., Kumar, R., and Sivakumar, D. (2003a). Efficient similarity search and classification via rank aggregation. In *SIGMOD Conference*, pages 301–312.

[Fagin et al., 2001] Fagin, R., Lotem, A., and Naor, M. (2001). Optimal aggregation algorithms for middleware. In *PODS*.

[Fagin et al., 2003b] Fagin, R., Lotem, A., and Naor, M. (2003b). Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656.

[Faloutsos and Christodoulakis, 1984] Faloutsos, C. and Christodoulakis, S. (1984). Signature files: An access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst.*, 2(4):267–288.

[Fuhr, 1990] Fuhr, N. (1990). A probabilistic framework for vague queries and imprecise information in databases. In *VLDB*, pages 696–707.

[Fuhr, 1993] Fuhr, N. (1993). A probabilistic relational model for the integration of ir and databases. In *Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 309–17, New York.

[Fuhr, 1999a] Fuhr, N. (1999a). A decision-theoretic approach to database selection in networked ir. *ACM Transactions on Information Systems*, 17(3):229–249.

[Fuhr, 1999b] Fuhr, N. (1999b). Resource discovery in distributed digital libraries. In *Digital Libraries '99: Advanced Methods and Technologies, Digital Collections*, pages 35–45. St. Petersburg State University.

[Fuhr, 2000] Fuhr, N. (2000). Probabilistic Datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science*, 51(2):95–110.

[Fuhr et al., 1998] Fuhr, N., Goevert, N., and Roelleke, T. (1998). Dolores: A system for logic-based retrieval of multimedia objects. In [Croft et al., 1998], pages 257–265.

[Fuhr and Gövert, 2006] Fuhr, N. and Gövert, N. (2006). Retrieval quality vs. effectiveness of specificity-oriented search in xml collections. *Inf. Retr.*, 9(1):55–70.

[Fuhr and Großjohann, 2004] Fuhr, N. and Großjohann, K. (2004). XIRQL: An XML query language based on information retrieval concepts. *ACM Trans. Inf. Syst.*, 22(2):313–356.

[Fuhr and Roelleke, 1998] Fuhr, N. and Roelleke, T. (1998). HySpirit — a probabilistic inference engine for hypermedia retrieval in large databases. In Schek, H.-J., Saltor, F., Ramos, I., and Alonso, G., editors, *Proceedings of the 6th International Conference on Extending Database Technology (EDBT), Valencia, Spain*, Lecture Notes in Computer Science, pages 24–38, Berlin et al. Springer.

[Fuhr and Rölleke, 1997] Fuhr, N. and Rölleke, T. (1997). A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. *ACM Transactions on Information Systems*, 15(1):32–66.

[Galil and Italiano, 1991] Galil, Z. and Italiano, G. F. (1991). Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344.

[Gao et al., 2005] Gao, Y., Ehsani, M., and Miller, J. (2005). Hybrid electric vehicle: Overview and state of the art. In *ISIE*, pages 307–316.

[Goldstein and Larson, 2001] Goldstein, J. and Larson, P.-Å. (2001). Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, pages 331–342.

[Golfarelli et al., 2002] Golfarelli, M., Rizzi, S., and Saltarelli, E. (2002). Index selection for data warehousing. In *DMDW*, pages 33–42.

[Grabs et al., 2001] Grabs, T., Böhm, K., and Schek, H.-J. (2001). PowerDB-IR - information retrieval on top of a database cluster. In *CIKM*, pages 411–418.

[Grabs et al., 2004] Grabs, T., Böhm, K., and Schek, H.-J. (2004). PowerDB-IR - scalable information retrieval and storage with a cluster of databases. *Knowl. Inf. Syst.*, 6(4):465–505.

[Graefe, 1993] Graefe, G. (1993). Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170.

[Graefe, 1994] Graefe, G. (1994). Sort-Merge-Join: An idea whose time has(h) passed? In *ICDE*, pages 406–417.

[Graefe, 2006] Graefe, G. (2006). Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3).

[Graefe and Ward, 1989] Graefe, G. and Ward, K. (1989). Dynamic query evaluation plans. In *SIGMOD Conference*, pages 358–366.

[Grossman and Frieder, 2004] Grossman, D. A. and Frieder, O. (2004). *Information Retrieval. Algorithms and Heuristics, 2nd ed.*, volume 15 of *The Information Retrieval Series*. Springer.

[Grossman et al., 1997] Grossman, D. A., Frieder, O., Holmes, D. O., and Roberts, D. C. (1997). Integrating structured data and text: A relational approach. *JASIS*, 48(2):122–132.

[Güntzer et al., 2000] Güntzer, U., Balke, W.-T., and Kießling, W. (2000). Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428.

[Guo et al., 2003] Guo, L., Shao, F., Botev, C., and Shanmugasundaram, J. (2003). Xrank: Ranked keyword search over xml documents. In *SIGMOD Conference*, pages 16–27.

[Haas and Hellerstein, 1999] Haas, P. J. and Hellerstein, J. M. (1999). Ripple joins for online aggregation. In *SIGMOD Conference*, pages 287–298.

[Harman, 1994] Harman, D. (1994). Overview of the third text retrieval conference (trec-3). In *TREC*, pages 0–.

[Hawking, 1998] Hawking, D. (1998). Efficiency/effectiveness trade-offs in query processing. *SIGIR Forum*, 32(2):16–22.

[Hawking, 2004] Hawking, D. (2004). Challenges in enterprise search. In *Proceedings of the Australasian Database Conference ADC2004*, pages 15–26, Dunedin, New Zealand. Invited paper: urlhttp://es.csiro.au/pubs/hawking_adc04keynote.pdf.

[Hébert, 2006] Hébert, L. (2006). The semantic graph. *Signo [online], Rimouski (Quebec), http://www.signosemio.com*.

[Hiemstra, 2000] Hiemstra, D. (2000). A probabilistic justification for using tf.idf term weighting in information retrieval. *International Journal on Digital Libraries*, 3(2):131–139.

[Hiemstra, 2002] Hiemstra, D. (2002). A database approach to content-based xml retrieval. In *INEX Workshop*, pages 111–118.

[Hoare, 1961] Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321.

[Hristidis et al., 2003] Hristidis, V., Gravano, L., and Papakonstantinou, Y. (2003). Efficient IR-style keyword search over relational databases. In *VLDB*, pages 850–861.

[Hristidis et al., 2001] Hristidis, V., Koudas, N., and Papakonstantinou, Y. (2001). Prefer: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD Conference*, pages 259–270.

[Hristidis and Papakonstantinou, 2002] Hristidis, V. and Papakonstantinou, Y. (2002). DISCOVER: Keyword search in relational databases. In *VLDB*, pages 670–681.

[Hristidis and Papakonstantinou, 2004] Hristidis, V. and Papakonstantinou, Y. (2004). Algorithms and applications for answering ranked queries using ranked views. *VLDB J.*, 13(1):49–70.

[Ilyas et al., 2002] Ilyas, I. F., Aref, W. G., and Elmagarmid, A. K. (2002). Joining ranked inputs in practice. In *VLDB*, pages 950–961.

[Ilyas et al., 2003] Ilyas, I. F., Aref, W. G., and Elmagarmid, A. K. (2003). Supporting Top-k join queries in relational databases. In *VLDB*, pages 754–765.

[Ilyas et al., 2006] Ilyas, I. F., Aref, W. G., Elmagarmid, A. K., Elmongui, H. G., Shah, R., and Vitter, J. S. (2006). Adaptive rank-aware query optimization in relational databases. *ACM Trans. Database Syst.*, 31(4):1257–1304.

[Ilyas et al., 2008] Ilyas, I. F., Beskales, G., and Soliman, M. A. (2008). A survey of top- query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4).

[Ilyas et al., 2004] Ilyas, I. F., Shah, R., Aref, W. G., Vitter, J. S., and Elmagarmid, A. K. (2004). Rank-aware query optimization. In *SIGMOD Conference*, pages 203–214.

[Kabra et al., 2003] Kabra, N., Ramakrishnan, R., and Ercegovac, V. (2003). The quiq engine: A hybrid ir db system. In *ICDE*, pages 741–.

[Kimball and Strehlo, 1995] Kimball, R. and Strehlo, K. (1995). Why decision support fails and how to fix it. *SIGMOD Record*, 24(3):92–97.

[Klug, 1982] Klug, A. C. (1982). Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717.

[Knuth, 1973] Knuth, D. E. (1973). *The Art of Computer Programming, Volume III: Fundamental Algorithms*, volume III.

[Lafferty and Zhai, 2003] Lafferty, J. and Zhai, C. (2003). *Probabilistic Relevance Models Based on Document and Query Generation*, chapter 1. Kluwer.

[Lalmas and Roelleke, 2002] Lalmas, M. and Roelleke, T. (2002). Four-valued knowledge augmentation for structured document retrieval. In *Proceedings of the 13th International Symposium on Methodologies for Intelligent Systems (ISMIS), Lyon, France*.

[Li et al., 2006] Li, C., Chang, K. C.-C., and Ilyas, I. F. (2006). Supporting ad-hoc ranking aggregates. In *SIGMOD Conference*, pages 61–72.

[Li et al., 2005] Li, C., Chang, K. C.-C., Ilyas, I. F., and Song, S. (2005). RankSQL: Query algebra and optimization for relational Top-k queries. In *SIGMOD Conference*, pages 131–142.

[Li et al., 2002] Li, W., Gao, D., and Snodgrass, R. T. (2002). Skew handling techniques in sort-merge join. In *SIGMOD Conference*, pages 169–180.

[MacFarlane, 2000] MacFarlane, A. (2000). *Distributed inverted files and performance: a study of parallelism and data distribution methods in IR*. City University London. PhD thesis.

[Manber and Myers, 1990] Manber, U. and Myers, G. (1990). Suffix arrays: A new method for on-line string searches. In *SODA*, pages 319–327.

[Marian et al., 2005] Marian, A., Amer-Yahia, S., Koudas, N., and Srivastava, D. (2005). Adaptive processing of Top-K queries in XML. In *ICDE*, pages 162–173.

[Marian et al., 2004] Marian, A., Bruno, N., and Gravano, L. (2004). Evaluating top- queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362.

[Maron and Kuhns, 1960] Maron, M. and Kuhns, J. (1960). On relevance, probabilistic indexing, and information retrieval. *Journal of the ACM*, 7:216–244.

[Melnik et al., 2001] Melnik, S., Raghavan, S., Yang, B., and Garcia-Molina, H. (2001). Building a distributed full-text index for the web. In *WWW*, pages 396–406.

[Mishra and Eich, 1992] Mishra, P. and Eich, M. H. (1992). Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113.

[Moffat and Zobel, 1996] Moffat, A. and Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379.

[Mokbel et al., 2004] Mokbel, M. F., Lu, M., and Aref, W. G. (2004). Hash-Merge Join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, pages 251–263.

[Motro, 1988] Motro, A. (1988). Vague: A user interface to relational databases that permits vague queries. *ACM Transactions on Office Information Systems*, 6(3):187–214.

[Mumick et al., 1990a] Mumick, I. S., Finkelstein, S. J., Pirahesh, H., and Ramakrishnan, R. (1990a). Magic is relevant. In *SIGMOD Conference*, pages 247–258.

[Mumick et al., 1990b] Mumick, I. S., Pirahesh, H., and Ramakrishnan, R. (1990b). The magic of duplicates and aggregates. In *VLDB*, pages 264–277.

[Mutsuzaki et al., 2007] Mutsuzaki, M., Theobald, M., de Keijzer, A., Widom, J., Agrawal, P., Benjelloun, O., Sarma, A. D., Murthy, R., and Sugihara, T. (2007). Trio-One: Layering uncertainty and lineage on a conventional DBMS (demo). In *CIDR*, pages 269–274.

[Natsev et al., 2001] Natsev, A., Chang, Y.-C., Smith, J. R., Li, C.-S., and Vitter, J. S. (2001). Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290.

[Nepal and Ramakrishna, 1999] Nepal, S. and Ramakrishna, M. V. (1999). Query processing issues in image (multimedia) databases. In *ICDE*, pages 22–29.

[Ponte and Croft, 1998] Ponte, J. and Croft, W. (1998). A language modeling approach to information retrieval. In [Croft et al., 1998], pages 275–281.

[Re et al., 2007] Re, C., Dalvi, N. N., and Suciu, D. (2007). Efficient Top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895.

[Rijsbergen, 1979] Rijsbergen, C. K. v. (1979). *Information retrieval, 2nd edition*. Butterworth-sLondon.

[Robertson, 2004] Robertson, S. (2004). Understanding inverse document frequency: On theoretical arguments for idf. *Journal of Documentation*, 60:503–520.

[Robertson and Sparck Jones, 1976] Robertson, S. and Sparck Jones, K. (1976). Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27:129–146.

[Robertson, 1981] Robertson, S. E. (1981). Term frequency and term value. In *SIGIR*, pages 22–29.

[Robertson and Walker, 1994] Robertson, S. E. and Walker, S. (1994). Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR*, pages 232–241.

[Robertson et al., 2004] Robertson, S. E., Zaragoza, H., and Taylor, M. J. (2004). Simple BM25 extension to multiple weighted fields. In *CIKM*, pages 42–49.

[Roelleke, 1994] Roelleke, T. (1994). Equivalences of the probabilistic relational algebra. Technical report, University of Dortmund, Department of Computer Science.

[Roelleke, 1999] Roelleke, T. (1999). *POOL: Probabilistic Object-Oriented Logical Representation and Retrieval of Complex Objects*. Shaker Verlag, Aachen. Dissertation.

[Roelleke and Fuhr, 1996] Roelleke, T. and Fuhr, N. (1996). Retrieval of complex objects using a four-valued logic. In Frei, H.-P., Harmann, D., Schaeuble, P., and Wilkinson, R., editors, *Proceedings of the 19th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 206–214, New York. ACM.

[Roelleke et al., 2006] Roelleke, T., Tsikrika, T., and Kazai, G. (2006). A general matrix framework for modelling information retrieval. *Journal on Information Processing & Management (IP&M), Special Issue on Theory in Information Retrieval*, 42(1).

[Roelleke and Wang, 2006] Roelleke, T. and Wang, J. (2006). A parallel derivation of probabilistic information retrieval models. In *ACM SIGIR*, pages 107–114, Seattle, USA.

[Roelleke and Wang, 2008] Roelleke, T. and Wang, J. (2008). Tf-idf undercovered: A study of theories and probabilities. In *SIGIR*.

[Roelleke et al., 2008] Roelleke, T., Wu, H., Wang, J., and Azzam, H. (2008). Modelling retrieval models in a probabilistic relational algebra with a new operator: the relational Bayes. *VLDB J.*, 17(1):5–37.

[Rölleke et al., 2001] Rölleke, T., Lübeck, R., and Kazai, G. (2001). The hyspirit retrieval platform. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, page 454, New York, NY, USA. ACM Press.

[Schek and Pistor, 1982] Schek, H.-J. and Pistor, P. (1982). Data structures for an integrated database management and information retrieval system. In *Proceedings of the 8th International Conference on Very Large Data Bases*, pages 197–207, Los Altos, California. Morgan Kaufman.

[Shapiro, 1986] Shapiro, L. D. (1986). Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264.

[Shmueli-Scheuer et al., 2009] Shmueli-Scheuer, M., Li, C., Mass, Y., Roitman, H., Schenkel, R., and Weikum, G. (2009). Best-effort top-k query processing under budgetary constraints. In *ICDE*, pages 928–939.

[Soffer et al., 2001] Soffer, A., Carmel, D., Cohen, D., Fagin, R., Farchi, E., Herscovici, M., and Maarek, Y. S. (2001). Static index pruning for information retrieval systems. In *SIGIR*, pages 43–50.

[Sowa, 1984] Sowa, J. F. (1984). *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley.

[Stonebraker and Moore, 1996] Stonebraker, M. and Moore, D. (1996). *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann.

[Theobald et al., 2005a] Theobald, M., Schenkel, R., and Weikum, G. (2005a). Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR*, pages 242–249.

[Theobald et al., 2005b] Theobald, M., Schenkel, R., and Weikum, G. (2005b). An efficient and versatile query engine for TopX search. In *VLDB*, pages 625–636.

[Theobald et al., 2005c] Theobald, M., Schenkel, R., and Weikum, G. (2005c). Topx and xxl at inex 2005. In *INEX*, pages 282–295.

[Theobald et al., 2004] Theobald, M., Weikum, G., and Schenkel, R. (2004). Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659.

[TPC, 2005] TPC (2005). *TPC Benchmark H (Decision Support), Standard Specification*. TPC. Revision 2.3.0.

[Tsaparas et al., 2003] Tsaparas, P., Palpanas, T., Kotidis, Y., Koudas, N., and Srivastava, D. (2003). Ranked join indices. In *ICDE*, page 277.

[Turtle and Croft, 1990] Turtle, H. and Croft, W. B. (1990). Inference networks for document retrieval. In Vidick, J.-L., editor, *Proceedings of the 13th International Conference on Research and Development in Information Retrieval*, pages 1–24, New York. ACM.

[Ullman, 1988] Ullman, J. D. (1988). *Principles of database and knowledge-base systems, Vol. I*. Computer Science Press, Inc., New York, NY, USA.

[Ullman, 1989] Ullman, J. D. (1989). *Principles of Database and Knowledge-Base Systems: The New Technologies*, volume II. Computer Science Press, Rockville, MD.

[van Rijsbergen, 1986] van Rijsbergen, C. J. (1986). A non-classical logic for information retrieval. *The Computer Journal*, 29(6):481–485.

[Weigel et al., 2004] Weigel, F., Meuss, H., Bry, F., and Schulz, K. U. (2004). Content-Aware DataGuides: Interleaving IR and DB indexing techniques for efficient retrieval of textual XML data. In *ECIR*, pages 378–393.

[Weigel et al., 2005a] Weigel, F., Schulz, K. U., and Meuss, H. (2005a). The BIRD numbering scheme for XML and tree databases - deciding and reconstructing tree relations using efficient arithmetic operations. In *XSym*, pages 49–67.

[Weigel et al., 2005b] Weigel, F., Schulz, K. U., and Meuss, H. (2005b). Exploiting native XML indexing techniques for XML retrieval in relational database systems. In *WIDM*, pages 23–30.

[Witten et al., 1994] Witten, I., Moffat, A., and Bell, T. (1994). *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold.

[won Hwang and Chang, 2007] won Hwang, S. and Chang, K. C.-C. (2007). Probe minimization by schedule optimization: Supporting top-k queries with expensive predicates. *IEEE Trans. Knowl. Data Eng.*, 19(5):646–662.

[Wong and Yao, 1995] Wong, S. and Yao, Y. (1995). On modeling information retrieval with probabilistic inference. *ACM Transactions on Information Systems*, 13(1):38–68.

[Wu et al., 2008a] Wu, H., Kazai, G., and Roelleke, T. (2008a). Modelling anchor text retrieval in book search based on back-of-book index. In *SIRIG Workshop on Focused Retrieval*, pages 51–58.

[Wu et al., 2008b] Wu, H., Kazai, G., and Taylor, M. (2008b). Book search experiments: Investigating IR methods for the indexing and retrieval of books. In *ECIR*, pages 234–245.

[Xin et al., 2006a] Xin, D., Chen, C., and Han, J. (2006a). Towards robust indexing for ranked queries. In *VLDB*, pages 235–246.

[Xin et al., 2006b] Xin, D., Han, J., Cheng, H., and Li, X. (2006b). Answering top-k queries with multi-dimensional selections: The ranking cube approach. In *VLDB*, pages 463–475.

[Yan and Larson, 1995] Yan, W. P. and Larson, P.-Å. (1995). Eager aggregation and lazy aggregation. In *VLDB*, pages 345–357.

[Yang and Larson, 1987] Yang, H. Z. and Larson, P.-Å. (1987). Query transformation for PSJ-queries. In *VLDB*, pages 245–254.

[Yu et al., 1982] Yu, C., Lam, K., and Salton, G. (1982). Term weighting in information retrieval using the term precision model. *Journal of the ACM*, 29(1):152–170.

[Yu et al., 2005] Yu, H., Li, H.-G., Wu, P., Agrawal, D., and Abbadi, A. E. (2005). Efficient processing of distributed top- queries. In *DEXA*, pages 65–74.

[Zhai and Lafferty, 2002] Zhai, C. and Lafferty, J. D. (2002). Two-stage language models for information retrieval. In *ACM SIGIR*, pages 49–56.

[Zobel and Moffat, 2006] Zobel, J. and Moffat, A. (2006). Inverted files for text search engines. *ACM Comput. Surv.*, 38(2).

Appendix A

Getting Started with Birdie

A.1 Introduction

Birdie, named from the acronym of Bayesian (probabilistic) Information Retrieval and Database Integrated Engine, is an IR+DB prototype that is implemented in C#.

Birdie was at the beginning to be developed as a lightweight version of another IR+DB prototyping system HySpirit [Fuhr and Roelleke, 1998, Fuhr et al., 1998, Rölleke et al., 2001] for the purpose of studying optimization methods for probabilistic relational algebra (PRA) [Fuhr and Roelleke, 1998], in which only core functionality such as PRA execution engine had been redesigned and re-engineered, whereas many other constituent parts of HySpirit such as various abstraction layers (e.g. probabilistic Datalog (PD) [Fuhr, 2000], probabilistic four-valued Datalog (FVPD) [Fuhr and Roelleke, 1998] and probabilistic object-oriented logic (POOL) [Roelleke, 1999]) and data processing toolkits had not been included.

At the moment, several new functionality with respect to the techniques for improving the efficiency and scalability for IR+DB technology had been implemented into Birdie: firstly, a rule-based optimizer for PRA based on a scoring-driven optimization method discussed in the previous chapter (see Chapter 3); in addition, various indexers for building relational inverted indexes based on different RIX structures (see Chapter 5); moreover, several special physical operators for efficient probability estimation and aggregation had also been implemented; in addition, physical operators incorporating top-k mechanisms have been proposed, while currently the developing status is work in progress.

In the remainder of this chapter, we present a quick start guide for the readers who might be interested in Birdie, and some inside overview of the underlying architecture of the system.

A.2 Quick Start Guide

A.2.1 Commands

Currently three commands are provided for calling Birdie functionality, which are given in Table A.1. Command *setup* should be executed only once for setting up working directories such as knowledge-bases and temporary directory. An "INI file" is a program configuration file which specify program *parameters*. The option "-c" for "Birdrix" is a verification mode, which could be applied to verify correctness of RIX indexes after construction.

Executable	*Command Usage*	*Description*
Setup.exe	"Setup"	initialise working directories and configuration file, execute only once
Bird.exe	"Bird <INI file> [source file]"	calling for main engine functionality
Birdrix.exe	"Birdrix <INI file> <table name> [-c]"	a shell for examining RIX indexes

Table A.1: Birdie commands and usage

A.2.2 Setup and Configuration

System Configuration The system can be set up by called the "Setup" command, which creates a system configuration file named "Birdie.cfg". A configuration file looks like the follow:

```
[Paths]
home="F:/Demo/Birdie/MyDir"
kb_home="F:/Demo/Data/KBase"
tmp_home="F:/Demo/Data/Temp"

[Settings]
# 4 KB
page_size="4096"
# 32 KB
block_size="32768"
# 40 MB
buffer_size="41943040"
read_limit="1000"
schema_file_format="XML"
bulk_insert_chunk_limit="1000"
```

Users can modify the values to configure the parameters of the system.

Program Configuration In addition, an *INI* file is necessary for calling Bird engine or Birdrix shell, while a template of INI file looks like the follow:

```
[CONFIG FILE]
sys_config_file="system config file name"

[ACTION="create"|"insert"|"index"|"retrieve"]
action="one of the four actions"

[KNOWLEDGE BASE]
kb="a knowledge name"

[LANGUAGE="PRA"]
language="query language name"

[SOURCE FILE (optional)]
source_file="a source file name"
```

In default, the "sys_config_file" parameter is set to "Birdie.cfg". There four actions would be allowed and handled by Bird engine: "create" is used to create table or define indexes; "insert" is used to load data into tables; "index" is used to start indexing; and "retrieve" is used for running retrieval. At the moment, Birdie only supports PRA, but more query languages such as PD and PSQL may be supported in the future. Different source files could be used for different actions. For instance, users may put table definitions in a "create.txt" file, while a retrieval strategy in an "tfidf.txt" file.

A.2.3 Defining Knowledge-Bases

Firstly, users can create a new knowledge-base, table, or index, users can use "CREATE" clause. The syntax is given as the follow:

```
CREATE KB <knowledge-base name>;

CREATE TABLE <table_name> (attribute_name, data_type [...]);

CREATE INDEX <index_name> <index_type> ON <table_name> (attribute_name)
    [GIVEN EVIDENCE (attribute_name)];
```

Examples are given as the follows:

```
CREATE KB test;
CREATE TABLE trec3 (term VARCHAR, docid VARCHAR);
CREATE INDEX trec3_rxl RXL ON trec3(term) GIVEN EVIDENCE (docid);
CREATE INDEX trec3_rxs RXS ON trec3(term) GIVEN EVIDENCE (docid);
```

An INI file for creation may look like the follow:

```
[CONFIG FILE]
sys_config_file="birdie.cfg"
```

```
[ACTION="create"|"insert"|"index"|"retrieve"]
action="create"

[KNOWLEDGE BASE]
kb=test

[SOURCE FILE (optional)]
source_file="create.txt"
```

A.2.4 Loading Data

To load data into a table, users may use "BULK INSERT" clause, which syntax is given as the follow:

```
BULK INSERT <table_name> FROM <bulk_file_type> <bulk_file_path>;
```

For instance:

```
BULK INSERT trec3 FROM MDS 'F:/Collections/MDS/trec3/term.mds';
```

An INI file for bulk insertion may look like the follow, where "insertion_soft_limit" or "insertion_hard_limit" can be set to limit the number of tuples to be inserted. To set a soft limit would allow insertion to finish tuples with the same "evidence_attribute", whereas a hard limit would stop insertion as soon as limitation is reached.

```
[CONFIG FILE]
sys_config_file="birdie.cfg"

[ACTION="create"|"insert"|"index"|"retrieve"]
action="insert"

[KNOWLEDGE BASE]
kb=test

[INSERTION_LIMIT]
insertion_soft_limit=1000000
#insertion_hard_limit=1000000

[EVIDENCE_ATTRIBUTE]
evidence_attribute="docid"

[SOURCE FILE (optional)]
source_file="bulkins.txt"
```

A.2.5 Building Indexes

To start indexer and build a defined index, "BUILD INDEX" is the clause should be used. The syntax is given as the follow:

```
BUILD INDEX <table_name> <index_name>;
```

For example:

```
BUILD INDEX trec3 trec3_rxl;
BUILD INDEX trec3 trec3_rxs;
```

In addition, the INI file for indexing may look like the follow:

```
[CONFIG FILE]
sys_config_file="birdie.cfg"

[ACTION="create"|"insert"|"index"|"retrieve"]
action="index"

[KNOWLEDGE BASE]
kb=test

[SOURCE FILE (optional)]
source_file="buildidx.txt"
```

A.2.6 Running Queries

To run a PRA query, users should write retrieval strategies in a source file and execute Bird command as suggested in Table A.1, while an INI file for retrieval may look like the follow:

```
[CONFIG FILE]
sys_config_file="birdie.cfg"

[ACTION="create"|"insert"|"index"|"retrieve"]
action="retrieve"

[KNOWLEDGE BASE]
kb=test

[LANGUAGE="PRA"|"PD"|"PSQL"]
language="PRA"

[SOURCE FILE (optional)]
source_file="tfidf.txt"

[EXECUTION="true"|"false"|"on"|"off"]
execution="on"

[VERBOSE="true"|"false"|"on"|"off"]
verbose="on"

[OPTIMIZATION SWITCH="true"|"false"|"on"|"off"]
algebra_optimization="on"
score_driven_optimization="on"
cost_driven_optimization="on"

[BATCH_MODE SWITCH="true"|"false"|"on"|"off"]
```

```
batch_mode="on"
batch_queries_dir="F:/Documents/Data/queries"
batch_queries_format="MDS"

# batch queries must be tied to a table with MRT
# (Memory-Resident Table) type
batch_queries_tied_table="qterm"
```

If "execution" is set to "on", then a given strategies would be actually executed, otherwise the engine would only show generated and interpreted scoring expressions corresponding to PRA operators. If "batch_mode" is set to "on", then the engine may execute batch queries, while the parameter "batch_queries_tied_table" should be specified as well.

A.3 Inside Birdie

A.3.1 Storage Management

Figure A.1 illustrates the architecture of storage management in Birdie. In general, the storage management components in Birdie can be categorised into three layers: a logical layer and two physical layers.

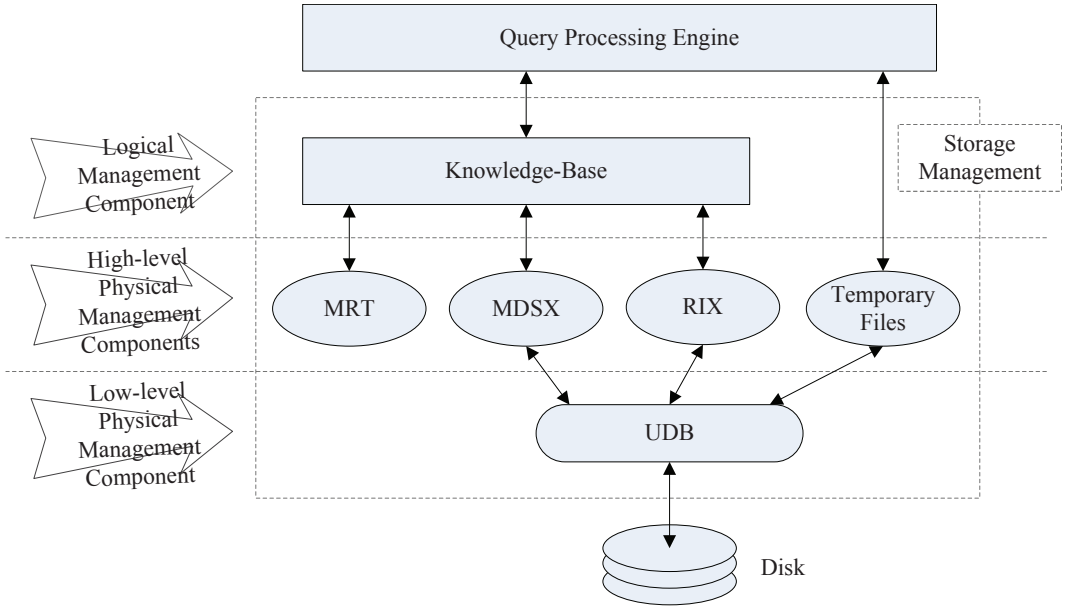


Figure A.1: Storage architecture of Birdie

In the logical layer, there is only one component named Knowledge-Base that manages the repositories of data. A repository is called a "knowledge-base", which contains the schema of tables and indexes and provides accessing entries of user-defined data to the query processing engine.

Moreover, the physical layers are subdivided into a high-level layer and a low-level layer. Firstly, the high-level physical layer consists of components for storing tables, indexes and temporary files. In particular, table could be stored externally or in-memory only. External storage of table uses Multi-Dimensional Space Extension (MDSX) file that is similar to the row-based tables in conventional databases; while in-memory storage of table uses Memory Residential Table (MRT), where a table is loaded at run-time only according to a predefined data schema. Secondly, the low-level physical layer contains a stored procedure named Universal Data Block (UDB), which is used by all high-level physical components that require external storage.

Next, we give some more details of the storage components.

Knowledge-Base A knowledge-base is where tables and indexes reside in, and the component manages these residents through data schema. A schema of a table or an index contains metadata which is a list of attributes that define the entity. The schema of a table is stored in an XML file using the same name as the table with a different file extension. For example, the schema of a table "trec3" for collection TREC-3 is could be:

```
<table name="trec3" path="F:/Demo/Data/KBase/test" type="MDSX" weight="False">
<attribute name="term" type="VARCHAR" />
<attribute name="docid" type="VARCHAR" />
<index name="trec3_rxs" type="RXS" primaryKey="term" evidentKey="docid" />
</table>
```

Multi-Dimensional Space Extension An Multi-Dimensional Space Extension (MDSX) file is an enhanced version of text MDS file, where text MDS files are used to store materialised tables in HySpirit [Rölleke et al., 2001]. A text MDS file uses row-based format to store tuples, where a row consists of a tuple and the tuple's score. The MDS format treats the score as a natural feature of a tuple where it coexists with the tuple, which is different from conventional database systems where the score is treated as a normal user-defined attribute. However, since the MDS format does not specify physical stored procedures, and MDS files are stored as delimiter-separated text files, which lacks the capability for efficient data access. For example, reading tuples from text MDS file always involve parsing; random access tuples from text MDS file is impossible; it is difficult to apply compression for text MDS file.

As a consequence, the shortcomings of text MDS file motivate us to develop a much more I/O efficient storage method, which inherits the logical design of MDS format while improves the physical stored procedures. MDSX enhances the original MDS from the following aspects:

- It stores data in binary format and parsing is not needed;

- It supports multiple basic data types including text types and numeric types, whereas MDS is type-insensitive and treats data as text strings;
- It supports random access for tuples;
- Compression methods could be applied on the text fields of tuples.

Memory Residential Table Memory Residential Table (MRT) is used to store dynamic tables that are loaded only at run-time. MRT is useful when a retrieval strategy is needed to be executed repeatedly with different query terms, for instance, running queries in batch mode.

Defining a table to be MRT is similar as defining an MDSX table, where a schema file would be generated as well. However, there are two differences: 1) a MRT table does not have an external storage in the knowledge-base, the tuples of the table would be dynamically loaded at run-time; and 2) indexes cannot be created for a MRT table. Therefore, MRT table is only used for tiny tables that can be entirely resided in memory.

Relational Inverted Index (RIX) More details refer to Chapter 5.

Temporary Files Temporary files are used by query execution engine for various purposes, for instance, to be used as buffers that materialise intermediate results, or to be used by certain algorithms such as hybrid hash join and aggregation.

Universal Data Block Universal Data Block (UDB) provides the basic I/O functionality for storing and retrieving data from external storage. A UDB includes a header area and a data area, and schema is required to access the data area. The header area contains a number of meta information about the block, such as the number of data fields and the length of the block; while the data area contains varied length n-tuples, where n could be one or many. In short, the UDB provides a flexible format and common interfaces for high-level storage types including MDSX, RIX and temporary files.

A.3.2 Query Language

At the moment, Birdie only supports a variant of probabilistic relational algebra (PRA) as query language. For example, the PRA expressions for modelling *tf-idf* model and language modelling in Section 3.5 are given in below.

The follow is the retrieval strategy for *tf-idf* model:

```
# tf-idf
# p_C_t_d
tfCollSpace = BAYES DISJOINT [$2](trec3);
p_C_t_d = PROJECT DISJOINT [$1,$4](
    JOIN[$1=$1](qterm, PROJET DISJOINT [$1,$2](tfCollSpace)));

# idf
{ idf = log(1 / df_t) };

dColl = PROJECT DISTINCT [](trec3);
docSpace = PROJECT DISTNCT [$2](trec3);
wDocSpace = BAYES DISJOINT [](docSpace);

termDoc = PROJECT [$1,$2] (
  JOIN [$2=$1](dColl, wDocSpace));

idf_t = PROJECT idf [$3](
  JOIN [$1=$1](qterm, PROJECT DISJOINT [$1](termDoc)));

retrieve = PROJECT DISJOINT [$2](
  JOIN [$1=$1](idf_t, p_C_t_d));

?- retrieve;
```

The follow is the retrieval strategy for language modelling:

```
# lm - language modelling
{
  lambda = 0.8,
  lm = log (1 + (lambda / (1 - lambda)) * (p_q_d / p_q_c))
};

p_C_t = PROJECT DISJOINT [$1](BAYES [](trec3));
p_q_c = PROJECT [$1](JOIN [$1=$1](qterm, p_C_t));

p_C_t_d = PROJECT DISJOINT [$1,$2](BAYES [$2](trec3));
p_q_d = PROJECT [$1,$4](JOIN [$1=$1](qterm, p_C_t_d));

retrieve = PROJECT DISJOINT [$3](JOIN lm [$1=$1](p_q_c, p_q_d));

?- retrieve;
```

A.3.3 Query Execution Engine

For implementing the query execution engine, we adopted pipelined execution engine architecture such as in conventional database (e.g. see [Graefe, 1993]); in addition, we also developed special probability estimator for the Bayes operator in logical PRA, and probability aggregators for other corresponding logical PRA operators.

A.3.4 Query Optimizer

Semantic graphs are generated by query optimizer to analyse the semantics of scoring expressions. For example, the semantic graph for unit fraction (a fraction which has constant one in numerator) is illustrated in Figure A.2.

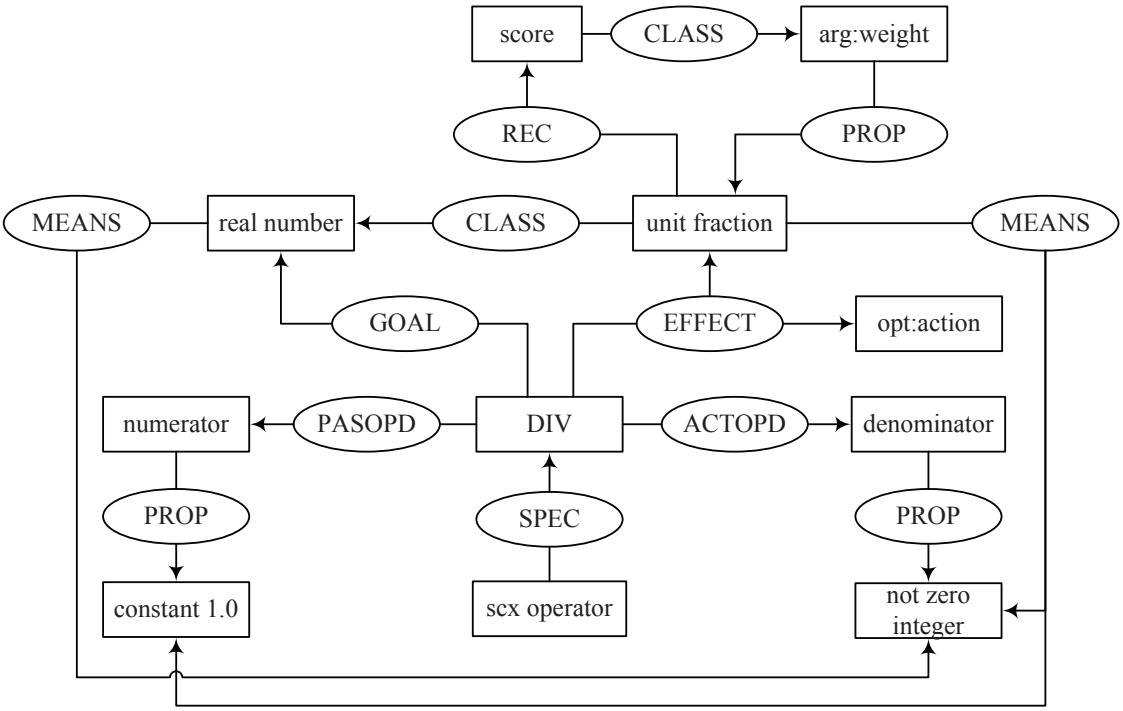


Figure A.2: Semantic graph formed for unit fraction

On the other hand, the semantic graph can be represented in XML so that to be handled by analysis program. For instance, the above graph can be represented in the following XML:

```
<scxsg>
  <scxopr>
    <spec>arg:operation</spec>
    <pasopd>opt:operand</pasopd>
    <actopd>arg:operand</actopd>
    <goal>arg:goal</goal>
    <effect>arg:result</effect>
  </scxopr>
  <score>
    <class>arg:weight</class>
    <prop>arg:result</prop>
  </score>
  <arg>
    <operation>
      <name>DIV</name>
    </operation>
    <operand>
      <name>denominator</name>
```

```
      <prop>not zero integer</prop>
    </operand>
    <goal>
      <name>real number</name>
      <means>not zero</means>
    </goal>
    <result>
      <name>unit fraction</name>
      <class>arg:goal</class>
      <rec>score</rec>
    </result>
    <weight>
      <name>probability</name>
    </weight>
  </arg>
  <opt>
    <operand>
      <name>numerator</name>
      <prop>constant integer 1.0</prop>
    </operand>
    <action></action>
  </opt>
</scxsg>
```

Appendix B

Full MagazineCorpus Table

	MagazineCorpus									
TID	*Term*	*DocID*	*ChapID*	*TitleID*	*SecID*	*AuthID*	*ParaID*	*LinkID*	*RefDocID*	*Font*
0	fortune	1	null	1	null	null	null	null	null	null
1	13	1	1	1	null	null	null	null	null	bold
2	test	1	1	1	null	null	null	null	null	bold
3	drive	1	1	1	null	null	null	null	null	bold
4	hybrid	1	1	null	1	null	1	1	2	null
5	wars	1	1	null	1	null	1	1	2	null
6	heat	1	1	null	1	null	1	null	null	null
7	honda	1	1	null	1	null	1	null	null	null
8	pushes	1	1	null	1	null	1	null	null	null
9	fray	1	1	null	1	null	1	null	null	null
10	gas-electric	1	1	null	1	null	1	null	null	null
11	insight	1	1	null	1	null	1	null	null	null
12	alex	1	1	null	null	1	null	null	null	italic
13	taylor	1	1	null	null	1	null	null	null	italic
14	iii	1	1	null	null	1	null	null	null	italic
15	46	1	2	1	null	null	null	null	null	bold
16	bavarias	1	2	1	null	null	null	null	null	bold
17	next	1	2	1	null	null	null	null	null	bold
18	top	1	2	1	null	null	null	null	null	bold
19	model	1	2	1	null	null	null	null	null	bold
20	new	1	2	null	1	null	1	null	null	null
21	gt	1	2	null	1	null	1	null	null	null
22	bmw	1	2	null	1	null	1	null	null	null
23	hopes	1	2	null	1	null	1	null	null	null
24	expand	1	2	null	1	null	1	null	null	null
25	definition	1	2	null	1	null	1	null	null	null
26	luxury	1	2	null	1	null	1	null	null	null
27	touring	1	2	null	1	null	1	null	null	null
28	car	1	2	null	1	null	1	null	null	null
29	down	1	2	null	1	null	2	null	null	null

(a) Table MagazineCorpus Part One

Table MagazineCorpus continues

	MagazineCorpus									
TID	*Term*	*DocID*	*ChapID*	*TitleID*	*SecID*	*AuthID*	*ParaID*	*LinkID*	*RefDocID*	*Font*
30	road	1	2	null	1	null	2	null	null	null
31	figure	1	2	null	1	null	2	null	null	null
32	out	1	2	null	1	null	2	null	null	null
33	consumers	1	2	null	1	null	2	null	null	null
34	want	1	2	null	1	null	2	null	null	null
35	premium	1	2	null	1	null	2	null	null	null
36	green	1	2	null	1	null	2	null	null	null
37	automobile	1	2	null	1	null	2	null	null	null
38	alex	1	2	null	null	1	null	null	null	italic
39	taylor	1	2	null	null	1	null	null	null	italic
40	iii	1	2	null	null	1	null	null	null	italic
41	time	2	null	1	null	null	null	null	null	null
42	hybrid	2	1	1	null	null	null	null	null	bold
43	hybrid	2	1	1	null	null	null	null	null	bold
44	cars	2	1	1	null	null	null	null	null	bold
45	future	2	1	1	null	null	null	null	null	bold
46	compare	2	1	1	null	null	null	null	null	bold
47	prius	2	1	1	1	null	null	null	null	smallcaps
48	toyota	2	1	1	1	null	null	null	null	smallcaps
49	original	2	1	null	1	null	1	null	null	null
50	hybrid	2	1	null	1	null	1	null	null	null
51	uses	2	1	null	1	null	1	null	null	null
52	gas	2	1	null	1	null	1	null	null	null
53	electric	2	1	null	1	null	1	null	null	null
54	engines	2	1	null	1	null	1	null	null	null
55	best	2	1	null	1	null	1	null	null	null
56	fuel	2	1	null	1	null	1	null	null	null
57	economy	2	1	null	1	null	1	null	null	null
58	car	2	1	null	1	null	1	null	null	null
59	usa	2	1	null	1	null	1	null	null	null
60	today	2	1	null	1	null	1	null	null	null
61	costs	2	1	null	1	null	1	null	null	null
62	volts	2	1	null	1	null	1	null	null	null
63	target	2	1	null	1	null	1	null	null	null
64	price	2	1	null	1	null	1	null	null	null
65	next	2	1	2	null	null	null	null	null	bold
66	future	2	1	null	1	null	2	null	null	null
67	versions	2	1	null	1	null	2	null	null	null
68	plug-ins	2	1	null	1	null	2	null	null	null
69	unlikely	2	1	null	1	null	2	null	null	null
70	volts	2	1	null	1	null	2	null	null	null
71	all-electric	2	1	null	1	null	2	null	null	null
72	range	2	1	null	1	null	2	null	null	null
73	volt	2	1	1	2	null	null	null	null	bold
74	general	2	1	1	2	null	null	null	null	bold
75	motors	2	1	1	2	null	null	null	null	bold
76	volt	2	1	null	2	null	1	null	null	null
77	extended-range	2	1	null	2	null	1	null	null	null
78	electric	2	1	null	2	null	1	null	null	null
79	vehicle	2	1	null	2	null	1	null	null	null
80	powered	2	1	null	2	null	1	null	null	null

(a) Table MagazineCorpos Part Two

Table MagazineCorpus continues

	MagazineCorpus									
TID	*Term*	*DocID*	*ChapID*	*TitleID*	*SecID*	*AuthID*	*ParaID*	*LinkID*	*RefDocID*	*Font*
81	electricity	2	1	null	2	null	1	null	null	null
82	amounts	2	1	null	2	null	1	null	null	null
83	gasoline-fueled	2	1	null	2	null	1	null	null	null
84	electric	2	1	null	2	null	1	null	null	null
85	generator	2	1	null	2	null	1	null	null	null
86	longer	2	1	null	2	null	1	null	null	null
87	drivers	2	1	null	2	null	1	null	null	null
88	question	2	1	2	2	null	null	null	null	smallcaps
89	cost	2	1	2	2	null	null	null	null	smallcaps
90	critics	2	1	null	2	null	2	null	null	null
91	love	2	1	null	2	null	2	null	null	null
92	volt	2	1	null	2	null	2	null	null	null
93	technology	2	1	null	2	null	2	null	null	null
94	wonder	2	1	null	2	null	2	null	null	null
95	car	2	1	null	2	null	2	null	null	null
96	affordable	2	1	null	2	null	2	null	null	null
97	bryan	2	1	null	null	1	null	null	null	italic
98	walsh	2	1	null	null	1	null	null	null	italic

(a) Table MagazineCorpus Part Three

Table B.1: An example MDSX table *MagazineCorpus* for a toy magazine corpus

Glossary

ACID	Atomicity, Consistency, Isolation, Durability. These are criteria for transaction-based processing in databases, 12
ADT	Abstract Data Type, 167
BNF	Backus-Naur Form, 67
CA	Combined Algorithm, 128
DBMS	Database Management System, 12
FA	Fagin's Algorithm, 126
FVPD	Four-Valued Probabilistic Datalog, 238
INEX	INitiative for the Evaluation of XML Retrieval, 44
IPT	Ideal Performances Tradeoff, 141
MDSX	Multi-Dimensional Space Extension, 244
MRT	Memory Residential Table, 245
NRA	No Random Access, 128
PA	Probability Aggregator, 138
PD	Probabilistic Datalog, 238
PE	Probability Estimator, 137

POOL Probabilistic Object-Oriented Logic, 238

PRA Probabilistic Relational Algebra, 33

RDBMS Relational Database Management System, 27

RIX Relational Inverted Index, 158

RixExt Extended version of RIX, 162

RixLite Lite version of RIX, 161

RixStd Standard version of RIX, 162

SCX Scoring Expression, 53

SEA Select-Estimate-Aggregate, 135

SPJ Select-Project-Join, 135

TA Threshold Algorithm, 125

TID Tuple Identifier or Tuple ID, 159

TIP Top-*k* Incorporated Pipeline, 134

TREC Text REtrieval Conference, 44

UDB Universal Data Block, 245

XML Extensible Markup Language, 23

XRA Random Index Access, 137

XS Index Scan, 137

Index